Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™



Patterns, Principles, and Practices of Domain-Driven Design

Scott Millett with Nick Tune

PATTERNS, PRINCIPLES, AND PRACTICES OF DOMAIN-DRIVEN DESIGN

INTRODUCTION XXXV

▶ PART I THE PRINCIPLES AND PRACTICES OF DOMAIN-DRIVEN DESIGN

CHAPTER 1 What Is Domain-Driven Design? 3

CHAPTER 2 Distilling the Problem Domain 15

CHAPTER 3 Focusing on the Core Domain 31

CHAPTER 4 Model-Driven Design 41

CHAPTER 5 Domain Model Implementation Patterns 59

CHAPTER 6 Maintaining the Integrity of Domain Models with Bounded Contexts 73

CHAPTER 7 Context Mapping 91

CHAPTER 8 Application Architecture 105

CHAPTER 9 Common Problems for Teams Starting Out with Domain-Driven Design 121

CHAPTER 10 Applying the Principles, Practices, and Patterns of DDD 131

▶ PART II STRATEGIC PATTERNS: COMMUNICATING BETWEEN BOUNDED CONTEXTS

CHAPTER 11 Introduction to Bounded Context Integration 151

CHAPTER 12 Integrating via Messaging 181

CHAPTER 13 Integrating via HTTP with RPC and REST 245

▶ PART III TACTICAL PATTERNS: CREATING EFFECTIVE DOMAIN MODELS

CHAPTER 14 Introducing the Domain Modeling Building Blocks. 309

CHAPTER 15 Value Objects 329

CHAPTER 16 Entities 361

Continues

CHAPTER 17 Domain Services 389
CHAPTER 18 Domain Events 405
CHAPTER 19 Aggregates 427
CHAPTER 20 Factories 469
CHAPTER 21 Repositories 479
CHAPTER 22 Event Sourcing 595

▶ **PART IV DESIGN PATTERNS FOR EFFECTIVE APPLICATIONS**

CHAPTER 23 Architecting Application User Interfaces 645
CHAPTER 24 CQRS: An Architecture of a Bounded Context 669
CHAPTER 25 Commands: Application Service Patterns for Processing Business Use Cases 687
CHAPTER 26 Queries: Domain Reporting 713

INDEX **737**

Patterns, Principles, and Practices of Domain-Driven Design

Patterns, Principles, and Practices of Domain-Driven Design

Scott Millett

Nick Tune



Patterns, Principles, and Practices of Domain-Driven Design

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2015 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-118-71470-6
ISBN: 978-1-118-71465-2 (ebk)
ISBN: 978-1-118-71469-0 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permissions.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at http://booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2014951018

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

For my darling buds, Primrose and Albert.

—Scott Millett

ABOUT THE AUTHOR

SCOTT MILLETT is the Director of IT for Iglu.com and has been working with .NET since version 1.0. He was awarded the ASP.NET MVP in 2010 and 2011. He is also the author of *Professional ASP.NET Design Patterns* and *Professional Enterprise .NET*. If you would like to contact Scott about DDD or working at Iglu, feel free to write to him at `scott@elbandit.co.uk`, by giving him a tweet `@ScottMillett`, or becoming friends via `https://www.linkedin.com/in/scottmillett`.

ABOUT THE CONTRIBUTING AUTHOR

NICK TUNE is passionate about solving business problems, building ambitious products, and constantly learning. Being a software developer really is his dream job. His career highlight so far was working at 7digital, where he was part of self-organizing, business-focused teams that deployed to production up to 25 times per day. His future ambitions are to work on exciting new products, with passionate people, and continually become a more complete problem solver.

You can learn more about Nick and his views on software development, software delivery, and his favorite technologies on his website (`www.ntcoding.co.uk`) and Twitter (`@ntcoding`).

ABOUT THE TECHNICAL EDITOR

ANTONY DENYER works as a developer, consultant, and coach and has been developing software professionally since 2004. He has worked on various projects that have effectively used DDD concepts and practices. More recently, he has been advocating the use of CQRS and REST in the majority of his projects. You can reach him via e-mail at `antonydenyer.co.uk`, and he tweets from `@tonydenyer`.

CREDITS

PROJECT EDITOR
Rosemarie Graham

TECHNICAL EDITOR
Antony Denyer

PRODUCTION EDITOR
Christine O'Connor

COPY EDITOR
Karen Gill

MANAGER OF CONTENT DEVELOPMENT AND ASSEMBLY
Mary Beth Wakefield

MARKETING DIRECTOR
David Mayhew

MARKETING MANAGER
Carrie Sherrill

PROFESSIONAL TECHNOLOGY & STRATEGY DIRECTOR
Barry Pruett

BUSINESS MANAGER
Amy Knies

ASSOCIATE PUBLISHER
Jim Minatel

PROJECT COORDINATOR, COVER
Brent Savage

PROOFREADER
Jenn Bennett, Word One

INDEXER
Johnna VanHoose Dinse

COVER DESIGNER
Wiley

COVER IMAGE
@iStockphoto.com/andynwt

ACKNOWLEDGMENTS

FIRSTLY I WOULD LIKE to give a massive thanks to Nick Tune for agreeing to help me out with this project and contributing greatly to many of the chapters. I would also like to thank Rosemarie Graham, Jim Minatel, and all those at Wrox who have helped to create this book. Thanks as well to Antony Denyer who did a sterling job as the technical editor. Lastly, many thanks to Isabel Mack for the grammar pointers and early feedback of the Leanpub draft.

CONTENTS

INTRODUCTION *xxxv*

PART I: THE PRINCIPLES AND PRACTICES OF DOMAIN-DRIVEN DESIGN

CHAPTER 1: WHAT IS DOMAIN-DRIVEN DESIGN? 3

The Challenges of Creating Software for Complex Problem Domains 4
Code Created Without a Common Language 4
A Lack of Organization 5
The Ball of Mud Pattern Stifles Development 5
A Lack of Focus on the Problem Domain 6
How the Patterns of Domain-Driven Design Manage Complexity 6
The Strategic Patterns of DDD 6
Distilling the Problem Domain to Reveal What Is Important 7
Creating a Model to Solve Domain Problems 7
Using a Shared Language to Enable Modeling Collaboration 7
Isolate Models from Ambiguity and Corruption 8
Understanding the Relationships between Contexts 9
The Tactical Patterns of DDD 9
The Problem Space and the Solution Space 9
The Practices and Principles of Domain-Driven Design 11
Focusing on the Core Domain 11
Learning through Collaboration 11
Creating Models through Exploration and Experimentation 11
Communication 11
Understanding the Applicability of a Model 12
Constantly Evolving the Model 12
Popular Misconceptions of Domain-Driven Design 12
Tactical Patterns Are Key to DDD 12
DDD Is a Framework 13
DDD Is a Silver Bullet 13
The Salient Points 13

CHAPTER 2: DISTILLING THE PROBLEM DOMAIN 15

Knowledge Crunching and Collaboration 15
Reaching a Shared Understanding through a Shared Language 16
The Importance of Domain Knowledge 17
The Role of Business Analysts 17
An Ongoing Process 17
Gaining Domain Insight with Domain Experts 18
Domain Experts vs Stakeholders 18
Deeper Understanding for the Business 19
Engaging with Your Domain Experts 19
Patterns for Effective Knowledge Crunching 19
Focus on the Most Interesting Conversations 19
Start from the Use Cases 20
Ask Powerful Questions 20
Sketching 20
Class Responsibility Collaboration Cards 21
Defer the Naming of Concepts in Your Model 21
Behavior-Driven Development 22
Rapid Prototyping 23
Look at Paper-Based Systems 24
Look For Existing Models 24
Understanding Intent 24
Event Storming 25
Impact Mapping 25
Understanding the Business Model 27
Deliberate Discovery 28
Model Exploration Whirlpool 29
The Salient Points 29

CHAPTER 3: FOCUSING ON THE CORE DOMAIN 31

Why Decompose a Problem Domain? 31
How to Capture the Essence of the Problem 32
Look Beyond Requirements 32
Capture the Domain Vision for a Shared Understanding of What Is Core 32
How to Focus on the Core Problem 33
Distilling a Problem Domain 34
Core Domains 35

Treat Your Core Domain as a Product Rather than a Project 36
Generic Domains 37
Supporting Domains 37
How Subdomains Shape a Solution 37
Not All Parts of a System will be Well Designed 37
Focus on Clean Boundaries over Perfect Models 38
The Core Domain Doesn't Always Have to Be Perfect the First Time 39
Build Subdomains for Replacement Rather than Reuse 39
What if You Have no Core Domain? 39
The Salient Points 40

CHAPTER 4: MODEL-DRIVEN DESIGN 41

What Is a Domain Model? 42
The Domain versus the Domain Model 42
The Analysis Model 43
The Code Model 43
The Code Model Is the Primary Expression of the Domain Model 44
Model-Driven Design 44
The Challenges with Upfront Design 44
Team Modeling 45
Using a Ubiquitous Language to Bind the Analysis to the Code Model 47
A Language Will Outlive Your Software 47
The Language of the Business 48
Translation between the Developers and the Business 48
Collaborating on a Ubiquitous Language 48
Carving Out a Language by Working with Concrete Examples 49
Teach Your Domain Experts to Focus on the Problem and Not Jump to a Solution 50
Best Practices for Shaping the Language 51
How to Create Effective Domain Models 52
Don't Let the Truth Get in the Way of a Good Model 52
Model Only What Is Relevant 54
Domain Models Are Temporarily Useful 54
Be Explicit with Terminology 54
Limit Your Abstractions 54
Focus Your Code at the Right Level of Abstraction 55
Abstract Behavior Not Implementations 55

Implement the Model in Code Early and Often 56
Don't Stop at the First Good Idea 56
When to Apply Model-Driven Design 56
If It's Not Worth the Effort Don't Try and Model It 56
Focus on the Core Domain 57
The Salient Points 57

CHAPTER 5: DOMAIN MODEL IMPLEMENTATION PATTERNS 59

The Domain Layer 60
Domain Model Implementation Patterns 60
Domain Model 62
Transaction Script 65
Table Module 67
Active Record 67
Anemic Domain Model 67
Anemic Domain Model and Functional Programming 68
The Salient Points 71

CHAPTER 6: MAINTAINING THE INTEGRITY OF DOMAIN MODELS WITH BOUNDED CONTEXTS 73

The Challenges of a Single Model 74
A Model Can Grow in Complexity 74
Multiple Teams Working on a Single Model 74
Ambiguity in the Language of the Model 75
The Applicability of a Domain Concept 76
Integration with Legacy Code or Third Party Code 78
Your Domain Model Is not Your Enterprise Model 79
Use Bounded Contexts to Divide and Conquer a Large Model 79
Defining a Model's Boundary 82
Define Boundaries around Language 82
Align to Business Capabilities 83
Create Contexts around Teams 83
Try to Retain Some Communication between Teams 84
Context Game 85
The Difference between a Subdomain and a Bounded Context 85
Implementing Bounded Contexts 85
The Salient Points 89

CHAPTER 7: CONTEXT MAPPING 91

A Reality Map 92
The Technical Reality 92
The Organizational Reality 93
Mapping a Relevant Reality 94
X Marks the Spot of the Core Domain 94
Recognising the Relationships between Bounded Contexts 95
Anticorruption Layer 95
Shared Kernel 96
Open Host Service 97
Separate Ways 97
Partnership 98
An Upstream/Downstream Relationship 98
Customer-Supplier 99
Conformist 100
Communicating the Context Map 100
The Strategic Importance of Context Maps 101
Retaining Integrity 101
The Basis for a Plan of Attack 101
Understanding Ownership and Responsibility 101
Revealing Areas of Confusion in Business Work Flow 102
Identifying Nontechnical Obstacles 102
Encourages Good Communication 102
Helps On-Board New Starters 102
The Salient Points 103

CHAPTER 8: APPLICATION ARCHITECTURE 105

Application Architecture 105
Separating the Concerns of Your Application 106
Abstraction from the Complexities of the Domain 106
A Layered Architecture 106
Dependency Inversion 107
The Domain Layer 107
The Application Service Layer 108
The Infrastructural Layers 108
Communication Across Layers 108
Testing in Isolation 109
Don’t Share Data Schema between Bounded Contexts 109

Application Architectures versus Architectures for Bounded Contexts 111
Application Services 112
Application Logic versus Domain Logic 114
Defining and Exposing Capabilities 114
Business Use Case Coordination 115
Application Services Represent Use Cases, Not Create, Read, Update, and Delete 115
Domain Layer As an Implementation Detail 115
Domain Reporting 116
Read Models versus Transactional Models 116
Application Clients 117
The Salient Points 120

CHAPTER 9: COMMON PROBLEMS FOR TEAMS STARTING OUT WITH DOMAIN-DRIVEN DESIGN 121

Overemphasizing the Importance of Tactical Patterns 122
Using the Same Architecture for All Bounded Contexts 122
Striving for Tactical Pattern Perfection 122
Mistaking the Building Blocks for the Value of DDD 123
Focusing on Code Rather Than the Principles of DDD 123
Missing the Real Value of DDD: Collaboration, Communication, and Context 124
Producing a Big Ball of Mud Due to Underestimating the Importance of Context 124
Causing Ambiguity and Misinterpretations by Failing to Create a UL 125
Designing Technical-Focused Solutions Due to a Lack of Collaboration 125
Spending Too Much Time on What's Not Important 126
Making Simple Problems Complex 126
Applying DDD Principles to a Trivial Domain with Little Business Expectation 126
Disregarding CRUD as an Antipattern 127
Using the Domain Model Pattern for Every Bounded Context 127
Ask Yourself: Is It Worth This Extra Complexity? 127
Underestimating the Cost of Applying DDD 127
Trying to Succeed Without a Motivated and Focused Team 128
Attempting Collaboration When a Domain Expert Is Not Behind the Project 128
Learning in a Noniterative Development Methodology 128

Applying DDD to Every Problem 129
Sacrificing Pragmatism for Needless Purity 129
Wasted Effort by Seeking Validation 129
Always Striving for Beautiful Code 130
DDD Is About Providing Value 130
The Salient Points 130

CHAPTER 10: APPLYING THE PRINCIPLES, PRACTICES, AND PATTERNS OF DDD 131

Selling DDD 132
Educating Your Team 132
Speaking to Your Business 132
Applying the Principles of DDD 133
Understand the Vision 133
Capture the Required Behaviors 134
Distilling the Problem Space 134
Focus on What Is Important 134
Understand the Reality of the Landscape 135
Modeling a Solution 135
All Problems Are Not Created Equal 136
Engaging with an Expert 136
Select a Behavior and Model Around a Concrete Scenario 137
Collaborate with the Domain Expert on the Most Interesting Parts 137
Evolve UL to Remove Ambiguity 138
Throw Away Your First Model, and Your Second 138
Implement the Model in Code 139
Creating a Domain Model 139
Keep the Solution Simple and Your Code Boring 139
Carve Out an Area of Safety 140
Integrate the Model Early and Often 140
Nontechnical Refactoring 140
Decompose Your Solution Space 140
Rinse and Repeat 141
Exploration and Experimentation 142
Challenge Your Assumptions 142
Modeling Is a Continuous Activity 142
There Are No Wrong Models 142
Supple Code Aids Discovery 143
Making the Implicit Explicit 143

Tackling Ambiguity 144
Give Things a Name 145
A Problem Solver First, A Technologist Second 146
Don't Solve All the Problems 146
How Do I Know That I Am Doing It Right? 146
Good Is Good Enough 147
Practice, Practice, Practice 147
The Salient Points 147

PART II: STRATEGIC PATTERNS: COMMUNICATING BETWEEN BOUNDED CONTEXTS

CHAPTER 11: INTRODUCTION TO BOUNDED CONTEXT INTEGRATION 151

How to Integrate Bounded Contexts 152
Bounded Contexts Are Autonomous 153
The Challenges of Integrating Bounded Contexts at the Code Level 153
Multiple Bounded Contexts Exist within a Solution 153
Namespaces or Projects to Keep Bounded Contexts Separate 154
Integrating via the Database 155
Multiple Teams Working in a Single Codebase 156
Models Blur 156
Use Physical Boundaries to Enforce Clean Models 157
Integrating with Legacy Systems 158
Bubble Context 158
Autonomous Bubble Context 158
Exposing Legacy Systems as Services 160
Integrating Distributed Bounded Contexts 161
Integration Strategies for Distributed Bounded Contexts 161
Database Integration 162
Flat File Integration 163
RPC 164
Messaging 165
REST 165
The Challenges of DDD with Distributed Systems 165
The Problem with RPC 166
RPC Is Harder to Make Resilient 167
RPC Costs More to Scale 167
RPC Involves Tight Coupling 168

Distributed Transactions Hurt Scalability and Reliability 169
Bounded Contexts Don't Have to Be Consistent with Each Other 169
Eventual Consistency 169
Event-Driven Reactive DDD 170
Demonstrating the Resilience and Scalability of Reactive Solutions 171
Challenges and Trade-Offs of Asynchronous Messaging 173
Is RPC Still Relevant? 173
SOA and Reactive DDD 174
View Your Bounded Contexts as SOA Services 175
Decompose Bounded Contexts into Business Components 175
Decompose Business Components into Components 176
Going Even Further with Micro Service Architecture 178
The Salient Points 180

CHAPTER 12: INTEGRATING VIA MESSAGING 181

Messaging Fundamentals 182
Message Bus 182
Reliable Messaging 184
Store-and-Forward 184
Commands and Events 185
Eventual Consistency 186
Building an E-Commerce Application with NServiceBus 186
Designing the System 187
Domain-Driven Design 187
Containers Diagrams 188
Evolutionary Architecture 191
Sending Commands from a Web Application 192
Creating a Web Application to Send Messages with NServiceBus 192
Sending Commands 197
Handling Commands and Publishing Events 200
Creating an NServiceBus Server to Handle Commands 200
Configuring the Solution for Testing and Debugging 201
Publishing Events 204
Subscribing to Events 206
Making External HTTP Calls Reliable with Messaging Gateways 208
Messaging Gateways Improve Fault Tolerance 208
Implementing a Messaging Gateway 209
Controlling Message Retries 212
Eventual Consistency in Practice 215
Dealing with Inconsistency 215

Rolling Forward into New States 215
Bounded Contexts Store All the Data They Need Locally 216
Storage Is Cheap—Keep a Local Copy 217
Common Data Duplication Concerns 223
Pulling It All Together in the UI 224
Business Components Need Their Own APIs 225
Be Wary of Server-Side Orchestration 226
UI Composition with AJAX Data 226
UI Composition with AJAX HTML 226
Sharing Your APIs with the Outside World 227
Maintaining a Messaging Application 227
Message Versioning 228
Backward-Compatible Message Versioning 228
Handling Versioning with NServiceBus's Polymorphic Handlers 229
Monitoring and Scaling 233
Monitoring Errors 233
Monitoring SLAs 234
Scaling Out 235
Integrating a Bounded Context with Mass Transit 235
Messaging Bridge 236
Mass Transit 236
Installing and Configuring Mass Transit 236
Declaring Messages for Use by Mass Transit 238
Creating a Message Handler 239
Subscribing to Events 239
Linking the Systems with a Messaging Bridge 240
Publishing Events 242
Testing It Out 243
Where to Learn More about Mass Transit 243
The Salient Points 243

CHAPTER 13: INTEGRATING VIA HTTP WITH RPC AND REST 245

Why Prefer HTTP? 247
No Platform Coupling 247
Everyone Understands HTTP 247
Lots of Mature Tooling and Libraries 247
Dogfooding Your APIs 247
RPC 248
Implementing RPC over HTTP 248
SOAP 249

Plain XML or JSON: The Modern Approach to RPC 259
Choosing a Flavor of RPC 263
REST 264
Demystifying REST 264
Resources 264
Hypermedia 265
Statelessness 265
REST Fully Embraces HTTP 266
What REST Is Not 267
REST for Bounded Context Integration 268
Designing for REST 268
Building Event-Driven REST Systems with ASP.NET Web API 273
Maintaining REST Applications 303
Versioning 303
Monitoring and Metrics 303
Drawbacks with REST for Bounded Context Integration 304
Less Fault Tolerance Out of the Box 304
Eventual Consistency 304
The Salient Points 305

PART III: TACTICAL PATTERNS: CREATING EFFECTIVE DOMAIN MODELS

CHAPTER 14: INTRODUCING THE DOMAIN MODELING BUILDING BLOCKS 309

Tactical Patterns 310
Patterns to Model Your Domain 310
Entities 310
Value Objects 314
Domain Services 317
Modules 318
Lifecycle Patterns 318
Aggregates 318
Factories 322
Repositories 323
Emerging Patterns 324
Domain Events 324
Event Sourcing 326
The Salient Points 327

CHAPTER 15: VALUE OBJECTS 329

When to Use a Value Object 330
Representing a Descriptive, Identity-Less Concept 330
Enhancing Explicitness 331
Defining Characteristics 333
Identity-Less 333
Attribute-Based Equality 333
Behavior-Rich 337
Cohesive 337
Immutable 337
Combinable 339
Self-Validating 341
Testable 344
Common Modeling Patterns 345
Static Factory Methods 345
Micro Types (Also Known as Tiny Types) 347
Collection Aversion 349
Persistence 351
NoSQL 352
SQL 353
Flat Denormalization 353
Normalizing into Separate Tables 357
The Salient Points 359

CHAPTER 16: ENTITIES 361

Understanding Entities 362
Domain Concepts with Identity and Continuity 362
Context-Dependent 363
Implementing Entities 363
Assigning Identifiers 363
Natural Keys 363
Arbitrarily Generated IDs 364
Datastore-Generated IDs 368
Pushing Behavior into Value Objects and Domain Services 369
Validating and Enforcing Invariants 371
Focusing on Behavior, Not Data 374
Avoiding the "Model the Real-World" Fallacy 377
Designing for Distribution 378
Common Entity Modeling Principles and Patterns 380

Implementing Validation and Invariants with Specifications 380
Avoid the State Pattern; Use Explicit Modeling 382
Avoiding Getters and Setters with the Memento Pattern 385
Favor Hidden-Side-Effect-Free Functions 386
The Salient Points 388

CHAPTER 17: DOMAIN SERVICES 389

Understanding Domain Services 390
When to Use a Domain Service 390
Encapsulating Business Policies and Processes 390
Representing Contracts 394
Anatomy of a Domain Service 395
Avoiding Anemic Domain Models 395
Contrasting with Application Services 396
Utilizing Domain Services 397
In the Service Layer 397
In the Domain 398
Manually Wiring Up 399
Using Dependency Injection 400
Using a Service Locator 400
Applying Double Dispatch 401
Decoupling with Domain Events 402
Should Entities Even Know About Domain Services? 403
The Salient Points 403

CHAPTER 18: DOMAIN EVENTS 405

Essence of the Domain Events Pattern 406
Important Domain Occurrences That Have Already Happened 406
Reacting to Events 407
Optional Asynchrony 407
Internal vs External Events 408
Event Handling Actions 409
Invoke Domain Logic 409
Invoke Application Logic 410
Domain Events' Implementation Patterns 410
Use the .Net Framework's Events Model 410
Use an In-Memory Bus 412
Udi Dahan's Static DomainEvents Class 415
Handling Threading Issues 417

Avoid a Static Class by Using Method Injection 418
Return Domain Events 419
Use an IoC Container as an Event Dispatcher 421
Testing Domain Events 422
Unit Testing 422
Application Service Layer Testing 424
The Salient Points 425

CHAPTER 19: AGGREGATES 427

Managing Complex Object Graphs 428
Favoring a Single Traversal Direction 428
Qualifying Associations 430
Preferring IDs Over Object References 431
Aggregates 434
Design Around Domain Invariants 435
Higher Level of Domain Abstraction 435
Consistency Boundaries 435
Transactional Consistency Internally 436
Eventual Consistency Externally 439
Special Cases 440
Favor Smaller Aggregates 441
Large Aggregates Can Degrade Performance 441
Large Aggregates Are More Susceptible to Concurrency Conflicts 442
Large Aggregates May Not Scale Well 442
Defining Aggregate Boundaries 442
eBidder: The Online Auction Case Study 443
Aligning with Invariants 444
Aligning with Transactions and Consistency 446
Ignoring User Interface Influences 448
Avoiding Dumb Collections and Containers 448
Don't Focus on HAS-A Relationships 449
Refactoring to Aggregates 449
Satisfying Business Use Cases—Not Real Life 449
Implementing Aggregates 450
Selecting an Aggregate Root 450
Exposing Behavioral Interfaces 452
Protecting Internal State 453
Allowing Only Roots to Have Global Identity 454
Referencing Other Aggregates 454

Nothing Outside An Aggregate's Boundary May Hold a Reference to Anything Inside 455
The Aggregate Root Can Hand Out Transient References to the Internal Domain Objects 456
Objects within the Aggregate Can Hold References to Other Aggregate Roots 456
Implementing Persistence 458
Access to Domain Objects for Reading Can Be at the Database Level 460
A Delete Operation Must Remove Everything within the Aggregate Boundary at Once 461
Avoiding Lazy Loading 461
Implementing Transactional Consistency 462
Implementing Eventual Consistency 463
Rules That Span Multiple Aggregates 463
Asynchronous Eventual Consistency 464
Implementing Concurrency 465
The Salient Points 468

CHAPTER 20: FACTORIES 469

The Role of a Factory 469
Separating Use from Construction 470
Encapsulating Internals 470
Hiding Decisions on Creation Type 472
Factory Methods on Aggregates 474
Factories for Reconstitution 475
Use Factories Pragmatically 477
The Salient Points 477

CHAPTER 21: REPOSITORIES 479

Repositories 479
A Misunderstood Pattern 481
Is the Repository an Antipattern? 481
The Difference between a Domain Model and a Persistence Model 482
The Generic Repository 483
Aggregate Persistence Strategies 486
Using a Persistence Framework That Can Map the Domain Model to the Data Model without Compromise 486
Using a Persistence Framework That Cannot Map the Domain Model Directly without Compromise 487

Public Getters and Setters 487
Using the Memento Pattern 488
Event Streams 491
Be Pragmatic 491
A Repository Is an Explicit Contract 492
Transaction Management and Units of Work 493
To Save or Not To Save 497
Persistence Frameworks That Track Domain Object Changes 497
Having to Explicitly Save Changes to Aggregates 498
The Repository as an Anticorruption Layer 499
Other Responsibilities of a Repository 500
Entity ID Generation 500
Collection Summaries 502
Concurrency 503
Audit Trails 506
Repository Antipatterns 506
Antipatterns: Don't Support Ad Hoc Queries 506
Antipatterns: Lazy Loading Is Design Smell 507
Antipatterns: Don't Use Repositories for Reporting Needs 507
Repository Implementations 508
Persistence Framework Can Map Domain Model to Data Model without Compromise 509
NHibernate Example 509
RavenDB Example 543
Persistence Framework Cannot Map Domain Model Directly without Compromise 557
Entity Framework Example 558
Micro ORM Example 577
The Salient Points 593

CHAPTER 22: EVENT SOURCING 595

The Limitations of Storing State as a Snapshot 596
Gaining Competitive Advantage by Storing State as a Stream of Events 597
Temporal Queries 597
Projections 599
Snapshots 599
Event-Sourced Aggregates 600
Structuring 600
Adding Event-Sourcing Capabilities 601

Exposing Expressive Domain-Focused APIs 602
Adding Snapshot Support 604
Persisting and Rehydrating 605
Creating an Event-Sourcing Repository 605
Adding Snapshot Persistence and Reloading 607
Handling Concurrency 609
Testing 610
Building an Event Store 611
Designing a Storage Format 612
Creating Event Streams 614
Appending to Event Streams 614
Querying Event Streams 615
Adding Snapshot Support 616
Managing Concurrency 618
A SQL Server-Based Event Store 621
Choosing a Schema 621
Creating a Stream 622
Saving Events 623
Loading Events from a Stream 624
Snapshots 625
Is Building Your Own Event Store a Good Idea? 627
Using the Purpose-Built Event Store 627
Installing Greg Young's Event Store 628
Using the C# Client Library 627
Running Temporal Queries 632
Querying a Single Stream 632
Querying Multiple Streams 634
Creating Projections 635
CQRS with Event Sourcing 637
Using Projections to Create View Caches 638
CQRS and Event Sourcing Synergy 638
Event Streams as Queues 639
No Two-Phase Commits 639
Recapping the Benefits of Event Sourcing 639
Competitive Business Advantage 639
Expressive Behavior-Focused Aggregates 639
Simplified Persistence 640
Superior Debugging 640
Weighing the Costs of Event Sourcing 640
Versioning 640
New Concepts to Learn and Skills to Hone 640

New Technologies to Learn and Master 641
Greater Data Storage Requirements 641
Additional Learning Resources 641
The Salient Points 641

PART IV: DESIGN PATTERNS FOR EFFECTIVE APPLICATIONS

CHAPTER 23: ARCHITECTING APPLICATION USER INTERFACES 645

Design Considerations 646
Owned UIs versus Composed UIs 646
Autonomous 646
Authoritative 647
Some Help Deciding 648
HTML APIs versus Data APIs 649
Client versus Server-Side Aggregation/Coordination 649
Example 1: An HTML API-Based, Server-Side UI for Nondistributed Bounded Contexts 651
Example 2: A Data API-Based, Client-Side UI for Distributed Bounded Contexts 658
The Salient Points 667

CHAPTER 24: CQRS: AN ARCHITECTURE OF A BOUNDED CONTEXT 669

The Challenges of Maintaining a Single Model for Two Contexts 670
A Better Architecture for Complex Bounded Contexts 670
The Command Side: Business Tasks 672
Explicitly Modeling Intent 672
A Model Free from Presentational Distractions 674
Handling a Business Request 675
The Query Side: Domain Reporting 676
Reports Mapped Directly to the Data Model 676
Materialized Views Built from Domain Events 678
The Misconceptions of CQRS 679
CQRS Is Hard 679
CQRS Is Eventually Consistent 679
Your Models Need to Be Event Sourced 680
Commands Should Be Asynchronous 680
CQRS Only Works with Messaging Systems 680
You Need to Use Domain Events with CQRS 680

Patterns to Enable Your Application to Scale 680
Scaling the Read Side: An Eventually Consistent Read Model 681
The Impact to the User Experience 682
Use the Read Model to Consolidate Many Bounded Contexts 682
Using a Reporting Database or a Caching Layer 682
Scaling the Write Side: Using Asynchronous Commands 683
Command Validation 683
Impact to the User Experience 684
Scaling It All 684
The Salient Points 685

CHAPTER 25: COMMANDS: APPLICATION SERVICE PATTERNS FOR PROCESSING BUSINESS USE CASES 687

Differentiating Application Logic and Domain Logic 689
Application Logic 689
Infrastructural Concerns 690
Coordinating Full Business Use Cases 698
Application Services and Framework Integration 698
Domain Logic from an Application Service's Perspective 700
Application Service Patterns 700
Command Processor 701
Publish/Subscribe 704
Request/Reply Pattern 706
async/await 708
Testing Application Services 709
Use Domain Terminology 709
Test as Much Functionality as Possible 710
The Salient Points 712

CHAPTER 26: QUERIES: DOMAIN REPORTING 713

Domain Reporting within a Bounded Context 714
Deriving Reports from Domain Objects 714
Using Simple Mappings 714
Using the Mediator Pattern 718
Going Directly to the Datastore 720
Querying a Datastore 721
Reading Denormalized View Caches 724
Building Projections from Event Streams 726
Setting Up ES for Projections 727

Creating Reporting Projections 728
Counting the Number of Events in a Stream 729
Creating As Many Streams As Required 729
Building a Report from Streams and Projections 730
Domain Reporting Across Bounded Contexts 733
Composed UI 733
Separate Reporting Context 734
The Salient Points 736

INDEX *737*

INTRODUCTION

WRITING SOFTWARE IS EASY— at least if it's greenfield software. When it comes to modifying code written by other developers or code you wrote six months ago, it can be a bit of a bore at best and a nightmare at worst. The software works, but you aren't sure exactly how. It contains all the right frameworks and patterns, and has been created using an agile approach, but introducing new features into the codebase is harder than it should be. Even business experts aren't helpful because the code bears no resemblance to the language they use. Working on such systems becomes a chore, leaving developers frustrated and devoid of any coding pleasure.

Domain-Driven Design (DDD) is a process that aligns your code with the reality of your problem domain. As your product evolves, adding new features becomes as easy as it was in the good old days of greenfield development. Although DDD understands the need for software patterns, principles, methodologies, and frameworks, it values developers and domain experts working together to understand domain concepts, policies, and logic equally. With a greater knowledge of the problem domain and a synergy with the business, developers are more likely to build software that is more readable and easier to adapt for future enhancement.

Following the DDD philosophy will give developers the knowledge and skills they need to tackle large or complex business systems effectively. Future enhancement requests won't be met with an air of dread, and developers will no longer have stigma attached to the legacy application. In fact, the term *legacy* will be recategorized in a developer's mind as meaning this: a system that continues to give value for the business.

OVERVIEW OF THE BOOK AND TECHNOLOGY

This book provides a thorough understanding of how you can apply the patterns and practices of DDD on your own projects, but before delving into the details, it's good to take a bird's-eye view of the philosophy so you can get a sense of what DDD is really all about.

The Problem Space

Before you can develop a solution, you must understand the problem. DDD emphasizes the need to focus on the business problem domain: its terminology, the core reasons behind why the software is being developed, and what success means to the business. The need for the development team to value domain knowledge just as much as technical expertise is vital to gain a deeper insight into the problem domain and to decompose large domains into smaller subdomains.

Figure I-1 shows a high-level overview of the problem space of DDD that will be introduced in the first part of this book.

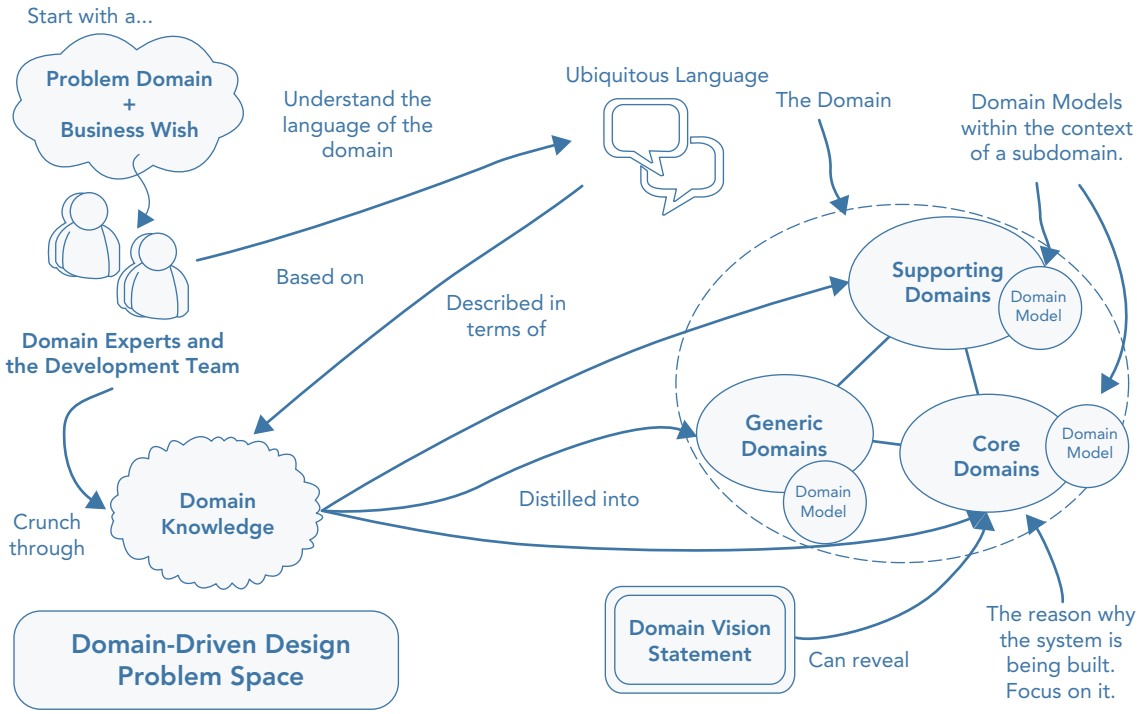


FIGURE I-1: A blueprint of the problem space of DDD.

The Solution Space

When you have a sound understanding of the problem domain, strategic patterns of DDD can help you implement a technical solution in synergy with the problem space. Patterns enable core parts of your system that are crucial to the success of the product to be protected from the generic areas. Isolating integral components allows them to be modified without having a rippling effect throughout the system.

Core parts of your product that are sufficiently complex or will frequently change should be based on a model. The tactical patterns of DDD along with Model-Driven Design will help you create a useful model of your domain in code. A model is the home to all of the domain logic that enables your application to fulfill business use cases. A model is kept separate from technical complexities to enable business rules and policies to evolve. A model that is in synergy with the problem domain will enable your software to be adaptable and understood by other developers and business experts.

Figure I-2 shows a high-level overview of the solution space of DDD that is introduced in the first part of this book.

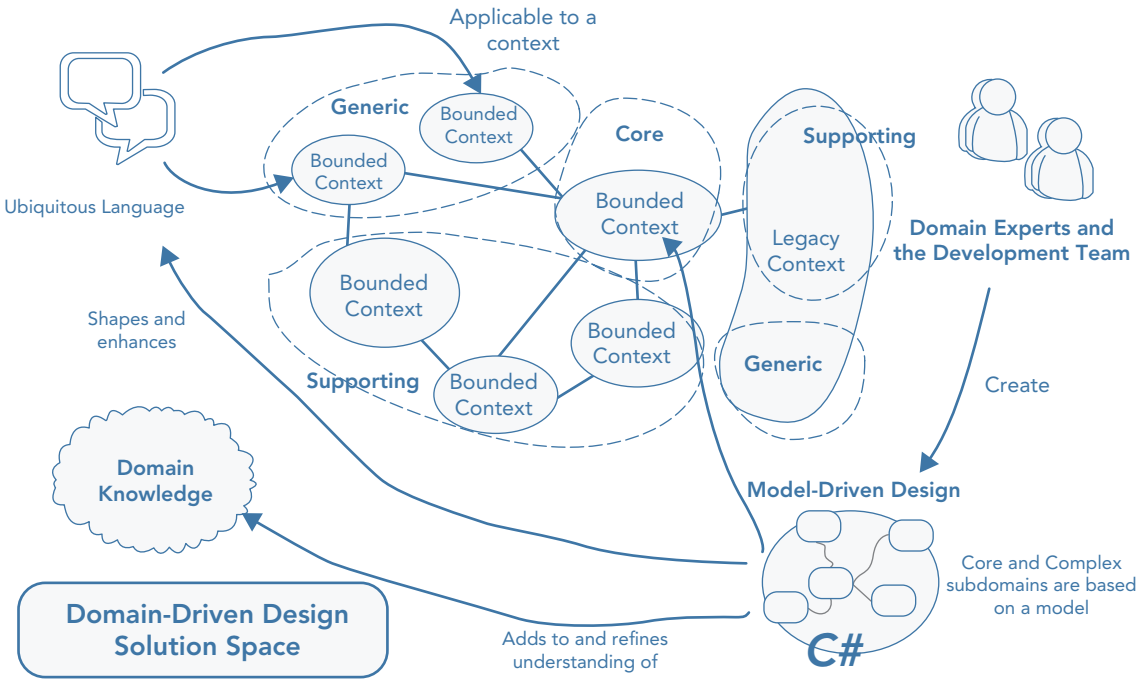


FIGURE I-2: A blueprint of the solution space of Domain-Driven Design.

HOW THIS BOOK IS ORGANIZED

This book is divided into four parts. Part I focuses on the philosophy, principles, and practices of DDD. Part II details the strategic patterns of integrating bounded contexts. Part III covers tactical patterns for creating effective domain models. Part IV delves into design patterns you can apply to utilize the domain model and build effective applications.

Part I: The Principles and Practices of Domain-Driven Design

Part I introduces you to the principles and practices of DDD.

Chapter 1: What Is Domain-Driven Design?

DDD is a philosophy to help with the challenges of building software for complex domains. This chapter introduces the philosophy and explains why language, collaboration, and context are the most important facets of DDD and why it is much more than a collection of coding patterns.

Chapter 2: Distilling the Problem Domain

Making sense of a complex problem domain is essential to creating maintainable software. Knowledge crunching with domain experts is key to unlocking that knowledge. Chapter 2 details techniques to enable development teams to collaborate, experiment, and learn with domain experts to create an effective domain model.

Chapter 3: Focusing on the Core Domain

Chapter 3 explains how to distill large problem domains and identify the most important part of a problem: the core domain. It then explains why you should focus time and energy in the core domain and isolate it from the less important supporting and generic domains.

Chapter 4: Model-Driven Design

Business colleagues understand an analysis model based on the problem area you are working within. Development teams have their own code version of this model. In order for business and technical teams to collaborate a single model is needed. A ubiquitous language and a shared understanding of the problem space is what binds the analysis model to the code model. The idea of a shared language is core to DDD and underpins the philosophy. A language describing the terms and concepts of the domain, which is created by both the development team and the business experts, is vital to aid communication on complex systems.

Chapter 5: Domain Model Implementation Patterns

Chapter 5 expands on the role of the domain model within your application and the responsibilities it takes on. The chapter also presents the various patterns that can be used to implement a domain model and what situations they are most appropriate for.

Chapter 6: Maintaining the Integrity of Domain Models with Bounded Contexts

In large solutions more than a single model may exist. It is important to protect the integrity of each model to remove the chance of ambiguity in the language and concepts being reused inappropriately by different teams. The strategic pattern known as bounded context is designed to isolate and protect a model in a context while ensuring it can collaborate with other models.

Chapter 7: Context Mapping

Using a context map to understand the relationships between different models in an application and how they integrate is vital for strategic design. It is not only the technical integrations that context maps cover but also the political relationships between teams. Context maps provide a view of the landscape that can help teams understand their model in the context of the entire landscape.

Chapter 8: Application Architecture

An application needs to be able to utilize the domain model to satisfy business use cases. Chapter 8 introduces architectural patterns to structure your applications to retain the integrity of your domain model.

Chapter 9: Common Problems for Teams Starting Out with Domain-Driven Design

Chapter 9 describes the common issues teams face when applying DDD and why it's important to know when not to use it. The chapter also focuses on why applying DDD to simple problems can lead to overdesigned systems and needless complexity.

Chapter 10: Applying the Principles, Practices, and Patterns of DDD

Chapter 10 covers techniques to sell DDD and to start applying the principles and practices to your projects. It explains how exploration and experimentation are more useful to build great software than trying to create the perfect domain model.

Part II: Strategic Patterns: Communicating between Bounded Contexts

Part II shows you how to integrate bounded contexts, and offers details on the options open for architecting bounded contexts. Code examples are presented that detail how to integrate with legacy applications. Also included are techniques for communicating across bounded contexts.

Chapter 11: Introduction to Bounded Context Integration

Modern software applications are distributed systems that have scalability and reliability requirements. This chapter blends distributed systems theory with DDD so that you can have the best of both worlds.

Chapter 12: Integrating via Messaging

A sample application is built showing how to apply distributed systems principles synergistically with DDD using a message bus for asynchronous messaging.

Chapter 13: Integrating via HTTP with RPC and REST

Another sample application is built showing an alternative approach to building asynchronous distributed systems. This approach uses standard protocols like Hypertext Transport Protocol (HTTP), REST, and Atom instead of a message bus.

Part III: Tactical Patterns: Creating Effective Domain Models

Part III covers the design patterns you can use to build a domain model in code, along with patterns to persist your model and patterns to manage the lifecycles of the domain objects that form your model.

Chapter 14: Introducing the Domain Modeling Building Blocks

This chapter is an introduction to all the tactical patterns at your disposal that allow you to build an effective domain model. The chapter highlights some best practice guidelines that produce more manageable and expressive models in code.

Chapter 15: Value Objects

This is an introduction to the DDD modeling construct that represents identityless domain concepts like money.

Chapter 16: Entities

Entities are domain concepts that have an identity, such as customers, transactions, and hotels. This chapter covers a variety of examples and complementary implementation patterns.

Chapter 17: Domain Services

Some domain concepts are stateless operations that do not belong to a value object or an entity. They are known as domain services.

Chapter 18: Domain Events

In many domains, focusing on events reveals greater insight than focusing on just entities. This chapter introduces the domain event design pattern that allows you to express events more clearly in your domain model.

Chapter 19: Aggregates

Aggregates are clusters of domain objects that represent domain concepts. Aggregates are a consistency boundary defined around invariants. They are the most powerful of the tactical patterns.

Chapter 20: Factories

Factories are a lifecycle pattern that separate use from construction for complex domain objects.

Chapter 21: Repositories

Repositories mediate between the domain model and the underlying data model. They ensure that the domain model is kept separate from any infrastructure concerns.

Chapter 22: Event Sourcing

Like domain events in Chapter 18, event sourcing is a useful technique for emphasizing, in code, events that occur in the problem domain. Event sourcing goes beyond domain events by storing the state of the domain model as events. This chapter provides a number of examples, including ones that use a purpose-built event store.

Part IV: Design Patterns for Effective Applications

Part IV showcases the design patterns for architecting applications that utilize and protect the integrity of your domain model.

Chapter 23: Architecting Application User Interfaces

For systems composed of many bounded contexts, the user interface often requires the composition of data from a number of them, especially when your bounded contexts form a distributed system.

Chapter 24: CQRS: An Architecture of a Bounded Context

CQRS is a design pattern that creates two models where there once was one. Instead of a single model to handle the two different contexts of reads and writes, two explicit models are created to handle commands or serve queries for reports.

Chapter 25: Commands: Application Service Patterns for Processing Business Use Cases

Learn the difference between application and domain logic to keep your model focused and your system maintainable.

Chapter 26: Queries: Domain Reporting

Business people need information to make informed business and product-development decisions. A range of techniques for building reports that empower the business is demonstrated in this chapter.

WHO SHOULD READ THIS BOOK

This book introduces the main themes behind DDD—its practices, patterns, and principles along with personal experiences and interpretation of the philosophy. It is intended to be used as a learning aid for those interested in or starting out with the philosophy. It is not a replacement for *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans (Addison-Wesley Professional, 2003). Instead, it takes the concepts introduced by Evans and distills them into simple straightforward prose, with practical examples so that any developer can get up to speed with the philosophy before going on to study the subject in more depth.

This book is based on the author's personal experiences with the subject matter. You may not always agree with it if you are a seasoned DDD practitioner, but you should still get something out of it.

SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually, or to use the source code files that accompany the book. All the source code used in this book is available for download at `www.wrox.com`. Specifically for this book, the code download is on the Download Code tab at: `www.wrox.com/go/domaindrivendesign`. Although code examples are presented in C# .NET. The concepts and practices can be applied to any programming language.

You can also search for the book at `www.wrox.com` by ISBN (the ISBN for this book is 978-1-1187-1470-6) to find the code. And a complete list of code downloads for all current Wrox books is available at `www.wrox.com/dynamic/books/download.aspx`.

ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata, you may save another reader hours of frustration, and at the same time, you will be helping us provide even higher quality information.

To find the errata page for this book, go to `www.wrox.com/go/domaindrivendesign`.

And click the Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors.

If you don't spot "your" error on the Book Errata page, go to `www.wrox.com/contact/techsupport.shtml` and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

P2P.WROX.COM

For author and peer discussion, join the P2P forums at `http://p2p.wrox.com`. The forums are a web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At `http://p2p.wrox.com`, you will find a number of different forums that will help you, not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to `http://p2p.wrox.com` and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join, as well as any optional information you wish to provide, and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

> **NOTE** *You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.*

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works, as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

SUMMARY

The aim of this book is to present the philosophy of DDD in a down-to-earth and practical manner for experienced developers building applications for complex domains. A focus is placed on the principles and practices of decomposing a complex problem space as well as the implementation patterns and best practices for shaping a maintainable solution space. You will learn how to build effective domain models by using tactical patterns and how to retain their integrity by applying the strategic patterns of DDD.

By the end of this book, you will have a thorough understanding of DDD. You will be able to communicate its value and when to use it. You will understand that even though the tactical patterns of DDD are useful, it is the principles, practices, and strategic patterns that will help you architect applications for maintenance and scale. With the information gained within this book, you will be in a better place to manage the construction and maintenance of complex software for large and complex problem domains.

PART I

The Principles and Practices of Domain-Driven Design

- **CHAPTER 1:** What Is Domain-Driven Design?
- **CHAPTER 2:** Distilling the Problem Domain
- **CHAPTER 3:** Focusing on the Core Domain
- **CHAPTER 4:** Model-Driven Design
- **CHAPTER 5:** Domain Model Implementation Patterns
- **CHAPTER 6:** Maintaining the Integrity of Domain Models with Bounded Contexts
- **CHAPTER 7:** Context Mapping
- **CHAPTER 8:** Application Architecture
- **CHAPTER 9:** Common Problems for Teams Starting Out with Domain-Driven Design
- **CHAPTER 10:** Applying the Principles, Practices, and Patterns of DDD

1

What Is Domain-Driven Design?

WHAT'S IN THIS CHAPTER?

- An introduction to the philosophy of Domain-Driven Design
- The challenges of writing software for complex problem domains
- How Domain-Driven Design manages complexity
- How Domain-Driven Design applies to both the problem and solution space
- The strategic and tactical patterns of Domain-Driven Design
- The practices and principles of Domain-Driven Design
- The misconceptions of Domain-Driven Design

Domain-Driven Design (DDD) is a development philosophy defined by Eric Evans in his seminal work *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley Professional, 2003). DDD is an approach to software development that enables teams to effectively manage the construction and maintenance of software for complex problem domains.

This chapter will give you a high-level introduction to DDD's practices, patterns, and principles along with an explanation of how it will improve your approach to software development. You will learn the value of analyzing a problem space and where to focus your efforts. You will understand why collaboration, communication, and context are so important for the design of maintainable software.

At the end of this chapter you will have a solid understanding of DDD that will provide context to the detail of the various patterns, practices, and principles that are contained throughout this book. However, before we delve into how DDD handles complexity it's important to understand what problems can cause software to get into an unmanageable state.

THE CHALLENGES OF CREATING SOFTWARE FOR COMPLEX PROBLEM DOMAINS

To understand how DDD can help with the design of software for a nontrivial domain, you must first understand the difficulties of creating and maintaining software. By far, the most popular software architectural design pattern for business applications is the Big Ball of Mud (BBoM) pattern. The definition of BBoM, as defined by Brian Foote and Joseph Yoder in the paper "Big Ball of Mud," is "… a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle."

Foote and Yoder use the term BBoM to describe an application that appears to have no distinguishable architecture (think big bowl of spaghetti versus dish of layered lasagna). The issue with allowing software to dissolve into a BBoM becomes apparent when routine changes in workflow and small feature enhancements become a challenge to implement due to the difficulties in reading and understanding the existing codebase. In his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley Professional, 2003), Eric Evans describes such systems as containing "code that does something useful, but without explaining how." One of the main reasons software becomes complex and difficult to manage is due to the mixing of domain complexities with technical complexities, as illustrated in Figure 1-1.

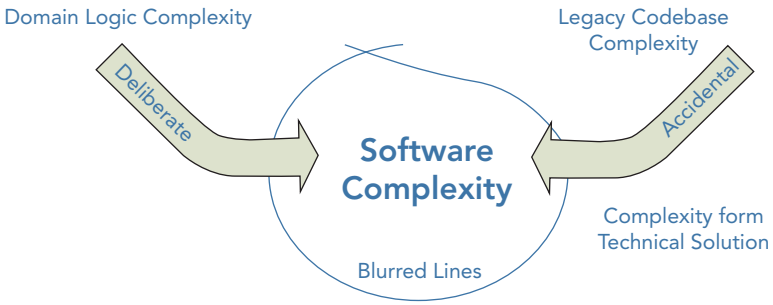


FIGURE 1-1: Complexity in software.

Code Created without a Common Language

A lack of focus on a shared language and knowledge of the problem domain results in a codebase that works but does not reveal the intent of the business. This makes codebases difficult to read and maintain because translations between the analysis model and the code model can be costly and error prone.

Code without a binding to an analysis model that the business understands will degrade over time and is therefore more likely to result in an architecture that resembles the BBoM pattern. Due to the cost of translation teams that do not utilize the rich vocabulary of the problem domain in code will decrease their chances of discovering new domain concepts when collaborating with business experts.

WHAT IS AN ANALYSIS MODEL?

An analysis model is used to describe the logical design and structure of a software application. It can be represented as sketches or by using modeling languages such as UML. It is the representation of software that non-technical people can conceptualize in order to understand how software is constructed.

A Lack of Organization

As highlighted in Figure 1-2, the initial incarnation of a system that resembles BBoM is fast to produce and often a well-rounded success, but because there is little focus based on the design of an application around a model of the problem domain, subsequent enhancements are troublesome. The codebase lacks the required synergy with the business behavior to make change manageable. Complexities of the problem domain are often mixed with the accidental complexities of the technical solution.

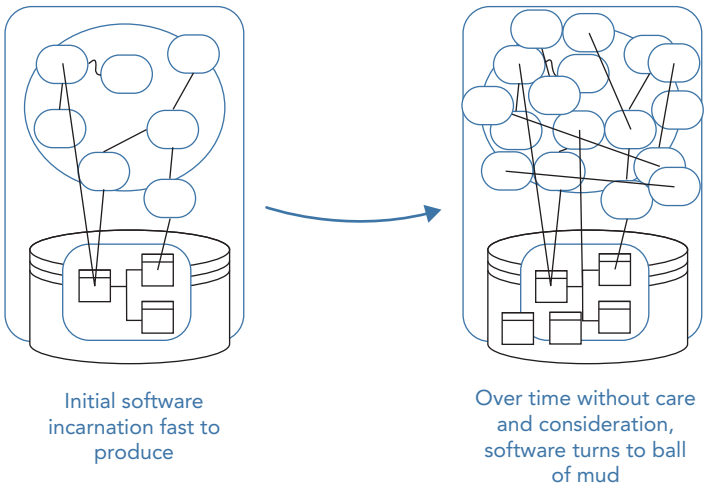


FIGURE 1-2: Code rot.

The Ball of Mud Pattern Stifles Development

Continuing to persist with an architectural spaghetti-like pattern can lead to a sluggish pace of feature enhancement. When newer versions of the product are released, they can be buggy due to the unintelligible mess of the codebase that developers have to deal with. Over time, the development team increasingly complains about the difficulty of working in such a mess. Even if resources are added to the project, velocity cannot be increased to a level that satisfies the business.

In the end, exasperated by the situation, the request for the dreaded application rewrite is granted. Without due care and consideration, however, even the greenfield project can fall foul of the same issues that created the original BBoM. This entire experience can be frustrating for the business that saw a great return on investment (ROI) in terms of features and speed of delivery at the beginning but over time, even with additional investment in resources, did not see the sustained evolution of the product to meet their needs. Ultimately the BBoM is bad news for you as a developer because it's a messy bug-prone code base that you hate dealing with. And it's bad news for the business because it reduces their capability to rapidly deliver business value

A Lack of Focus on the Problem Domain

Software projects fail when you don't understand the business domain you are working within well enough. Typing is not the bottleneck for delivering a product; coding is the easy part of development. Outside of non-functional requirements creating and keeping a useful software model of the domain that can fulfill business-use cases is the difficult part. However, the more you invest in understanding your business domain the better equipped you will be when you are trying to model it in software to solve its inherent business problems.

WHAT IS A PROBLEM DOMAIN?

A problem domain refers to the subject area for which you are building software. DDD stresses the need to focus on the domain above anything else when working on creating software for large-scale and complex business systems. Experts in the problem domain work with the development team to focus on the areas of the domain that are useful to be able to produce valuable software. For example, when writing software for the health industry to record patient treatment, it is not important to learn to become a doctor. What is important to understand is the terminology of the health industry, how different departments view patients and care, what information doctors gather, and what they do with it.

HOW THE PATTERNS OF DOMAIN-DRIVEN DESIGN MANAGE COMPLEXITY

DDD deals with both the challenge of understanding a problem domain and creating a maintainable solution that is useful to solve problems within it. It achieves this by utilizing a number of strategic and tactical patterns.

The Strategic Patterns of DDD

The strategic patterns of DDD distil the problem domain and shape the architecture of an application.

Distilling the Problem Domain to Reveal What Is Important

Not all of a large software product needs be perfectly designed—in fact trying to do so would be a waste of effort. Development teams and domain experts use analysis patterns and knowledge crunching to distill large problem domains into more manageable subdomains. This distillation reveals the core sub domain—the reason the software is being written. The core domain is the driving force behind the product under development; it is the fundamental reason it is being built. DDD emphasizes the need to focus effort and talent on the core subdomain(s) as this is the area that holds the most value and is key to the success of the application.

This clarity on where to focus effort can also empower teams to look for open source off-the-shelf solutions for some of the less important parts of a system, which means that they have more time to focus on what is important and ensure that the core domain does not become a BBoM.

Discovering the core domain helps teams understand why they're producing the software and what it means for the software to be successful to the business. It is the appreciation for the business intent that will enable the development team to identify and invest its time in the most important parts of the system. As the business evolves, so in turn must the software; it needs to be adaptable. Investment in code quality for the key areas of an application will help it change with the business. If key areas of the software are not in synergy with the business domain then, over time, it is likely that the design will rot and turn into a big ball of mud, resulting in hard-to-maintain software.

Creating a Model to Solve Domain Problems

In the solution space a software model is built for each subdomain to handle domain problems and to align the software with the business contours. This model is not a model of real life but more an abstraction built to satisfy the requirements of business use cases while still retaining the rules and logic of the business domain. The development team should focus as much energy and effort on the model and domain logic as it does on the pure technical aspects of the application. To avoid accidental technical complexity the model is kept isolated from infrastructure code.

All models are not created equal; the most appropriate design patterns are used based on the complexity needs of each subdomain rather than applying a blanket design to the whole system. Models for subdomains that are not core to the success of the product or that are not as complex need not be based on rich object-oriented designs, and can instead utilize more procedural or data-driven architectures.

Using a Shared Language to Enable Modeling Collaboration

Models are built through the collaboration of domain experts and the development team. Communication is achieved using an ever-evolving shared language known as the ubiquitous language (UL) to efficiently and effectively connect a software model to a conceptual analysis model. The software model is bound to the analysis model by using the same terms of the UL for its structure and class design. Insights, concepts, and terms that are discovered at a coding

level are replicated in the UL and therefore the analytical model. Likewise when the business reveals hidden concepts at the analysis model level this insight is fed back into the code model; this is the key that enables the domain experts and development teams to evolve the model in collaboration.

Isolate Models from Ambiguity and Corruption

Models sit within a bounded context, which defines the applicability of the model and ensures that its integrity is retained. Larger models can be split into smaller models and defined within separate bounded contexts where ambiguity in terminology exists or where multiple teams are a working in order to further reduce complexity.

Bounded contexts are used to form a protective boundary around models that helps to prevent software from evolving into a BBoM. This is achieved by allowing the different models of the overall solution to evolve within well-defined business contexts without having a negative, rippling impact on other parts of the system. Models are isolated from infrastructure code to avoid the accidental complexity of merging technical and business concepts. Bounded contexts also prevent the integrity of models being corrupt by isolating them from third-party code.

Compare the diagram in Figure 1-3 to Figure 1-2. The diagram shows how the strategic patterns of DDD have been applied to the software to manage the large problem domain and protect discrete models within it.

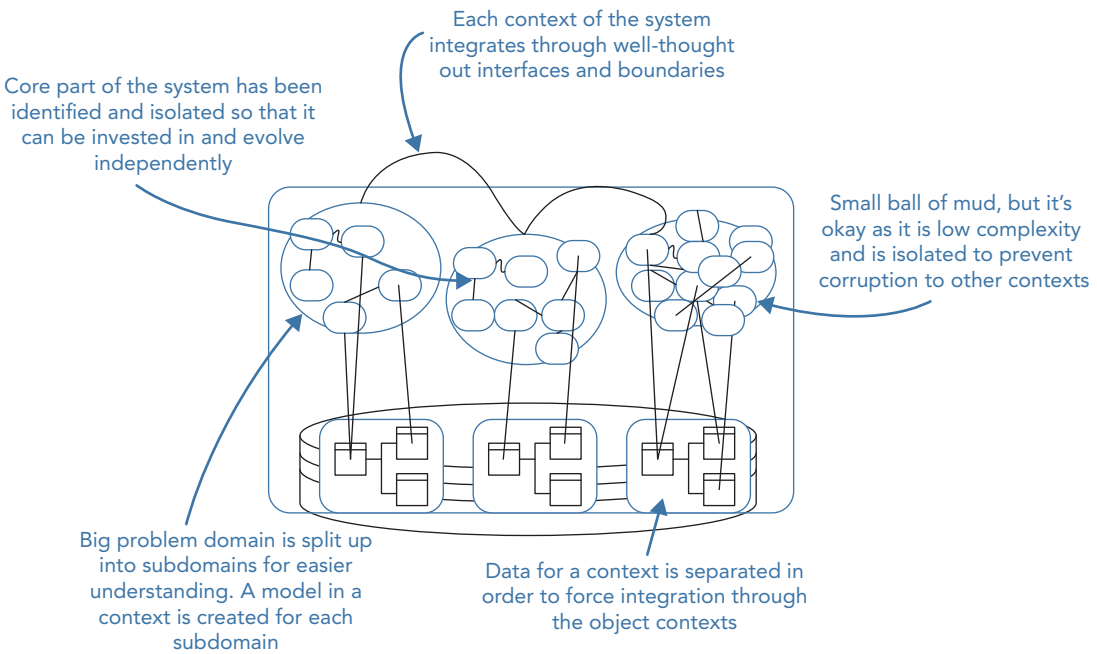


FIGURE 1-3: Applying the strategic patterns of Domain-Driven Design.

THE BIG BALL OF MUD IS NOT ALWAYS AN ANTIPATTERN

Not all parts of a large application will be designed perfectly—nor do they need to be. Although it's not advisable to build an entire enterprise software stack following the BBoM pattern, you can still utilize the pattern. Areas of low complexity or that are unlikely to be invested in can be built without the need for perfect code quality; working software is good enough. Sometimes feedback and first-to-market are core to the success of a product; in this instance, it can make business sense to get working software up as soon as possible, whatever the architecture. Code quality can always be improved after the business deems the product to be a success and worthy of prolonged investment. The key to reaping the benefits of the BBoM is to define a context around the bounded contexts that use the BBoM to avoid them corrupting the core subcomain.

Understanding the Relationships between Contexts

DDD understands the need to ensure that teams and the business are clear on how separate models and contexts work together in order to solve domain problems that span across subdomains. Context maps help you to understand the bigger picture; they enable teams to understand what models exist, what they are responsible for, and where their applicability boundaries are. These maps reveal how different models interact and what data they exchange to fulfill business processes. The relationships between the connections and more importantly the grey area of process that sits between them is often not captured or well understood by the business.

The Tactical Patterns of DDD

The tactical patterns of DDD, also known as model building blocks, are a collection of patterns that help to create effective models for complex bounded contexts. Many of the coding patterns presented within the collection of tactical patterns have been widely adopted before Evans's text and catalogued by the likes of Martin Fowler in *Patterns of Enterprise Application Architecture* and Erich Gamma, et al. in *Design Patterns: Elements of Reusable Object-Oriented Software*. These patterns are not applicable to all models, and each must be taken on its own merit with the correct architectural style applied.

The Problem Space and the Solution Space

All of the patterns detailed in this section help to manage the complexity of a problem—aka the problem space or they manage complexity in the solution—aka the solution space. The problem space, as shown in Figure 1-4, distils the problem domain into more manageable subdomains. DDD's impact in the problem space is to reveal what is important and where to focus effort. In the next chapter we will look in more detail on the patterns that can help reduce complexity in the problem space.

The solution side of DDD, shown in Figure 1-5, covers patterns that can shape the architecture of your applications and make it easier to manage.

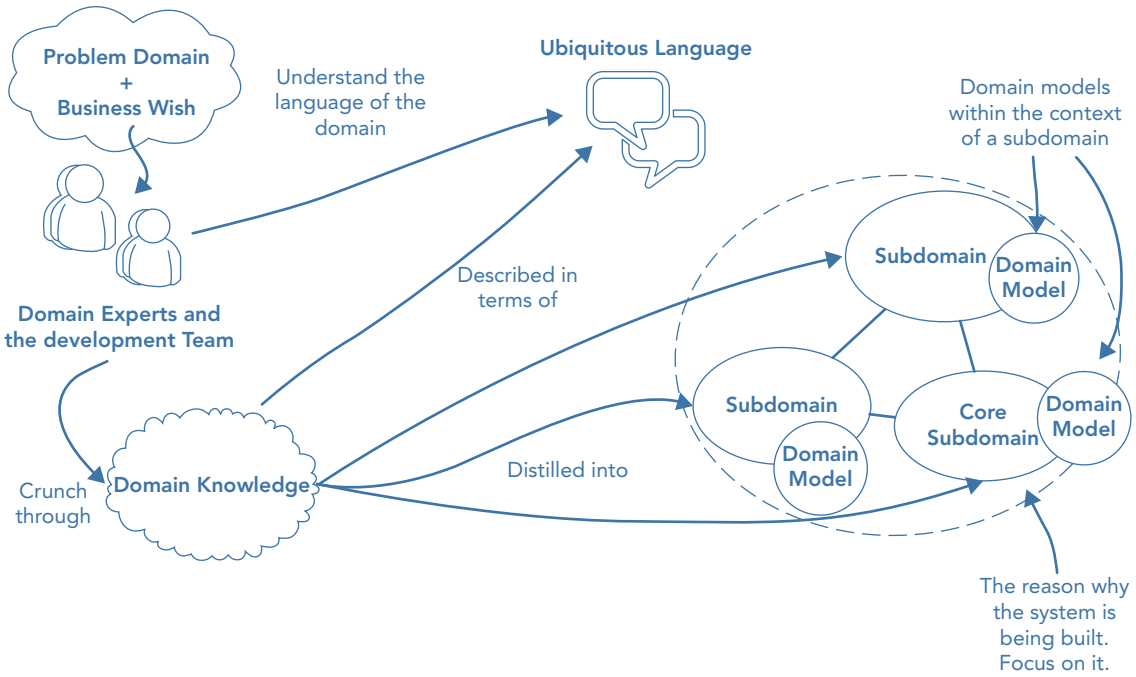


FIGURE 1-4: DDD patterns that are applicable to the problem space.

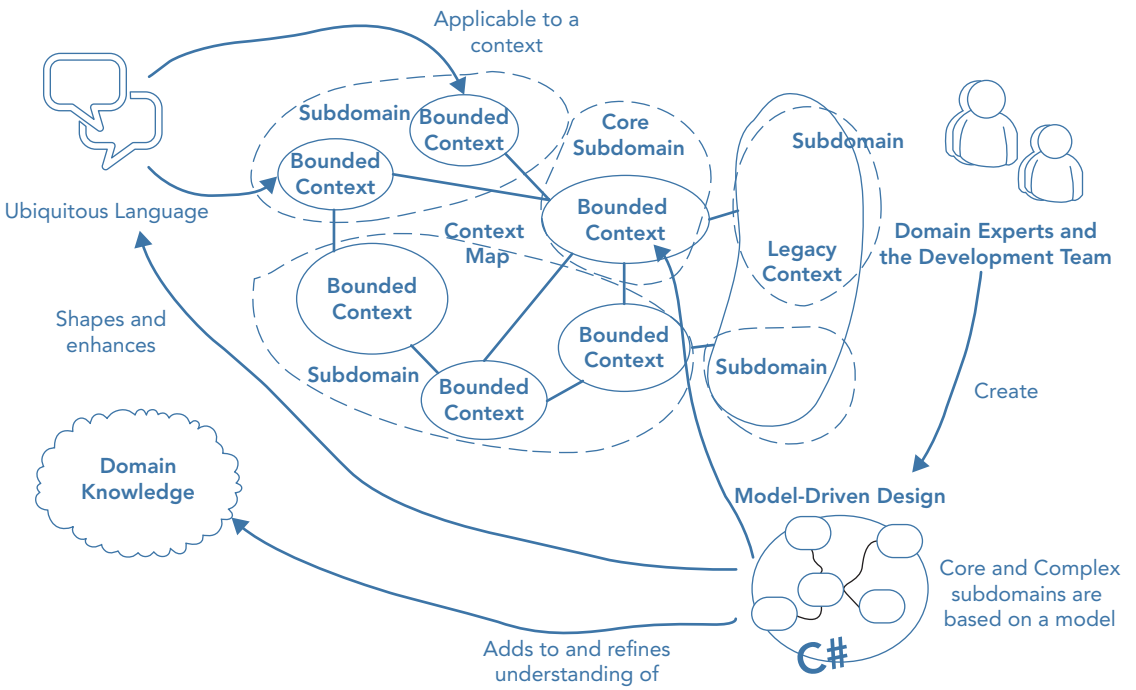


FIGURE 1-5: DDD patterns that are applicable to the solution space.

THE PRACTICES AND PRINCIPLES OF DOMAIN-DRIVEN DESIGN

Whilst there are many patterns of DDD, there are a number of practices and guiding principles that are key to success with its philosophy. These key principles, which form the essence of DDD, are often missed as too much focus is placed upon the tactical design patterns that are used to create software models.

Focusing on the Core Domain

DDD stresses the need to focus the most effort on the core subdomain. The core subdomain is the area of your product that will be the difference between it being a success and it being a failure. It's the product's unique selling point, the reason it is being built rather than bought. The core domain is the area of the product that will give you a competitive advantage and generate real value for your business. It is vital that all of the team understand what the core domain is.

Learning through Collaboration

DDD stresses the importance of collaboration between the development teams and business experts to produce useful models to solve problems. Without this collaboration and commitment from the business experts, much of the knowledge sharing will not be able to take place, and development teams will not gain deeper insights into the problem domain. It is also true that, through collaboration and knowledge crunching, the business has the opportunity to learn much more about its domain.

Creating Models through Exploration and Experimentation

DDD treats the analysis and code models as one. This means that the technical code model is bound to the analysis model through the shared UL. A breakthrough in the analysis model results in a change to the code model. A refactoring in the code model that reveals deeper insight is again reflected in the analysis model and mental models of the business. Breakthroughs only occur when teams are given time to explore a model and experiment with its design. Spending time prototyping and experimenting can go a long way in helping you shape a better design. It can also reveal what a poor design looks like. Eric Evans suggests that for every good design there must be at least three bad ones, this will prevent teams stopping at the first useful model.

Communication

The ability to effectively describe a model built to represent a problem domain is the foundation of DDD. This is why, without a doubt, the single most important facet of DDD is the creation of the UL. Without a shared language, collaboration between the business and development teams to solve problems would not be effective. Analysis and mental models produced in knowledge-crunching sessions between the teams need a shared language to bind them to a technical implementation. Without an effective way to communicate ideas and solutions within a problem domain, design breakthroughs cannot occur.

It is the collaboration and construction of a UL that makes DDD so powerful. It enables a greater understanding of the problem domain (for the business and the development team) and more

effective communication. These key values have a massive impact on projects because while technical frameworks and methodologies are important, DDD places as much, if not, greater importance on the analysis and understanding of the problem domain that ultimately makes software products successful.

Understanding the Applicability of a Model

Each model that is built is understood within the context of its subdomain and described using the UL. However, in many large models, there can be ambiguity within the UL, with different parts of an organization having different understandings of a common term or concept. DDD addresses this by ensuring that each model has its own UL that is valid only in a certain context. Each context defines a linguistic boundary; ensuring models are understood in a specific context to avoid ambiguity in language. Therefore a model with overlapping terms is divided into two models, each clearly defined within its own context. On the implementation side, strategic patterns can enforce these linguistic boundaries to enable models to evolve in isolation. These strategic patterns result in organized code that is able to support change and rewriting.

Constantly Evolving the Model

Any developer working on a complex system can write good code and maintain it for a short while. However, without synergy between the source code and the problem domain, continued development will likely end up in a codebase that is hard to modify, resulting in a BBoM. DDD helps with this issue by placing emphasis on the team to continually look at how useful the model is for the current problem. It challenges the team to evolve and simplify complex models of domains as and when it gains domain insights. DDD is still no silver bullet and requires dedication and constant knowledge crunching to produce software that is maintainable for years and not just months. New business cases may break a previously useful model, or may necessitate changes to make new or existing concepts more explicit.

POPULAR MISCONCEPTIONS OF DOMAIN-DRIVEN DESIGN

You can think of DDD as a development philosophy; it promotes a new domain-centric way of thinking. It is the learning process, not the end goal, which is the greatest strength of DDD. Any team can write a software product to meet the needs of a set of use cases, but teams that put time and effort into the problem domain they are working on can consistently evolve the product to meet new business use cases. DDD is not a strict methodology in itself but must be used with some form of iterative software project methodology to build and evolve a useful model.

Tactical Patterns Are Key to DDD

DDD is not a book on object-oriented design, nor is it a code-centric philosophy or a patterns language. However, if you search the web for articles on DDD, you would be mistaken for thinking that it is just a handful of implementation patterns as most articles and blogs on DDD focus on the modeling patterns. It is much easier for developers to see tactical patterns of DDD implemented in code rather than conversations between business users and teams on

a domain that they do not care about or do not understand. This is why DDD is sometimes mistakenly thought of as nothing more than a pattern language made up of entities, value objects, and repositories. You can, in fact, implement DDD without ever creating a rich domain model or using a repository. DDD is less about software design patterns and more about problem solving through collaboration.

Evans presents techniques to use software design patterns to enable models created by the development team and business experts to be implemented using the UL. However, without the practices of analysis, and collaboration, the coding implementation really means very little on its own. DDD is not code centric; its purpose is not to make elegant code. Software is merely an artifact of DDD.

DDD Is a Framework

DDD does not require a special framework or database. The model implemented in code follows a POCO (Plain Old C# Object) principle that ensures it is devoid of any infrastructural code so that nothing distracts from its domain-centric purpose. An object-oriented methodology is useful for constructing models, but it is by no means mandatory.

DDD is architecturally agnostic in that there is no single architectural style you must follow to implement it. A layered architectural style was presented in Evans's text, but this is not the only option. Architectural styles can vary because they should apply at the bounded context level and not the application level. A single product can include one bounded context that follows an event-centric architecture, another that utilizes a layered rich domain model, and a third that applies the active record pattern.

DDD Is a Silver Bullet

DDD can take a lot of effort, it requires an iterative development methodology, an engaged business, and smart developers. All software projects can benefit from the analysis practices of DDD such as distilling the problem domain as well as the strategic patterns such as isolating a code model that represents domain logic. However, not all require the tactical patterns of DDD to build a rich domain model. Trivial domains don't warrant the level of sophistication as they have little or no domain logic. For example, it would be a waste of time and costly to apply all of the patterns of DDD when creating a simple blogging application.

THE SALIENT POINTS

- Domain-Driven Design (DDD) is a development philosophy that is designed to manage the creation and maintenance of software written for complex problem domains.
- DDD is a collection of patterns, principles, and practices, which can be applied to software design to manage complexity.
- DDD has two types of patterns. Strategic patterns shape the solution, while tactical patterns are used to implement a rich domain model. Strategic patterns can be useful for any application but tactical patterns are only useful if your model is sufficiently rich in domain logic.

- Distillation of large problem domains into subdomains can reveal the core domain—the area of most value. Not all parts of a system will be well designed; teams must invest more time in the core subdomain(s) of a product.
- An abstract model is created for each subdomain to manage domain problems.
- A ubiquitous language is used to bind the analysis model to the code model in order for the development team and domain experts to collaborate on the design of a model. Learning and creating a language to communicate about the problem domain is the process of DDD. Code is the artifact.
- In order to retain the integrity of a model, it is defined within a bounded context. The model is isolated from infrastructure concerns of the application to separate technical complexities from business complexities.
- Where there is ambiguity in terminology for a model or multiple teams at work the model can be split and defined in smaller bounded contexts.
- DDD does not dictate any specific architectural style for development, it only ensures that the model is kept isolated from technical complexities so that it can focus on domain logic concerns.
- DDD values a focus on the core domain, collaboration, and exploration with domain experts, experimentation to produce a more useful model, and an understanding of the various contexts in play in a complex problem domain.
- DDD is not a patterns language, it is a collaboration philosophy focused on delivery, with communication playing a central role.
- DDD is a language- and domain-centric approach to software development.

2

Distilling the Problem Domain

WHAT'S IN THIS CHAPTER?

- The need for knowledge crunching
- How collaboration can foster a shared understanding and a shared language
- What a domain expert is and why the role is essential
- Effective methods for gaining domain knowledge

Making sense of a complex problem domain in order to create a simple and useful model requires in-depth knowledge and deep insight that can only be gained through collaboration with the people that understand the domain inside and out. Continuous experimentation and exploration in the design of a model is where the power of DDD is realized. Only through collaboration and a shared understanding of the problem domain can you effectively design a model to solve the challenges of the business that will be supple enough to adapt as new requirements surface.

This chapter introduces methods to facilitate the distilling of domain knowledge in order to better understand the problem domain, which will enable you to build an effective domain model. Methods to extract important information on the behaviors of an application along with techniques to discover deep insights within the problem domain are also presented.

KNOWLEDGE CRUNCHING AND COLLABORATION

Complex problem domains will contain a wealth of information, some of which will not be applicable to solving the problem at hand and will only act to distract from the real focus of your modelling efforts. Knowledge crunching is the art of distilling relevant information from the problem domain in order to build a useful model that can fulfill the needs of business use cases.

Knowledge crunching is key to bridging any knowledge gaps for the technical team when designing a solution for a problem domain based on a set of requirements. In order for a team to produce a useful model they need to have a deep insight of the problem domain to ensure important concepts are not overlooked or misunderstood. This can only be done through working in collaboration with the people that understand the domain the most; i.e., the business users, stakeholders, and subject matter experts. Without this there is a danger that a technical solution will be produced that is void of any real domain insight and something that cannot be understood by the business or by other developers during software maintenance or subsequent enhancements.

Knowledge gathering occurs on whiteboards, working through examples with business experts and generally brainstorming together. It is the quest to discover and agree on a shared understanding of the problem domain to produce a model that can fulfill business use cases. The process of knowledge crunching, as shown in Figure 2-1, starts with the behaviors of a system. The team go through the scenarios of the application with the business stakeholders and experts. This process is the catalyst to conversation, deep insight, and a shared understanding of the domain for all participants. It is therefore vital that stakeholders and subject matter experts are actively involved and engaged.

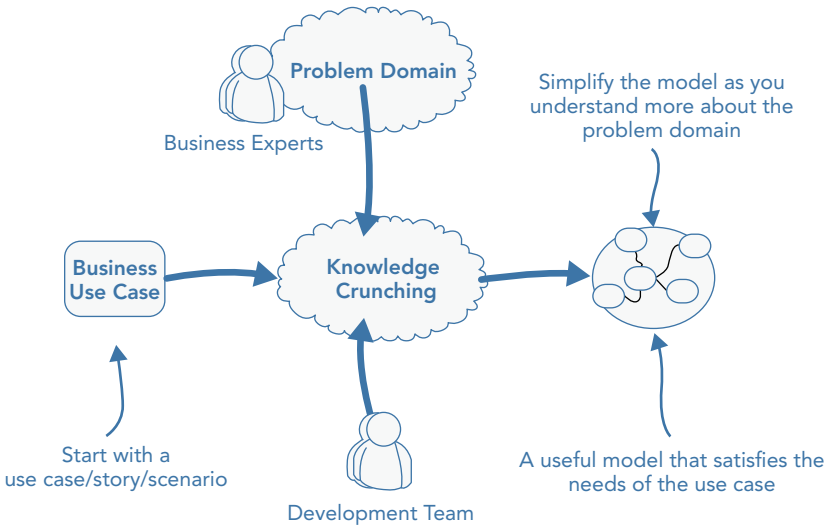


FIGURE 2-1: Knowledge crunching.

Reaching a Shared Understanding through a Shared Language

An output of knowledge crunching and an artifact of the shared understanding is a common Ubiquitous Language (UL). When modeling with stakeholders and subject matter experts everyone should make a conscious effort to consistently apply a shared language rich in domain-specific terminology. This language must be made explicit and be used when describing the domain model and problem domain. The language should also be used in the code implementation of the model, with the same terms and concepts used as class names, properties, and method names. It is the language that enables both the business and development teams to have meaningful communication about the software. You will learn more about the UL in Chapter 4, "Model-Driven Design."

UL is used to bind the code representation of the model to the conceptual model communicated in language and diagrams, which the business can understand. The UL will contain terminology from the business as well as new concepts and terms discovered when modeling the use case of the problem domain. The shared understanding that a UL prevents the need to constantly translate from a technical model to a business model, thus removing the chance of vital insight being lost.

The Importance of Domain Knowledge

Domain knowledge is key, even more so than technical know-how. Teams working in a business with complex processes and logic need to immerse themselves in the problem domain and, like a sponge, absorb all the relevant domain knowledge. This insight will enable teams to focus on the salient points and create a model at the heart of their application's code base that can fulfill the business use cases and keep doing so over the lifetime of the application.

If you can't speak to your business users in simple terms about complex concepts in the problem domain, you are not ready to start developing software within it. To constantly deliver updates at a rapid pace on the complex applications you are building, even as the whimsical business keeps chopping and changing features, you need to refocus your efforts during design and development—you need to focus your team on the business's problems and not just technologies.

The Role of Business Analysts

It may seem that the role of the traditional business analyst is no longer required in the world of DDD; however, this is not the case. A business analyst can still help stakeholders flesh out their initial ideas and capture inputs and outputs of the product. If you have odd whiz-kid developers and are nervous about putting them in front of domain experts, you can also use business analysts as facilitators to help communication. What you don't want to do is remove the direct communication between the development team and the people who understand that part of the business the most.

An Ongoing Process

Knowledge crunching is an ongoing process; teams should continually be working toward a simple view of the problem domain that focuses only on the relevant pieces to aid the creation of a useful model. As you will learn in Chapter 4, model-driven design and the evolution of a domain model is an ongoing process. Many models must be rejected in order to ensure you have a useful model for the current use cases of a system. Collaboration between the development team, business stakeholders, and subject matter experts should not be constrained to the start of a project. Knowledge crunching should be an ongoing concern with the business engaged throughout the lifetime of the application build.

It is important to realize also that with each iteration of the system the model will evolve. It will change as and when new requirements are added. New behaviors and use cases will require changes to the model therefore it is important for the technical team and the business experts to understand that the model is never done; it is only useful for the problems at hand.

As each iteration passes the team's understanding of the problem domain improves. This leads to deeper insight and design breakthroughs that can vastly simplify a model. When the system is in use the model may also need to evolve due to technical reasons such as performance or better understanding of the systems usage. A good model is one that is supple to change; a mature model holds rich and expressive concepts and terminology of the problem domain and is understood by both the business and technical teams.

GAINING DOMAIN INSIGHT WITH DOMAIN EXPERTS

The collaboration between the business and the development team is an essential aspect of DDD and one that is crucial to the success of a product under development. However, it is important to seek out those who are subject matter experts in the domain you are working in and who can offer you deeper insight into the problem area. DDD refers to these subject matter experts as domain experts. The domain experts are the people who deeply understand the business domain from its policies and work flows, to its nuisances and idiosyncrasies. They are the experts within the business of the domain; they will rarely, if ever, have the title of domain expert. Instead, look for the product owners, users, and anyone who has a great grasp and understanding for the domain you are working in regardless of title.

Domain Experts vs Stakeholders

A problem space gives you a set of requirements, inputs, and expected outputs—this is usually provided from your stakeholders. A solution space contains a model that can meet the needs of the requirements—this is where domain experts can help.

As shown in Figure 2-2, if your stakeholder is not a domain expert then his role will differ greatly from that of a domain expert. A stakeholder will tell you what they want the system to do; they will focus on the inputs and outputs. A domain expert on the other hand will work with you in order to produce a useful model that can satisfy the needs of a stakeholder and the behaviors of the application.

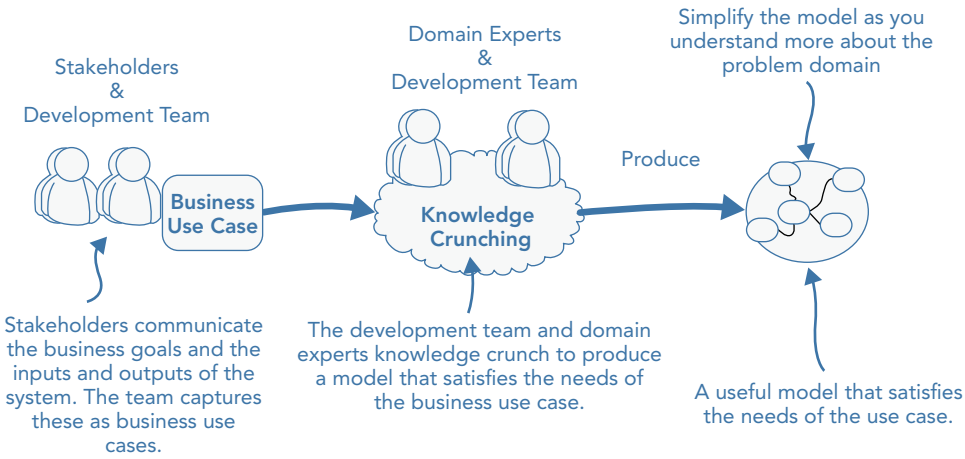


FIGURE 2-2: The roles of stakeholders, domain experts, and the development team.

Deeper Understanding for the Business

Working with a domain expert will not only enable development teams to gain knowledge about the problem domain that they are working in but also help the domain expert to qualify her understanding of the domain. Concepts that may have been implicitly understood by the business are explicitly defined by the development team and domain expert, which leads to improved communication within the business itself.

Engaging with Your Domain Experts

To enable a high level of collaboration, it is recommended that you collocate the development team with domain experts who will be on hand to answer questions and participate during analysis at impromptu corridor or break room meetings; that's something that is lost when the communication is restricted to weekly project meetings. Collaboration is such an essential part of DDD; without it, a lot of the design breakthroughs would not happen. It is this deeper design insight that makes the software useful and able to adapt when the business processes change.

If it's not possible to sit with your domain expert, join her for lunch. Spend as much time as you can with her, and learn as much as possible. If you thrive on delivering high-quality software for personal satisfaction or career goals, then eat, sleep, and breathe the domain—you will be immensely rewarded

MAKE THE MOST OF YOUR DOMAIN EXPERT; YOU NEVER KNOW WHEN SHE WILL BE GONE

Utilize the time spent with your domain expert; don't just ask her to produce sets of requirements or validate ones that you have produced. Actively engage with your domain expert in knowledge-distilling sessions, whiteboard with her, experiment and show how you have a desire to learn more about the domain for which you are producing software.

A domain expert's time will be precious; meeting her halfway and showing a genuine interest will display to her that there is value in sharing knowledge.

PATTERNS FOR EFFECTIVE KNOWLEDGE CRUNCHING

Creating a useful model is a collaborative experience; however, business users can also find it tiring and can deem it unproductive. Business users are busy people. To make your knowledge-crunching session fun and interactive, you can introduce some facilitation games and other forms of requirement gathering to engage your business users.

Focus on the Most Interesting Conversations

Don't bore domain experts and business stakeholders by going through a list of requirements with them one item at a time. As stated before, a domain expert's time is precious. Start with the areas of the problem domain that keep the business up at night—the areas that will make a difference to

the business and that are core for the application to be a success. For example, asking the domain experts which parts of the current system are hard to use, or which manual processes stop them from doing more creative, value-adding work. Or what changes would increase revenue or improve operational efficiencies and save money from the bottom line. It's often a good idea to follow the money and look for the areas that are costing the business money or preventing them from increasing revenue. The most interesting conversations will reveal where you should spend most of your effort on creating a shared understanding and a shared language.

Start from the Use Cases

The best place to start when trying to understand a new domain is by mapping out use cases. A use case lists the steps required to achieve a goal, including the interactions between users and systems. Work with business users to understand what users do with the current system, be it a paper-based process or one that's computerized. Be careful to listen for domain terminology, because this forms the start of your shared language for describing and communicating the problem domain. It's also useful to read back the use case to the domain expert in your own understanding, so they can validate that you do understand the use case as they do. Remember: capture a process map of reality, understand the work flow as it is, and don't try to jump to a solution too quickly before you truly understand and appreciate the problem.

Ask Powerful Questions

What does good look like? What is the success criteria of this product? What will make it a worthwhile endeavor? What is the business trying to achieve? The questions you ask during knowledge-crunching sessions will go a long way toward your understanding of the importance of the product you are building and the intent behind it.

Here are some examples to get your domain expert talking and revealing some deeper insight into the domain:

- Where does the need of this system come from?
- How will this system give value to the business?
- What would happen if this system wasn't built?

> **NOTE** *Greg Young has a great blog post on powerful questions with comments offering many more good examples of questions that can unlock domain knowledge. You can find it at* `http://goodenoughsoftware.net/2012/02/29/powerful-questions/`.

Sketching

People often learn quicker by seeing visual representations of the concepts they are discussing. Sketching simple diagrams is a common visualization technique DDD practitioners use to enhance knowledge-crunching sessions and maximize their time with domain experts.

You can start by drawing sketches on the whiteboard or on paper. If you keep them quick and informal you can quickly iterate on them as the conversation progresses.

Unfortunately, many developers find it difficult to create effective diagrams. However, when drawing sketches, one basic principle can help you to create highly effective diagrams: keep your diagrams at a consistent level of detail. If you're talking about high-level concepts like the way independent software systems communicate to fulfill a business use case, try not to drop down into lower-level concepts like class or module names that will clutter the diagram. Keeping your diagrams at a consistent level of detail will prevent you from showing too much detail or too little detail, meaning everyone can understand what you are trying to convey. It's often better to create multiple diagrams each at a different level of detail.

UML is a wonderful language you can use to communicate complex systems in an understandable manner with little or no technical expertise. However it maybe too formal for rapid knowledge-crunching sessions, as the team will need to retry and model many times. Don't try to use elaborate packages such as Visio or Rational Rose to capture a moving model. Instead, sketch out the model on the whiteboard. You will be less attached to a sketch that took you minutes to draw than a diagram in Visio that took you most of the morning. If you must write up your knowledge-crunching sessions, do it at the end when you know the most about the problem domain.

Class Responsibility Collaboration Cards

Capturing information visually is an effective way to quickly communicate ideas and concepts. However, because DDD is built around the core idea of a shared language, it is important to use knowledge-gathering techniques that focus on creating a concise and powerful language.

CRC (Class Responsibility Collaboration) cards are divided into three areas and contain the following information:

- A class name, which represents a concept in the domain
- The responsibilities of the class
- Classes that are associated and are required to fulfill its purpose

CRC cards focus the team and the business experts on thinking about the language of the concepts in the problem domain.

Defer the Naming of Concepts in Your Model

Naming is important when modeling a domain. However, premature naming can be a problem when you discover through further knowledge crunching that concepts turn out to be different from what you first understood them to be. The issue is the association with the word that was chosen as the original name and the way it shapes your thinking. Greg Young suggests (`http://codebetter.com/gregyoung/2012/02/28/the-gibberish-game-4/`) making up words for areas of the model you are not sure about, using gibberish. I tend to favor using colors, but the idea is the same. Instead of giving areas or concepts of the model real names, use gibberish until you have understood all the responsibilities, behavior, and data of a concern. Deferring the naming of concepts in your model will go a long way toward helping you avoid modeling a reality that you are trying to change to the business's benefit.

Also look out for overloaded terms. The types of names that you want to avoid are XXXXService and XXXXManager. If you find yourself appending service or manager to a class or concept, think more creatively, strive for real intent behind a name. When you feel you have really understood a part of the model, you will be in a better place to give it a sensible and meaningful name.

Behavior-Driven Development

Behavior-Driven Development (BDD) is a software development process, based on Test-Driven Development (TDD), which focuses on capturing the behavior of a system and then driving design from the outside in. BDD uses concrete domain scenarios during conversations with domain experts and stakeholders to describe the behaviors of a system.

Much like DDD, BDD does not focus on the technical aspects of an application. Where it differs is that BDD focuses on the software behavior, how the system should behave, whereas DDD focuses on the domain model at the heart of the software that is used to fulfil the behaviors—a small but important distinction.

BDD has its own form of UL to specify requirements—an analysis language, if you will, known as GWT (Given, When, Then). The GWT format helps to structure conversations with domain experts and reveal the real behaviors of a domain.

To demonstrate how BDD is used, look at how the requirements for a product are captured utilizing user stories.

An example of a feature for an e-commerce site:

Feature: Free Delivery for Large Orders

Some stories for this feature may be:

In order to increase the average order total, which is $50,

As the marketing manager

I would like to offer free delivery if customers spend $60.

Another story example for this feature:

In order to target different countries that have different spending habits,

As the marketing manager

I would like to set the qualifying order total threshold for each delivery country.

A feature describes a behavior that gives value to the business. In a feature story, a role and benefit are also included. The clarity of the role that relates to the feature enables the development team, along with domain experts, to understand who to talk to or who to proxy. The benefit justifies the existence of the feature, helping to clarify why the business user wants the feature.

To better understand a feature and its behavior, use BDD scenarios to describe the feature under different use cases. Scenarios start with an initial condition, the Givens. Scenarios then contain one or more events, the Whens, and then describe the expected outcomes, the Thens.

An example of a BDD scenario:

Scenario: Customer satisfies the spend threshold for free delivery

Given: Threshold for free delivery is set at $60

And: I am a customer who has a basket totaling $50

When: I add an item to my basket costing $11

Then: I should be offered free delivery

A further example:

Scenario: Customer does not satisfy the spend threshold for free delivery but triggers message to up sale

Given: Threshold for free delivery is set at $60

And: I am a customer who has a basket totaling $50

When: I add an item to my basket costing $9

Then: I should be told that if I increase my basket goods total by $1.00, I will be offered free delivery

In addition to being a light way of capturing requirements, the scenarios provide acceptance criteria that developers and testers can use to determine when a feature is complete, and business users can use to confirm that the team understands the feature.

Using this method of capturing requirements removes the ambiguity that traditional requirement documentation can result in while also heavily emphasizing the domain language. The features and scenarios are a product of collaboration between the development team and business experts, and can help to shape the UL.

Rapid Prototyping

Favor rapid prototyping during knowledge-crunching sessions. Business users like nothing more than screen mock-ups, because they reveal so much more about the intent they have behind a product. Users understand UI; they can interact with it and act out work flows clearly.

Another form of rapid prototyping is to capture requirements in code. Greg Young calls this code as analysis; he has a presentation on this topic we can access here: `http://skillsmatter.com/podcast/open-source-dot-net/mystery-ddd`. Again, business users will love the fact that you are writing and creating before their eyes. Starting to code will help focus analysis sessions. Starting to implement abstract ideas from knowledge crunching will enable you to validate and prove your model. It also helps to avoid only abstract thinking, which can lead to analysis paralysis (`http://sourcemaking.com/antipatterns/analysis-paralysis`).

Coding quickly helps create your powerful questions and helps find missing use cases. Use the code to identify and solve the problems. After an hour or so of talking, see if you can create a code model of your brainstorming. I often find that implementing ideas in code helps to cement domain concepts and prove model designs. This process helps to keep the development team engaged and deeply engrossed in learning about the domain as they can start to get feedback on a design immediately.

Remember: Only create a code model of what is relevant and within the specific context to solve a given problem; you can't effectively model the entire domain. Think small and model around rules; then build up. Most important, remember that you are writing throw-away code. Don't stop at the first useful model, and don't get too attached to your first good idea.

Look at Paper-Based Systems

If you are developing a solution for a problem domain that does not have an existing software solution, look to how the business uses language in the current paper-based solution. Some processes and work flows may not benefit from an elaborate model to handle edge cases. Rare edge-case scenarios may be better solved by handing power back to the manual process; modeling this may result in a lot of effort for little business value.

LOOK FOR EXISTING MODELS

Sometimes you don't need to reinvent the wheel. If you are working in a domain that has been around for a long time, such as a financial institution, you can bet that it probably follows a known model. You won't have time to become an expert in your problem domain, so seek out information that teaches you more about the domain. *Analysis Patterns: Reusable Object Models* by Martin Fowler (Addison-Wesley, 1996) presents many common models in a variety of domains that you can use as a starting point in discussions.

Models of the problem domain could already exist in the organization. Ask for existing process maps and work flow diagrams to help you understand the domain at a deeper level. Create a knowledge base like a wiki with terms and definitions to share with the team. Remember that you are only as good as your weakest developer; this applies to domain knowledge as much as technical expertise.

TRY, TRY, AND TRY AGAIN

You won't get a useful model on the first attempt; you might not even get one on the second or third attempts. Don't be afraid of experimentation. Get used to ripping up designs and starting over. Remember that there is not a correct model, only a model that is useful for the current context and the set of problems you are facing.

Understanding Intent

Be wary of customers asking for enhancements to existing software, because they will often give you requirements that are based on the constraints of the current systems rather than what they really desire. Ask yourself how often you have engaged with a user to really find the motivation behind a requirement. Have you understood the why behind the what? Once you share and understand the real needs of a customer, you can often present a better solution. Customers are usually surprised when you engage them like this, quickly followed by the classic line: "Oh, really? I didn't know

you could do that!" Remember: You are the enabler. Don't blindly follow the user's requirements. Business users may not be able to write effective features or effectively express goals. You must share and understand the underlying vision and be aware of what the business is trying to achieve so you can create real business value.

Event Storming

Event Storming is a workshop activity that is designed to quickly build an understanding of a problem domain in a fun and engaging way for the business and development teams. Groups of domain experts, the ones with the answers, and development team members, the ones with the questions, work together to build a shared understanding of the problem domain. Knowledge crunching occurs in an open environment that has plenty of space for visual modeling, be that lots of whiteboards or an endless roll of brown paper.

The problem domain is explored by starting with a domain event; i.e., events that occur within the problem domain that the business cares about. A Post-it note representing the domain event is added to the drawing surface and then attention is given to the trigger of that event. An event could be caused by a user action that is captured and added to the surface as a command. An external system or another event could be the originator of the event; these are also added to the canvas. This activity continues until there are no more questions. The team can then start to build a model around the decision points that are made about events and when they, in turn, produce new events.

Event storming is an extremely useful activity for cultivating a UL as each event and command is explicitly named, this goes a long way to producing a shared understating between the developers and business experts. It can also reveal sub domains and the core domain of the problem domain, which will be covered in detail in Chapter 3, "Focusing on the Core Domain." The biggest benefit however is that it's fun, engaging, and can be done quickly. Alberto Brandolini created this activity and more information can be found on his blog at `http://ziobrando.blogspot.co.uk/`.

Impact Mapping

A new technique for better understanding the intent of business stakeholders is impact mapping. With impact mapping, you go beyond a traditional requirements document and instead you try to work out what impacts the business is trying to make. Do they want to increase sales? Is their goal to increase market share? Do they want to enter a new market? Maybe they want to increase engagement to create more loyal customers who have a higher lifetime value.

Once you understand the impact the business is trying to make you can play a more effective role in helping them to achieve it. Significantly for DDD, you will be able to ask better questions during knowledge-crunching sessions since you know what the business wants to achieve.

Surprisingly, impact mapping is a very informal technique. You simply create mind-map-like diagrams that accentuate key business information. You work with the business so that, like knowledge crunching, it is a collaborative exercise that helps to build up a shared vision for the product. Figure 2-3 shows an example impact map demonstrating an e-commerce company's desired impact of selling 25% more bicycles.

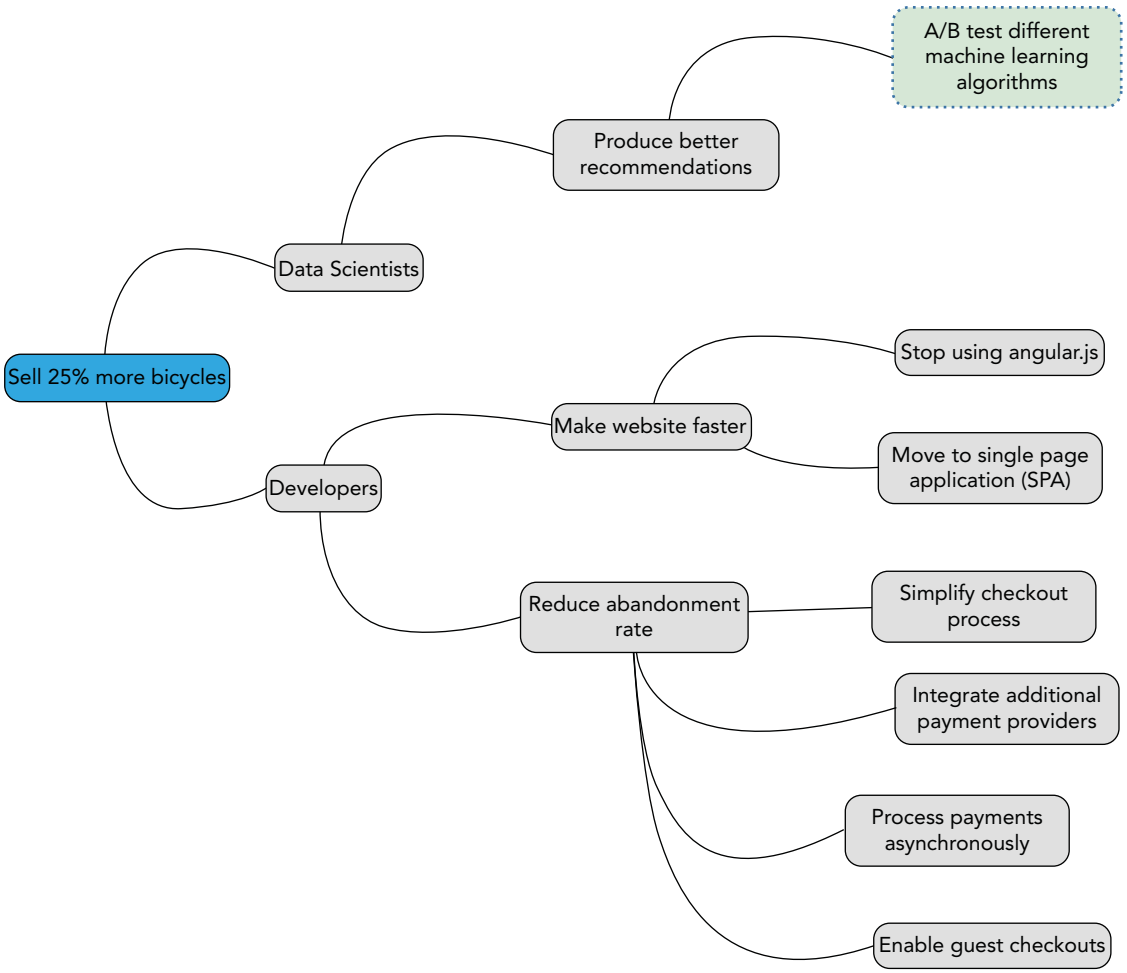


FIGURE 2-3: An impact map.

An impact map, rather obviously, starts with the impact. In Figure 2-3, this is to sell 25% more bicycles, as mentioned. Directly connected to the impact are the actors—the people who can contribute to making the desired impact. In Figure 2-3, that would be developers and data scientists. Child nodes of the actors are the ways in which the actors can help. In Figure 2-3, one way the developers can help to create the business impact is to improve the performance of the website so that people are more likely to make a purchase. Finally, the last level of the hierarchy shows the actual tasks that can be carried out. You can see in Figure 2-3 that one way the developers may be able to make the website faster is to remove slow frameworks.

On many software projects the developers only get the lower tiers of an impact map—what the business thinks they need and how they think the developers should achieve it. With an impact map, though, you can unwind their assumptions and find out what they really want to achieve. And then

you can use your technical expertise to suggest superior alternatives that they would never have thought of.

Some DDD practitioners rate impact mapping very highly, both when applied with DDD or in isolation. You are highly encouraged to investigate impact mapping by browsing the website (`http://www.impactmapping.org/`) or picking up a copy of the book: "Impact Mapping," by Gojko Adzic.

Understanding the Business Model

A business model contains lots of useful domain information and accentuates the fundamental goals of a business. Unfortunately, very few developers take the time to understand the business model of their employers or even to understand what business models really are.

One of the best ways to learn about a company's business model is to visualize it using a Business Model Canvas; a visualization technique introduced by Alexander Osterwalder and Yves Pigneur in their influential book, "Business Model Generation" highly recommended and very accessible reading for developers. A Business Model Canvas is extremely useful because it breaks down a business model into nine building blocks, as shown in Figure 2-4, which illustrates an example Business Model Canvas for an online sports equipment provider.

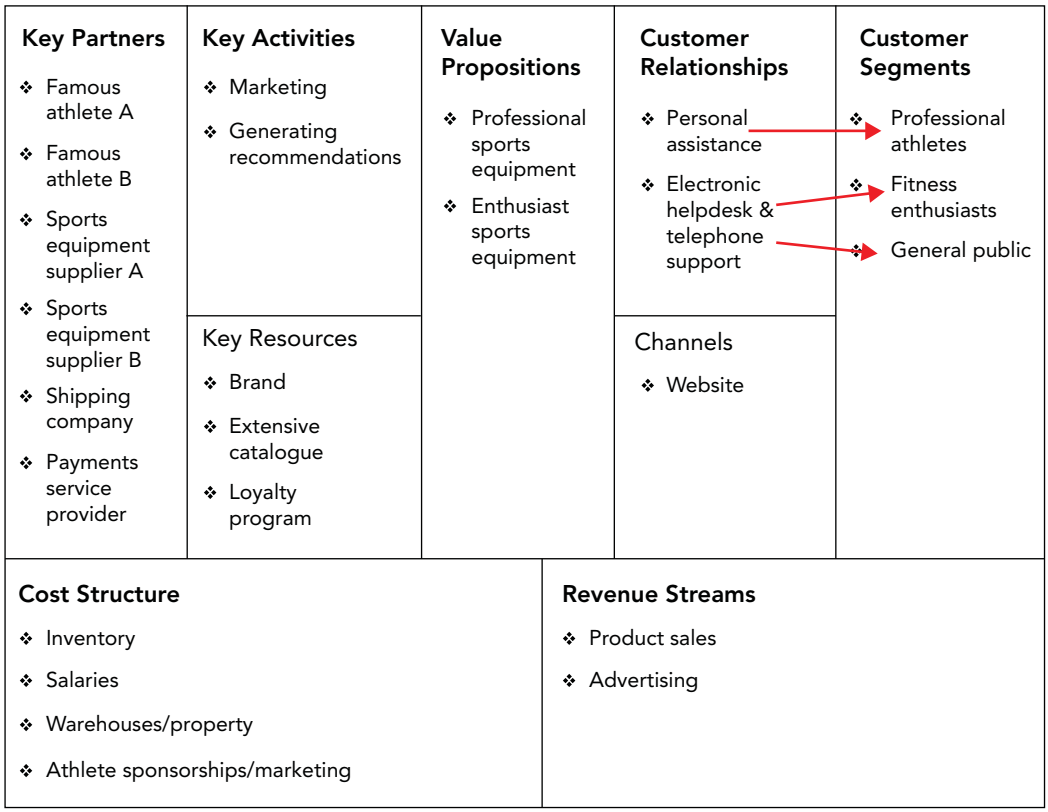


FIGURE 2-4: A Business Model Canvas.

Understanding the nine building blocks of a business model tells you what is important to the business. Key information like: how it makes money, what its most important assets are, and crucially its target customers. Each of the sections of a business model is introduced below. For more information, the "Business Model Generation" book is the ideal learning resource.

- Customer Segments—the different types of customers a business targets. Examples include niche markets, mass markets, and business-to-business (b2b).
- Value Propositions—the products or services a business offers to its customers. Examples include physical goods and cloud hosting.
- Channels—how the business delivers its products or services to customers. Examples include physical shipping and a website.
- Customer Relationships—the types of relationships the business has with each customer segment. Examples include direct personal assistance and automated electronic help facilities.
- Revenue Streams—the different ways the business makes money. Examples include advertising revenue and recurring subscription fees.
- Key Resources—a business's most important assets. Examples include intellectual property and important employees.
- Key Activities—the activities fundamental to making the business work. Examples include developing software and analyzing data.
- Key Partnerships—a list of the business's most significant partners. Examples include suppliers and consultants.
- Cost Structure—the costs that the business incurs. Examples include salaries, software subscriptions, and inventory.

Armed with the information presented by a Business Model Canvas you will be empowered to ask meaningful questions of domain experts and help to drive the evolution of the business—not just the technical implementation. The small effort of finding and understanding your employer's business model is well worth it.

Deliberate Discovery

Dan North, the creator of BDD, has published a method for improving domain knowledge called deliberate discovery (`http://dannorth.net/2010/08/30/introducing-deliberate-discovery/`). Instead of focusing on the framework of agile methodologies during planning and requirement gathering stages, such as the activities of planning poker and story creation, you should devote time to learning about areas of the problem domain that you are ignorant about. Dan states that "Ignorance is the single greatest impediment to throughput." Therefore a greater amount of domain knowledge will improve your modeling efforts.

At the start of a project teams should make a concerted effort to identify areas of the problem domain that they are most ignorant of to ensure that these are tackled during knowledge-crunching sessions. Teams should use knowledge-crunching sessions to identify the unknown unknowns, the parts of the domain that they have not yet discovered. This should be led by the domain experts and stakeholder who can help the teams focus on areas of importance and not simply crunching the

entire problem domain. This will enable teams to identify the gaps in domain knowledge and deal with them in a rapid manner.

Model Exploration Whirlpool

Eric Evans, the creator of Domain-Driven Design, has created a draft document named the Model Exploration Whirlpool (`http://domainlanguage.com/ddd/whirlpool/`). This document presents a method of modeling and knowledge crunching that can complement other agile methodologies and be called upon at any time of need throughout the lifetime of application development. It is used not as a modeling methodology but rather for when problems are encountered during the creation of a model. Telltale signs such as breakdowns in communication with the business and overly complex solution designs or when there is a complete lack of domain knowledge are catalysts to jump into the process defined in the Model Exploration Whirlpool and crunch domain knowledge.

The whirlpool contains the following activities:

- Scenario Exploring
  A domain expert describes a scenario that the team is worried about or having difficulty with in the problem domain. A scenario is a sequence of steps or processes that is important to the domain expert, is core to the application, and that is within the scope of the project. Once the domain expert has explained the scenario using concrete examples the team understands, the group then maps the scenario, like event storming in a visual manner in an open space.
- Modeling
  At the same time of running through a scenario, the team starts to examine the current model and assesses its usefulness for solving the scenario expressed by the domain expert.
- Challenging the Model
  Once the team has amended the model or created a new model they then challenge it with further scenarios from the domain expert to prove its usefulness.
- Harvesting and Documenting
  Significant scenarios that help demonstrate the model should be captured in documentation. Key scenarios will form the reference scenarios, which will demonstrate how the model solves key problems within the problem domain. Business scenarios will change less often than the model so it is useful to have a collection of important ones as a reference for whenever you are changing the model. However, don't try and capture every design decision and every model; some ideas should be left at the drawing board.
- Code Probing
  When insight into the problem domain is unlocked and a design breakthrough occurs the technical team should prove it in code to ensure that it can be implemented.

THE SALIENT POINTS

- Knowledge crunching is the art of processing domain information to identify the relevant pieces that can be used to build a useful model.
- Knowledge is obtained by developers collaborating with domain experts. Collaboration helps to fill any knowledge gaps and fosters a shared understanding.

- A shared understanding is enabled through a shared language known as the ubiquitous language (UL).
- Knowledge crunching is an ongoing process; collaboration and engagement with the business should not be constrained to the start of a project. Deep insights and breakthroughs only happen after living with the problem through many development iterations.
- Knowledge is gained around whiteboards, water coolers, brainstorming, and prototyping in a collaborative manner, with all members of the team at any time of the project.
- Domain experts are the subject matter experts of the organization. They are anyone who can offer insight into the problem domain (users, product owners, business analysts, other technical teams).
- Your stakeholders will give you the requirements of your application but they may not be best placed to answer detailed questions of the domain. Utilize domain experts when modeling core or complex areas of the problem domain.
- Engage with your domain experts on the most important parts of a system. Don't simply read out a list of requirements and ask them to comment on each item.
- Plan to change your model; don't get too attached as a breakthrough in knowledge crunching may render it obsolete.
- When working with domain experts focus on the most important parts of the problem domain; put most effort into the areas that will make the application a success.
- Drive knowledge crunching around the most important uses case of the system. Ask the domain experts to walk through concrete scenarios of system use cases to help fill knowledge gaps.
- Ask powerful questions and learn the intent of the business. Don't simply implement a set of requirements but actively engage with the business; work with them, not for them.
- Visualize your learning with sketches and event storming techniques. Visualizing a problem domain can increase collaboration with the business experts and make knowledge-crunching sessions fun.
- Use BDD to focus on the behavior of the application and focus domains experts and stakeholders around concrete scenarios. BDD is a great catalyst for conversations with the domain experts and stakeholders. It has a template language to capture behavior in a standard and actionable way.
- Experiment in code to prove the usefulness of the model and to give feedback on the compromises that a model needs to make for technical reasons.
- Look at existing processes and models in the industry to avoid trying to reinvent the wheel and to speed up the gaining of domain knowledge.
- Find out what you don't know, identify the team's knowledge gaps early then activate deliberate discovery. Eliminate unknown unknowns and increase domain knowledge early.
- Leverage Eric Evans' Model Exploration Whirlpool when you need guidance on how to explore models. The activities in the whirlpool are particularly helpful when you are having communication breakdowns, overly complex designs, or when the team is entering an area of the problem domain of which they don't have much knowledge.

3

Focusing on the Core Domain

WHAT'S IN THIS CHAPTER?

- Why you should distill a large problem domain
- How to identify the core domain
- How to focus effort on the core domain
- The responsibilities of the supporting and generic domains
- Why not all parts of a system need to be well designed

It's important to understand that not all parts of a problem are equal. Some parts of the application are more important than others. Some parts need more attention and investment than others to make the application a success. During knowledge crunching with domain experts, it's important to reduce the noise of what's unimportant to enable you to focus on what *is* important. Model-Driven Design is hard and should only be reserved to the areas of your systems that are vital to its success. This chapter covers how you can reveal the most important areas of a system and how by using distillation you can focus on those areas. With the knowledge of where to focus you can deeply model what is core, and focus on what will make a difference.

WHY DECOMPOSE A PROBLEM DOMAIN?

Large systems built for complex problem domains will be made up of a combination of components and sub systems, all of which are essential to ensure the entire systems works. However some parts of the solution will be more valuable than others. It is essential therefore to focus effort and attention on the areas that are important to the business. You cannot equally spread effort and quality throughout the entire system, nor do you need to. Trying to strive for equality will result in a loss of focus on the real area that is essential to get right.

In order to understand the most valuable areas of a problem domain we need to distill it to reveal what is core. By breaking up the large problem domain we can more effectively

resource the different areas to ensure the most talented developers are working in the areas of most importance to the business rather than the areas that may be more technically challenging or that utilize new frameworks of infrastructure. The subdomains distilled from the large problem domain are also an input to the way we will architect the solution.

ONE MODEL TO RULE THEM ALL?

It may seem sensible to model the entire problem domain using a single model. However, this can be problematic because it needs to cater to all the needs of your domain. This renders the model either too complex or overly generic and devoid of any behavior. If you have large systems, it is far better and more manageable to break down the problem space into smaller, more focused models that can be tied to a specific context. Remember DDD is all about reducing complexity; a single monothlic model would increase complexity. Instead you should break the problem domain down so that you are able to create smaller models in the solution space.

HOW TO CAPTURE THE ESSENCE OF THE PROBLEM

To know where to focus effort you need to understand what makes the application worth designing in the first place. You need to understand the business strategy and why the existence of the software you are creating will enable it. It is worth asking why the custom software is being written rather than opting for a commercial off-the-shelf product. How will building an application make a difference to the business? How does it fit within the strategy of the company? Why is it being built in-house rather than being outsourced? Does part of the software give the business a competitive edge?

Look Beyond Requirements

Be wary of business users asking for enhancements to existing software, because they will often give you requirements that are based on the constraints of the current systems rather than what they really desire. Ask yourself how often you have engaged with a user to really find the motivation behind a requirement. Have you understood the why behind the what? Once you share and understand the real needs of a customer, you can often present a better solution. Customers are usually surprised when you engage them like this, quickly followed by the classic line: "Oh, really? I didn't know you could do that!" Remember: You are the enabler. Don't blindly follow the user's requirements. Business users may not be able to write effective features or effectively express goals. You must share and understand the underlying vision and be aware of what the business is trying to achieve so you can offer real business value.

Capture the Domain Vision for a Shared Understanding of What Is Core

Before embarking on any product engagement, always ask for a project overview. In any large organization, the process of project inception starts long before the development team gets involved.

Often there will be a small paper on why the company wants to build the software for this initiative over another. This paper often holds the key to the core domain. The paper is a justification on why writing the software is a good idea; study this and pick out what is important. Make it explicit by writing the salient points on the whiteboard so all on the team understand why they are writing the software.

A domain vision statement can be created at the start of a project to explicitly capture what is central to the success of the software, what the business goal is, and where the value is. This message should be shared with the team and even stick it up on a wall in the office as a reminder to why the software is being written.

AMAZON'S APPROACH TO PRODUCT DEVELOPMENT

Amazon has a unique approach when it comes to forming a domain vision statement called *working backwards* (see: `http://www.quora.com/What-is-Amazons-approach-to-product-development-and-product-management`). For new enhancements, a product manager produces an internal press release announcing the finished product, listing the benefits the feature brings. If the intended customer doesn't feel the benefits are exciting or worthwhile, the product manager refactors the press release until the feature offers real value for the customer. At all times, Amazon is focused on the customer and is clear about the advantage a new feature can bring before it sets out with development.

HOW TO FOCUS ON THE CORE PROBLEM

Large problem domains can be partitioned into subdomains to manage complexity and to separate the important parts from the rest of the system. Figure 3-1 shows how in the domain of butchery, a pig is divided into cuts much like a problem space. Understanding the subdomains of your system enables you to break down the problem space. Subdomains are abstract concepts; don't get subdomains confused with the organizational structure of a company. Subdomains represent areas of capability, define business processes, and represent the functionality of a system.

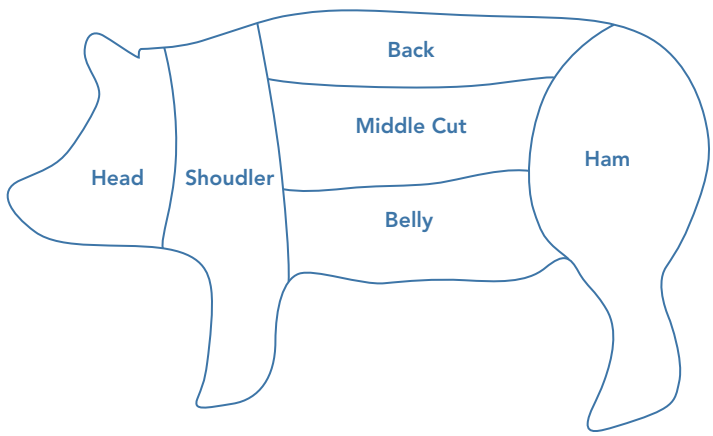


FIGURE 3-1: Cuts of a pig.

Try not to bring technical concerns into conversation when you are analyzing the problem space. Security is a technical concern unless your problem

space is security. Audit trails and logging are also infrastructural concerns. Instead, keep focused on the domain first and foremost.

By distilling the problem domain you reduce complexity by dividing and conquering the problem. Smaller models can be created and understood within the context of a subdomain. This removes the need for a single large model to represent the entire problem domain. Many of these subdomains that are defined may be generic to any enterprise business software, such as reporting and notification needs. These subdomains, which do not define the application, are referred to as generic domains. The areas that distinguish your company's unique product offering from a rival's and define what gives it a competitive edge in the market are known as your core domains. The core domains are the reason why you are writing this software yourself. The remainder of the subdomains that make up large-scale applications are known as supporting domains, which are enablers for the core domain and the system.

Distilling a Problem Domain

Take the domain model of an online auction site, as shown in Figure 3-2. There are many different components that make up the large overall system. Some parts will be found in any online system, but some will be unique to the domain and specific business.

Figure 3-3 shows how the large problem domain is partitioned into subdomains. Membership represents the area of the systems that deals with the registrations, preferences, and details of members. The seller partition represents all the processes and behaviors that deal with seller activities. Auction is the area of the problem domain that deals with managing the timing of auctions and dealing with bid activity. Listings are the catalogues of items that are available on the auction site. Finally, the dispute resolution domain deals with disputes between members and sellers.



FIGURE 3-2: The domain of an online auction site.

The distillation of knowledge after sessions with domain experts should reveal what's unique and important about the application you are about to create. You can separate the subdomains into core, generic, and supporting domains, as shown in Figure 3-4.

In figure 3-4 you can see that the core domains of the online auction site are the seller and the auction. The seller domain contains the ratings for a seller and the domain logic for determining seller fees. The auction core domain is the mechanism for running an auction and handling bids. Both of these areas are vital for the success of the auction site. The membership and listing domains support the core domains by providing bidders the opportunity to create accounts

and find items for sale. The dispute resolution domain is generic in that it can be served using a commercial off-the-shelf package; in this scenario it is merely a ticking system to handle customer dispute cases.

To know where to invest the most effort and quality, it's crucial to understand where the core domains are, because these are key to making the software successful. This knowledge is distilled from knowledge-crunching sessions working in collaboration with domain experts to understand the most important aspect of the product under development.

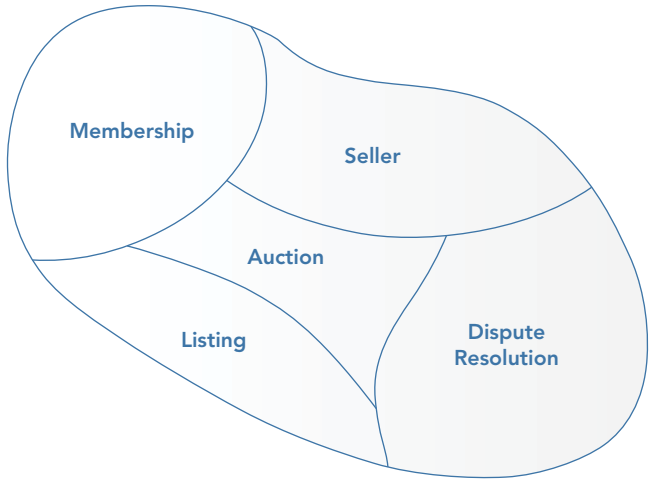


FIGURE 3-3: The domain of an online auction site distilled into subdomains.

Core Domains

To understand what's core to the product that your business is asking you to develop, you need to ask yourself some questions. What are the parts of the product that will make it a success? Why are these parts of the system important? And why can't they be bought off the shelf? In other words, what makes your system worth building?

The core parts of the system represent the fundamental competitive advantage that your company can gain through the delivery of this software. What's core is not always obvious.

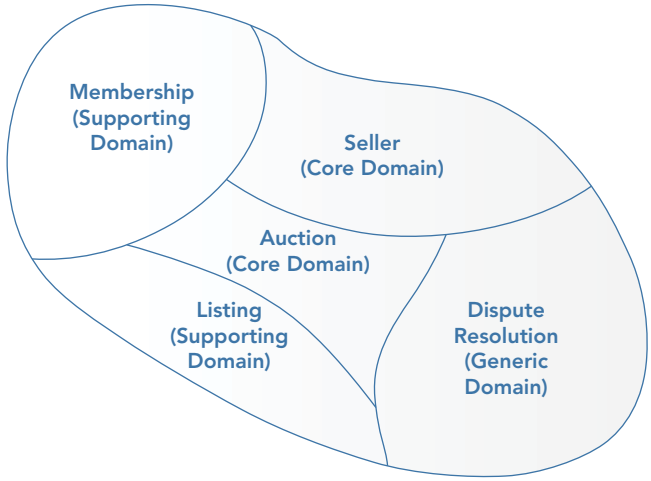


FIGURE 3-4: The distilled domain of an online auction site partitioned into core, generic, and supporting domains.

If the generic domains should be brought in and have little development, the core domain is the polar opposite. The core domains require your best developers—your commandos, if you will. The core domains may not make up the lion's share of your company's technology, but they require the most investment.

What is core certainly changes over time. If you are successful, competitors mimic, so the core domain must evolve to set your business apart from the rest and keep it ahead of the game. It's vital that the development team take this on board and ensure it is in synergy with the values of the software and the business.

THE CORE DOMAIN OF POTTERMORE.COM

Pottermore.com is the only place on the web where you can buy digital copies of the *Harry Potter* books. Like any e-commerce site, it allows you to browse products, store products in a basket, and check out. The core domain of the Pottermore site is not what the customer sees, but rather what he does not. Pottermore books aren't DRM-locked (`http://www.futurebook.net/content/pottermore-finally-delivers-harry-potter-e-books-arrive`); they are watermarked. This invisible watermark allows the books that are purchased to be tracked in case they're hosted illegally on the web. The core domain of the Pottermore system is the subdomain that enables this watermarking technology to deter illegal distribution of a book without infringing on the customer. (The customer can copy the book to any other of his devices.) This is what's most important to the business, what sets it apart from other e-book sellers, and what ensures the system was built rather than being sold on iTunes or other e-book sellers.

Treat Your Core Domain as a Product Rather Than a Project

One of the fundamental shifts in mentality required for writing software for complex core domains, from both the development team and the business, is to focus on the product rather than view it as a standalone project. Often, the development of software for a business product is never finished; instead, the product under development will live through periods of feature enhancements. The software is invested in until it is no longer providing value for the business or it can't be enriched with further modifications.

Your product is an evolving succession of feature enhancements and tweaks. Just as developers iterate, so, too, does the business. A good idea becomes better after it is improved upon and fleshed out over time. Understand the value of the product you are working on and what return on investment (ROI) it brings to the company. Talk to your business sponsors about the future of the product to help focus your coding efforts; know what is important to them.

All too often, software for the core domain of a business isn't viewed as a product that requires care and attention. Instead, business software sacrifices quality and long-term investment for speed to market. Too much emphasis is placed on thinking about the project and looming deadlines, rather than investing in the product for the future. This results in a codebase that is hard to maintain and enhance, and falls into the architectural pattern of the Big Ball of Mud (BBoM), as discussed in Chapter 1, "What Is Domain-Driven Design?"

The flip side, however, is a prolonged release date, which is often nonnegotiable if other business interests depend on the launch date of the software. The solution to this quandary is to look to descope features to keep quality high and the deadline on track. To be in a position to do this, you must understand and share the vision and ultimate goal that the software is expected to meet. This understanding enables you to include only the most important features of the product and ensure that it delivers the value the business expects.

Generic Domains

A generic domain is a subdomain that many large business systems have. An example of a generic domain is an e-mail sending service, an accounts package, or a report suite. These subdomains aren't core to the business, but the business can't operate without them. Because these subdomains aren't core and won't give you a competitive edge, it doesn't make sense to spend a lot of effort or investment in building them. Instead, look to buy in software for generic domains. Alternatively, use junior developers to build these systems, freeing up more experienced resources to work on what's core to your business.

Note, however, that a business defined by communication and targeted e-mails on limited-time offers, like a Groupon or a Wowcher, could have its core domain as a sophisticated e-mail/CRM system. What is core to one business may well be generic to another.

Supporting Domains

The remaining subdomains in the system are defined as the supporting domains. These are subdomains that, although not defining what your system does, help to support your core domains. For example, Amazon's supporting domains would be the functionality that enables a customer to browse a catalog for products. Amazon's product-browsing functionality doesn't define it as a company, and neither is it that different from any other e-commerce site, but it does support the tracking of user journeys to feed a recommendations engine.

As with the generic domains, if possible, you should look to buy off-the-shelf solutions. Failing that, do not invest heavily in these systems; they need to work but do not require your prolonged attention. It's important to note that you may not always need to implement a technical solution to a supporting domain. Perhaps a manual process could meet the needs of the business while developers focus on the core domain.

HOW SUBDOMAINS SHAPE A SOLUTION

Within each subdomain a model can be created. Figure 3-5 shows how the online auction site has been divided into two physical applications. The dispute domain has been fulfilled by an off-the-shelf package while the core and supporting domains have been built using a custom web application.

NOT ALL PARTS OF A SYSTEM WILL BE WELL DESIGNED

Within each subdomain there will be a model that represents the domain logic and business rules that are relevant to that area of the system. Not all of these models will be of equal quality. With an understanding of the various subdomains that comprise your system you can apportion effort accordingly and apply the model-driven design patterns of DDD to the areas that will benefit most.

Don't waste time and effort on refactoring all of your code—ensure your primary focus is on the core domain. If you end up with working but "messy code" for supporting and generic domains then

leave it alone. Good is good enough. Leaving small silos of BBoM is fine as long as they are within clearly defined boundaries. Perfection is an illusion. Perfection should be reserved for only what is core. The business doesn't care about quality code for areas that are required but are not key to the system and which are unlikely to be invested in over time.

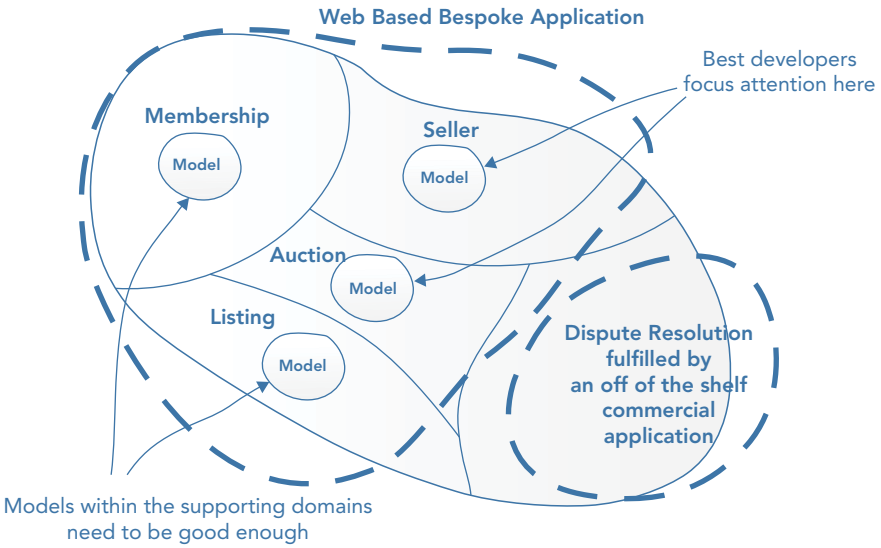


FIGURE 3-5: How a solution maps to the subdomains of the auction system.

Focus on Clean Boundaries Over Perfect Models

The Big Ball of Mud pattern is the most popular software architectural pattern. In large-scale software systems that have evolved over time there are more than likely areas of the system that are not perfect. If you have areas of an application that resemble the BBoM pattern then the best thing to do is to put a boundary around them to prevent the mud spreading into new areas of the application. Figure 3-6 shows the solution space of an application that has defined explicit boundaries between the legacy BBoM and the new models. An anti-corruption layer can be used to prevent one model blurring into another.

ANTICORRUPTION LAYER

An anticorruption layer wraps the communication with legacy or third-party code to protect the integrity of a bounded context. An anticorruption layer manages the transformation of one context's view to another, retaining the integrity of new code and preventing it from becoming a BBoM. You will learn more about the anticorruption layer pattern in Chapter 7, "Context Mapping."

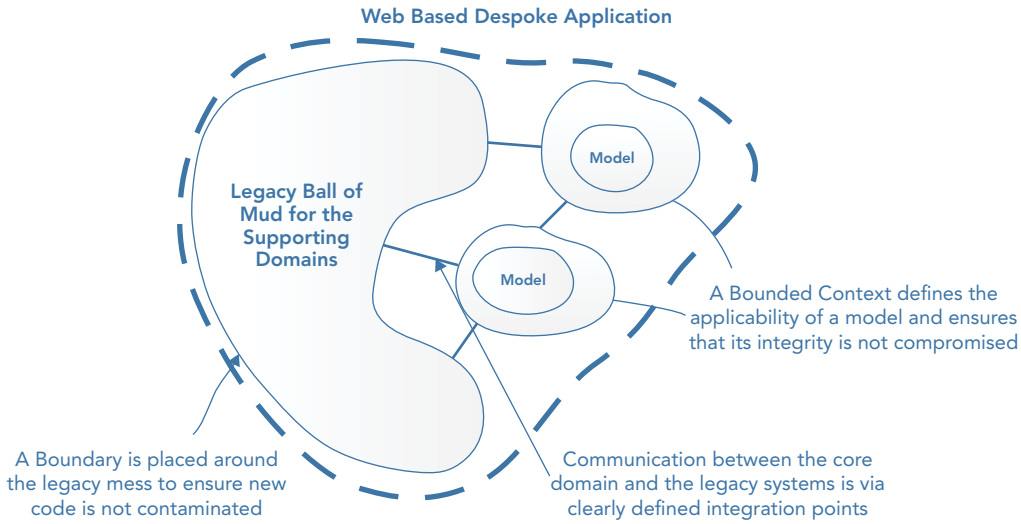


FIGURE 3-6: Dealing with legacy.

The Core Domain Doesn't Always Have to Be Perfect the First Time

In an ideal world, quality software would always be top of your agenda; however, it's important to be pragmatic. Sometimes a new system's core domain could be first to market, or sometimes a business may not be able to tell if a particular idea will be successful and become core to its success. In this instance, the business wants to learn quickly and fail fast without putting in a lot of up-front effort.

The first version of a product that is not well understood by the business may not be well crafted. This is fine, because the business is unsure if it will be invested in over time, and the development team should understand why the business wants speed of delivery over supple design. However, if the product is a success and there is value in a prolonged investment in the software, you need to refactor to support the evolution; otherwise, the technical debt racked up in the rush to deliver starts to become an issue.

Build Subdomains for Replacement Rather Than Reuse

When developing models in subdomains try and build them in isolation with replacement in mind. Keep them separated from other models, legacy code, and third party services by using clean boundaries. By coding for replacement rather than reuse you can create good enough supporting subdomains without wasting effort on perfecting them. In the future they can be replaced by off-the-shelf solutions or can be rewritten as business needs change.

WHAT IF YOU HAVE NO CORE DOMAIN?

There are many reasons that businesses build rather than buy software. If you can do it cheaper, faster, or smarter then it's a good candidate for a custom build. If you find that the software you are building is all generic or is supporting other applications in your enterprise and therefore you have

no core domain then don't try and apply all of the practices and principles of DDD to your project. You can still benefit from following the strategic patterns of DDD but the Model-Driven Design tactical patterns could be wasted effort. You will learn more about when to and when not to apply the model-driven patterns of DDD in Chapter 9, "Common Problems for Teams Starting Out with Domain-Driven Design."

THE SALIENT POINTS

- Distillation is used to break down a large problem domain to discover the core, supporting, and generic domains.
- Distillation helps reduce complexity within the problem space.
- Focus and effort should be invested on the core domain. Use your best developers here.
- The core domain is the reason you are writing the software.
- Consider outsourcing, buying in, or putting juniors on the supporting and generic domains.
- A domain vision statement reveals a shared understanding of what is core to the success of a product. Use domain experts, project initiation documents, and business strategy presentations to help inform the domain vision statement.
- Plan to change the model within the core domain as you learn more about the problem. Don't get too attached to a solution—your core domain may change over time.
- Not all of a system will be well designed. Focus effort on the core domain. For legacy BBoM systems define an anti-corruption boundary to avoid new code becoming tangled within the mess of old.

4

Model-Driven Design

WHAT'S IN THIS CHAPTER?

- The definition of a domain model
- Binding a code model to the analysis model using a ubiquitous language
- The importance of a ubiquitous language
- How to collaborate on a ubiquitous language for improved communication
- Tips on how to create effective domain models
- When you should apply Model-Driven Design

With a deep and shared understanding of the problem domain, along with insight into the core areas that are fundamental to the success of an application, you are now able to focus on the solution space. However, it is important to implement in code the analysis model that was produced during knowledge-crunching sessions; i.e., the model that the business understands. Traditional software processes keep the code model and analysis model separate, which leads to an implementation that rarely resembles the blueprint due to new insight and constraints of the technical solution. DDD acknowledges the need to produce a single model that serves as an analysis model for business people to understand and which is implemented using the same terminology and concepts in code.

This process is known as Model-Driven Design and is heavily dependent on Ubiquitous Language to tie the technical implementation of the model to the analysis model and keep them in sync throughout the lifetime of the system. As well as detailing Model-Driven Design and Ubiquitous Language, this chapter also covers patterns to create effective domain models and the scenarios where Model-Driven Design should be used.

WHAT IS A DOMAIN MODEL?

The domain model, as shown in Figure 4-1, is at the center of Domain-Driven Design (DDD). It is formed first as an analysis model through the collaboration between a development team and business experts during knowledge-crunching sessions. It represents a view, not the reality, of the problem domain designed only to meet the needs of business use cases. It is described in a shared language that the team speaks and the diagrams that the team sketches. When it is expressed as a code implementation, it is bound to the analysis model through the use of the shared language. Its usefulness comes from its ability to represent complex logic and polices in the domain to solve business use cases. The model contains only what is relevant to solve problems in the context of the application being created. It needs to constantly evolve with the business to keep itself useful and valid.

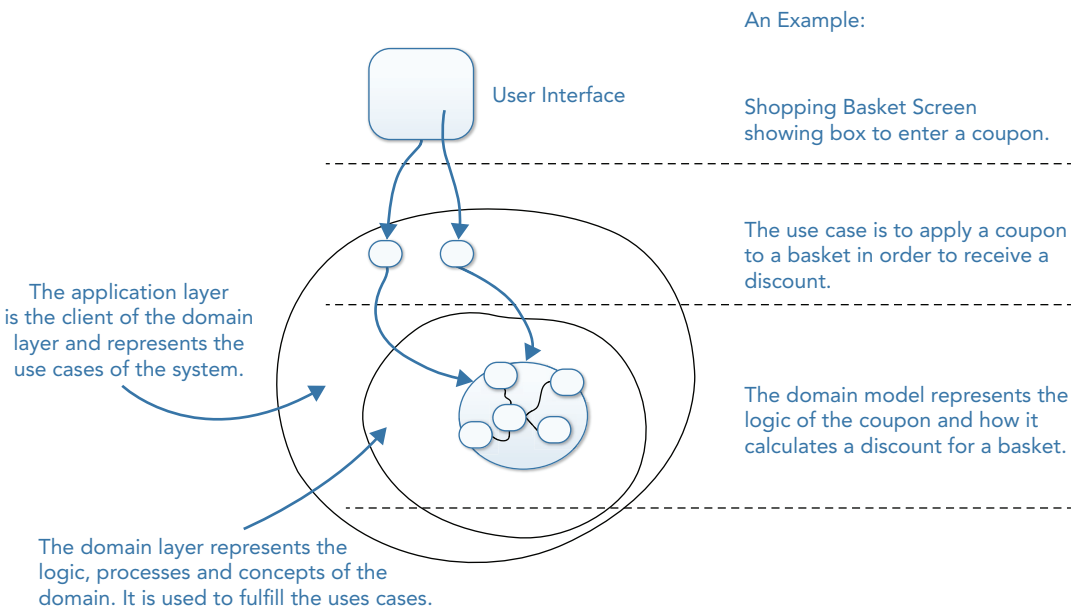


FIGURE 4-1: The role of a domain model.

The Domain versus the Domain Model

The domain represents the problem area you are working within. It is the firm reality of the situation. The domain model, on the other hand, is an abstraction of the problem domain, expressed as a code implementation that represents a view, not the reality, of the problem. This difference is highlighted in Figure 4-2. The usefulness of the domain model comes in its ability to represent complex logic and polices in the domain to solve business problems and not how well it reflects reality. It also exists in a more abstract space: in the language the team speaks and the diagrams it sketches. The model is built from the collaboration between the development team and the business

experts. The model contains only what is relevant to solve problems in the context of the application being created. It needs to constantly evolve with the business to keep itself useful and valid. The domain model only exists to help us solve problems; in order to be effective it needs to have clarity and be free of technical complexities. This way both the business and development teams can collaborate on its design.

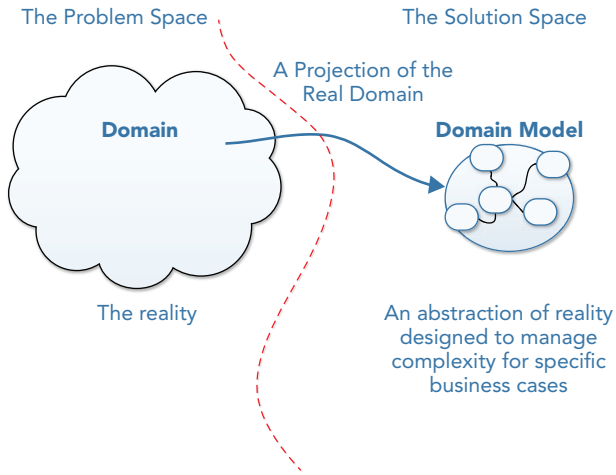


FIGURE 4-2: The domain versus the domain model.

The Analysis Model

Also sometimes known as a business model, an *analysis model* is a collection of artifacts that describe the model of a system. These artifacts can be anything from cigarette packet sketches to informal UML. The analysis model exists to help both the development teams and business users to understand the problem domain; it is not a blueprint for the technical implementation.

The Code Model

DDD doesn't advocate the removal of the analysis model. Far from it, because there is much value to be gained from a model that describes the system. Instead, DDD emphasizes the need to keep the code model, the implementation, in close synergy with the analysis model, the design. This synergy is achieved by ensuring both models are described and share the UL, as shown in Figure 4-3. The utopia is a single model that has value in both implementation and design. To achieve this, it is crucial to keep the code model clean of technical concerns and focused on the domain. In turn, it is important to have an analysis model that can be implemented—not too abstract or high level to be of any use.

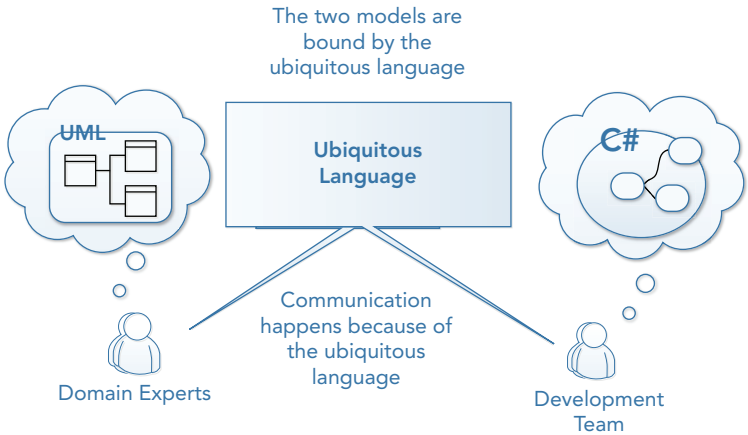


FIGURE 4-3: The binding between the code and analysis model.

The Code Model Is the Primary Expression of the Domain Model

The code model is the realization of the analysis model; it validates the assumptions of the business and quickly highlights any inconsistencies with the analysis model. If, during the creation of the code model, issues are found and logic doesn't seem to fit, the development team should work with the domain experts to resolve these problems. This update to the code model is reflected in the analysis model by making changes to work flow and polices that may not have exposed issues before. Likewise, any changes from a business perspective need to be reflected in the code model. The code and business models are kept in synergy. The code *is* the model; the code *is* the truth.

MODEL-DRIVEN DESIGN

Model-Driven Design is the process of binding an analysis model to a code implementation model, ensuring that both stay in sync and are useful during evolution. It is the process of validating and proving the model in practice, because it's pointless to have an elaborate model if you can't actually implement it. Model-Driven Design differs from DDD in that it is focused on implementation and any constraints that may require changes to an initial model, whereas DDD focuses on language, collaboration, and domain knowledge. The two complement each other; a Model-Driven Design approach enables domain knowledge and the shared language to be incorporated into a software model that mirrors the language and mental models of the business experts. This then supports collaboration because business experts and software developers are able to solve problems together as a result of their respective models being valid. Insights gained in either model are shared and knowledge is increased, leading to better problem solving and clearer communication between the business and development team.

The Challenges with Upfront Design

Historically, the capturing of requirements for software systems was seen as an activity that could occur long before coding was due to start. Business experts would talk to business analysts, who in turn would talk to architects, who would produce an analysis model based on all the information from the problem domain. This analysis model would then be handed over to the developers, along with wireframes and work flow diagrams, so they could build the system

As developers start to implement the analysis model in code, they often find a mismatch between the high-level artifacts produced by architects and the reality of building the system. However, at this stage there is often no feedback loop for developers to talk to the business and architects, so the analysis model can be updated and their input enacted. Instead, the developers diverge from the analysis model, and their implementation often overlooks important and descriptive domain terms and concepts that would have provided deeper insight and understanding of the domain.

As the development team further evolves away from the analysis model, it becomes less and less useful. Crucial insight into the model is lost as the development team focuses on abstracting technical concerns instead of business concepts. In the end the job gets done, but the code bears no reflection to the original analysis model. The business still believes the original analysis models are correct and is unaware of the alterations within the code model.

Figure 4-4 shows how the analysis and code models can diverge from each other if the development team is not involved in domain knowledge crunching.

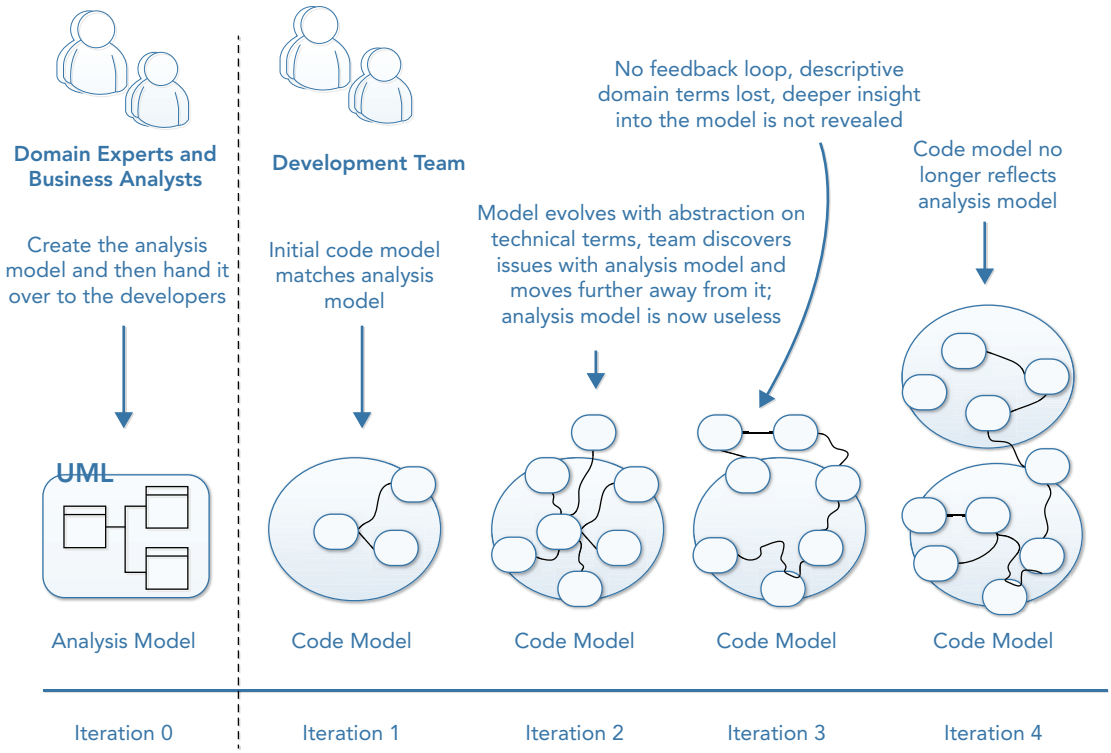


FIGURE 4-4: The problems with upfront design.

The problem is revealed when later enhancements to the codebase are difficult to implement. The difficulties are due to the business experts and developers having different models of the business. The code doesn't have a synergy with the business processes and is not rich in domain knowledge.

Team Modeling

DDD suggests a more collaborative method of capturing system requirements and understanding existing work flow. Emphasis is placed on the entire team, with business experts and architects (as long as they code) having discussions around the problem space. Discussions can include any documentation or legacy code that is related to the system in question. The idea behind the collaborative knowledge-crunching sessions is for the developers, testers, business analysts, architects, and business experts to work as a unified team. This enables the developers and testers to learn about the meaning behind domain terms, and understand complex logic in the problem area. It also enables business experts to experience the modeling techniques employed. With an understanding of modeling, business experts will themselves be able to model and validate designs with the development team.

The sharing of information enables business experts to contribute to the software design, and provides a deeper insight and understanding of the domain to the development team. After a period of time, developers and business experts will discover the relevant information to build an initial model of a problem domain. This initial model is put to the test by using domain scenarios: real problems of the domain to validate its usefulness. Modeling out loud, using the terms and language of the model, can also help to validate early designs.

The important aspect of modeling together is the constant feedback the development team gets from the business experts. This leads to the discovery of important concepts and allows the team to understand what is not important and can be excluded from the model. Breakthroughs in sessions are manifested as simple abstractions that clarify complex domain concepts and lead to a more expressive model.

The model is then expressed in code and the team, along with business experts, can gain fast feedback with early versions of software. Feedback in turn fuels deeper insight, which can be reflected in the code and analysis models, as highlighted in Figure 4-5.

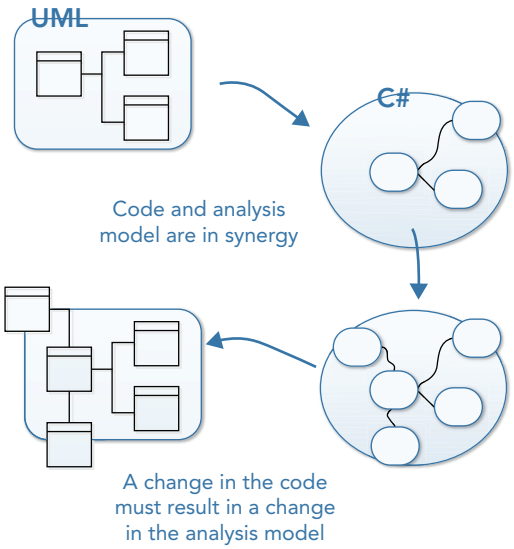


FIGURE 4-5: The code model and the analysis model are kept in synergy.

During each iteration, the development team members may come across parts of the model that they thought were useful and could solve a problem but during implementation had to change. This knowledge is fed back to the business experts for clarification and to refine their own understanding of the problem domain. In this process, the code model and analysis model are one, and a change in one will result in a change to the other.

Figure 4-6 shows how the analysis and code model are in synergy and evolve as one during the creation of a product.

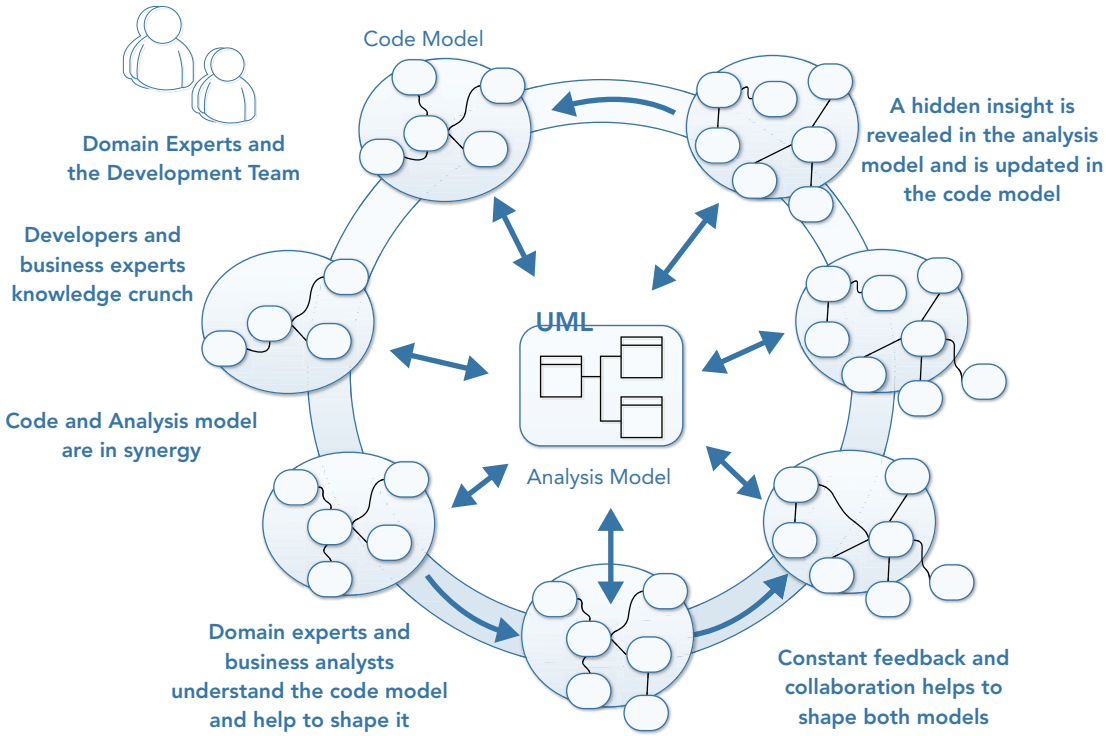


FIGURE 4-6: Team modeling.

USING A UBIQUITOUS LANGUAGE TO BIND THE ANALYSIS TO THE CODE MODEL

The true value of following the Domain-Driven Design (DDD) philosophy is in the collaboration of developers and domain experts to produce a better understanding of the domain. The code that is written is just an artifact of that process, albeit an important one. To reach a better understanding, teams need to communicate effectively. It is the creation of the ubiquitous language (UL) that enables a deeper understanding that will live on after code is rewritten and replaced.

A UL enables teams to organize both the mental and the code model with ease. It achieves an unambiguous meaning because of the shared understanding that it brings to the teams. A UL also provides clarity and consistency in meaning. The language is ultimately expressed in code, but speech, sketch, and documentation are also important for creating the language. The language is constantly explored, verified, and refined with new insights and greater knowledge.

A Language Will Outlive Your Software

The usefulness of creating a UL has an impact that goes beyond its application to the current product under development. It helps define explicitly what the business does, it reveals deeper insights into the process and logic of the business, and it improves business communication.

The Language of the Business

I recently went curtain shopping with my wife. Pleated, hang length, interlining—these were all terms that meant something specific in the domain of curtain makers. Employees in the shop could spend hours describing what they wanted, but that could lead to ambiguity in meaning. But because the employees use terms in the domain of the curtain shop, conversations are kept short and concise, and everybody who understands the domain understands their meanings.

It's the same with carpenters, financial traders, the military, and nearly every domain you can imagine. Each has terms and concepts that mean something very particular to them. A secret language enables complex topics to be covered in concise and meaningful dialogue without the need for confusing babble. It's vital for a development team to understand and collaborate on this language, known as the ubiquitous language (UL). The UL's terms and concepts are used when communicating with team members, including domain experts. They're also used to name classes, methods, and namespaces in the codebase.

Translation between the Developers and the Business

The business language is a rich dialect with highly descriptive and insightful terminology. However, if the development team doesn't engage with domain experts to fully understand the language and use it within the code implementation, much of its benefit is lost. Developers instead create their own language and set of abstractions for a problem domain. Without a shared model and UL, effective communication between the development team and domain experts is a challenge and requires some form of translation. Translation from domain concepts to technical concepts can be time consuming and error prone. Vital domain insights can be lost when the team implementing the code is using a different model than that of the domain expert. Furthermore, lengthy and convoluted communication is required to explain problems that the team faces in a software implementation that could be solved easily with a better understanding of the problem domain and a more efficient way of communicating.

Figure 4-7 shows how a different model in the minds of a developer can make communication with the domain expert problematic. In the code, the developer is focused on technical abstractions, design patterns, and design principles, whereas the domain expert is focused on business process and work flow.

Developers should think in domain terms and concepts, not technical terms, to avoid the need to translate from business jargon into technical jargon. If the development team makes a mistake when translating complex logic and work flow, the chance of creating a bug in code significantly increases.

COLLABORATING ON A UBIQUITOUS LANGUAGE

The rich language that the business uses to describe what it does is one ingredient of the UL. However, when creating a model of the problem domain and implementing it in code, you may need to create new concepts and terminology. The business may use jargon much in the same way that the IT community does, with some terms proving to be too generic. The development team and domain experts need to create new terms and explicitly define the meaning of existing terms to implement the model in code.

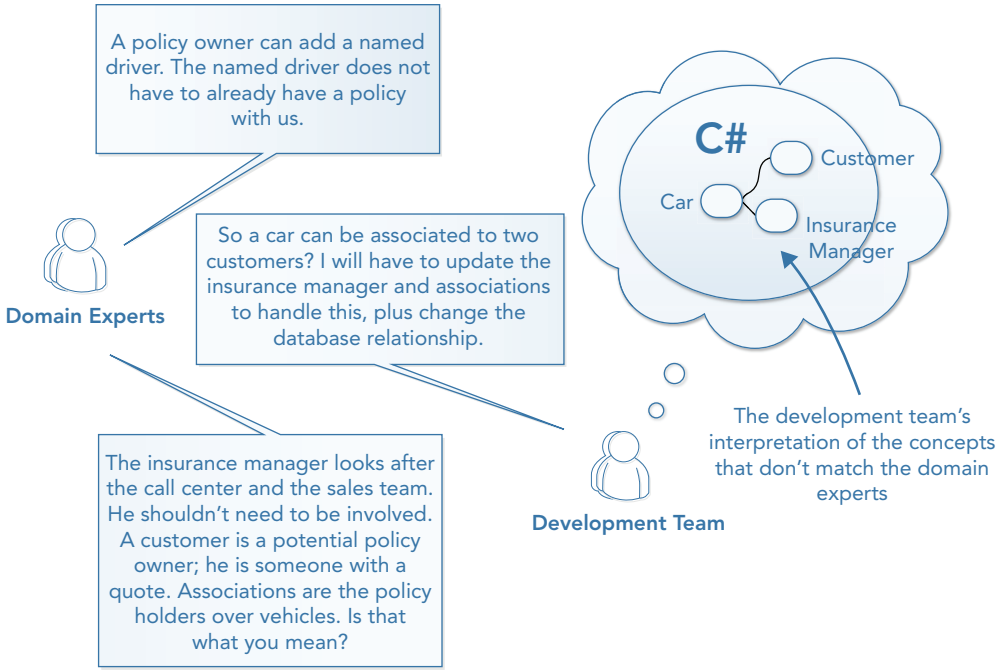


FIGURE 4-7: Translation costs of the project.

As teams are implementing the model in code, new concepts may appear, often highlighted by a collection on logic that needs to be named. These discovered terms need to be fed back to the domain experts for validation and clarification.

Not only must the development team learn the explicit terms and concepts from the business, but they must collaborate with the domain experts to define the assumed or implicit concepts that may not have terminology. These concepts must be named by the entire team and included in the shared UL. The team may also need to create terms for concepts that don't exist in the problem domain but have been discovered and labeled during modeling in the software.

The team members must communicate with each other using the UL. The development team must use it in code, and the domain experts must use it when talking to the team. A shared language removes the need to translate from business speak into technical language and vice versa. It also removes the possibility of ambiguity and misinterpretation because everyone understands the meaning behind the concepts.

The UL should be clear and concise. Technical terms should be removed so they don't distract from business concepts. Likewise, domain terms not relevant for the software under creation must not be allowed to cloud the shared language.

Carving Out a Language by Working with Concrete Examples

As mentioned in Chapter 2, "Distilling the Problem Domain," to better understand the domain you're in, it's a good idea to take specific examples of domain behavior. Concrete examples of real scenarios

help to cement processes and concepts within the domain. However, it's important to reveal the intention of the business process and not the implementation. Talk only in business terms; don't get technical.

In the following dialogue, a business user is describing the process of customers at an e-commerce site requesting a replacement for an order that wasn't delivered:

When a customer doesn't receive her goods, she can request a new order for free. She logs into her account and clicks on the I Have Not Received My Items button. If she has already been flagged as having received a free order, she can't get another one without speaking to customer service. Otherwise, we will send her a free order and update the database to show that this customer has already claimed for a lost item. We will then contact the courier to see if we can claim back the cost of the lost order.

You will notice in the description that the business user is not focusing on the business process, but rather the implementation concerns. The following sentence gives no value or insight into the domain or business process:

She logs into her account and clicks on the I Have Not Received My Items button.

In the next sentence, the business user is already second-guessing how you will implement the business policy. Some experts may have experience with databases and may go as far as suggesting data schemas. Again, this gives the team no deep understanding of the domain:

If she has already been flagged as having received a free order, she can't get another one without speaking to customer service.

From this set of requirements, a team not interested in the domain may simply implement what it is told and end up with a poor model that doesn't reflect the concepts and policies of the domain. The impact of this could be a misunderstanding of what "flagging the customer" means; it may mean more than simply a tick in a database column and perhaps the catalyst for the start of a separate business work flow. Without understanding the domain and the intent of a feature, the developers won't appreciate the repercussions of just implementing what they are told.

Teach Your Domain Experts to Focus on the Problem and Not Jump to a Solution

Training and collaboration will help business people focus on the process rather than the implementation and the problem space rather than the solution space. Next, the previous requirements statement has been rewritten using the language of the domain. It focuses on the business and its processes:

If you have not received an order, you can submit an undelivered order notification. If this is your first claim, a replacement order is created. If you have made a claim before, your claim case is opened and assigned to a customer service representative, who will investigate the claim. In all cases, a lost mail compensation case is opened and sent to the courier with details of the consignment that was undelivered.

In this description, you have discovered many important domain concepts that were missing before. The rewritten prose introduces some terms into the UL, and the terminology of the domain has

been made crystal clear. In fact, the second description doesn't even contain the customer concept; instead, it focuses only on terms that are directly related to the process.

Remember: domain experts have no, or limited, understanding of technical terminology. Keep examples focused on the business, and if domain experts are trying to help you by jumping to implementation details, just gently remind them to focus on the what and the why of a system and ask them to leave the how up to you.

Best Practices for Shaping the Language

The following best practices can help to shape your UL.

- Ensure that you have linguistic consistency. If you are using a term in code that the domain expert doesn't say, you need to check it with her. It could be that you have found a concept that was required, so it needs to be added to the UL and understood by the domain expert. Alternatively, maybe you misunderstood something that the domain expert said; therefore, you should rectify the code with the correct term.
- Create a glossary of domain terms with the domain expert to avoid confusion and to help make concepts explicit.
- Ensure that you use one word for a specific concept. Don't let the domain expert or developers have two words for something because this can lead to confusion, or there might be two concepts with different contexts.
- Stay away from overloaded terms like *policy*, *service*, or *manager*. Be explicit even if it means being wordy.
- Don't use terms that have a specific meaning in software development, such as design pattern names, because developers may assume its implementation rather than behavior.
- Naming is very important. Validate your code design by speaking to your business users about classes. Would a business user understand "Query sent to the cache with, users matched using regex to determine if they get discount". Does your code and concepts make sense when you say them aloud? If not ask your domain expert on how they would name concepts.
- Name exceptions in terms of the UL.
- Don't use the name of a design pattern within your domain model. What does a decorator mean to a business user? Would they understand the role of a factory? Perhaps your business already has the concept of an adapter; the Gang of Four design pattern could confuse them.
- The UL should be visible everywhere, from namespaces to classes, and from properties to method names. Use the language to drive the design of your code.
- As you gain a deeper understanding of the domain you are working in, your UL will evolve. Refactor your code to embrace the evolution by using more intention-revealing method names. If you find a grouping of complex logic starting to form, talk through what the code is doing with your domain expert and see if you can define a domain concept for it. If you find one, separate the logical grouping of code into a specification or policy class.

VALIDATING THE MODEL OUT LOUD

Linguistic consistency can validate the usefulness of a model. For example, listen to conversations about the model, and focus on concepts in the design that don't fit or can't easily satisfy business scenarios. Use the language of the domain to validate solutions.

WHAT IS A SPECIFICATION?

A *specification* represents a business rule that needs to be satisfied by at least part of the domain model. You can also use specifications for query criteria. For example, you can query for all objects that satisfy a given specification.

HOW TO CREATE EFFECTIVE DOMAIN MODELS

Rich domain models are built to satisfy complex problems, the best way to create effective domain models is to firstly focus on areas of the application that are important to the business. Ignore the parts of a system that simply manage data and where most of the operations are CRUD based. Instead look for the hard parts, the areas in the core domain that the business cares passionately about and often the parts that are key to making or saving money.

MODELER'S BLOCK?

Wake yourself up. Brainstorm in code by capturing business requirements as classes and methods. Model on the whiteboard, on paper, with your colleagues, or even with your partner. Warm up your brain by doing something—anything—and the design will eventually bubble up in your consciousness. If that doesn't work, try to spark ideas by sitting in quiet contemplation. Either way, invest in your problem by giving yourself time to think.

Don't Let the Truth Get in the Way of a Good Model

A common misunderstanding is that a domain model should match reality; in fact, you should not look to model real life at all but rather model useful abstractions within the problem domain. Look for commonalities and variations within the problem domain. Understand which are likely to change and are considered complex. Use this information to build your

model. It will be far more useful than identifying nouns and verbs based on the world of the problem domain. Most importantly model only what is needed to meet the need of the business case scenario.

A domain model is not a model of real life; it is a system of abstractions on reality, an interpretation that only includes aspects of the problem domain that are prevalent to solving specific business use cases. A domain model should exclude any irrelevant details of a domain that do not serve to solve problems. The London Tube map shown in Figure 4-8 was designed to solve a problem. It doesn't reflect real life. It isn't useful for calculating distances between landmarks in London, but it is useful for traveling on the underground. It's simple and effective within the context that it was designed for.



FIGURE 4-8: London Tube map bearing little resemblance to the distance between stations.

Because it's not concerned with modeling real life, the domain model cannot be deemed as being wrong or right. Rather, it should be viewed as useful or not for the given problem it is being used to solve.

Creating an effective domain model is fundamental to DDD. It is the artifact of knowledge crunching and sharing, design insight, and breakthroughs. Having a useful model that is rich in the UL is the key to meeting business objectives in the problem domain. Creating a

useful domain model is hard and takes lots of exploration, experimentation, collaboration, and learning.

Model Only What Is Relevant

The domain model exists for one reason: to serve the application under development. Remember to be selective when creating your domain models; you don't have to include everything. Businesses are big and complex with a lot going on. Trying to create that world within a single model would be at best foolish and at worst extremely time consuming and rather pointless. Needless to say, it would be a maintenance nightmare. If you are modeling a large system, break it down to more manageable chunks by clearly sectioning off parts of the model.

Try not to model real relationships; instead, define associations (meaningful connections) in terms of invariants and rules in the system. In real life, a customer has both a credit history and a contact e-mail address, but how often would you come across a rule requiring you to have a good credit history and an e-mail address starting with "A" to be able to purchase an item? Instead, group behavior and data to satisfy the needs of the problem domain rather than what you think might belong together. Remember that you are producing a model to fulfill the needs of a business use case (or set of business use cases), not trying to model real life.

To keep your domain model relevant and focused, you should constantly challenge the model you create against new scenarios and validate your understanding with domain experts. Remove any behavior that is no longer relevant to avoid noise.

Domain Models Are Temporarily Useful

A domain model needs to be constantly refined to continually be useful. A domain model is only ever temporarily useful for a given iteration and set of use cases. Future use cases or changes to the business may render the model useless. The domain model represents an implementation of the shared language that is applicable for only that moment in time. It is with this understanding that developers should not be too attached to an elegant model. They need to be willing to rip up and start again if the model becomes irrelevant.

Be Explicit with Terminology

Being able to communicate effectively is the most important skill for solving problems. A developer's purpose is not to code; it's to solve problems. That's why it's vital to talk to the business you are working for in a language without ambiguity or need of translation. By removing linguistic barriers, domain experts and the development team are free to collaborate, explore, and experiment with designs for a useful model. Technical implementations can then be expressed using the same UL, and any design insights can then be fed back to domain experts for validation without need for translation and loss of meaning.

Limit Your Abstractions

Introduce abstractions for commonality only, and even then try and avoid them. Abstractions come at a cost. It is far better to be explicit than worry about not repeating yourself as trying to tie loosely related concepts under a super class can cause problems with code maintenance.

An abstract class or an interface should represent an idea or a concept in your domain. It is really important to limit abstractions in your code base and only create them for concepts in your domain that have variations. Don't seek to abstract every domain concern. If it's not a variation of a concept then keep it concrete and only abstract if, and when, you create a variation of it. Remember it is always better to be explicit rather than hiding an important domain concept behind layers of needless abstraction.

So when should you abstract? Take the example of traveling to work. The abstract concept would be to commute whereas walking, taking the train, or driving is a variation of that concept; i.e., the concrete implementation. If there were no variation in traveling to work (i.e., we all drove) we would not need to introduce an abstract concept such as commuting.

Focus Your Code at the Right Level of Abstraction

An effective domain model should express the intent of the business use case by aiming code at the right level of abstraction. Readers should be able to quickly grasp domain concepts without having to drill down into the code to understand the implementation details.

Create abstractions at a high level; too many abstractions at a low level will cause a great amount of friction when you need to refactor your model to handle a new scenario or when you have a design breakthrough.

There is always a cost to introducing abstraction so we must be careful to apply it at the right level and to areas of code that will benefit from it. At a low level we should avoid abstraction and instead favor composition of behavior from explicit concrete objects. Abstraction creates a dependency between classes and more dependencies equate to higher code coupling.

Abstract Behavior Not Implementations

You shouldn't have an abstraction that is specific for a particular problem; abstractions represent general concepts such as a `IShippingNoteGenerator` for an order processing application. Variations of this concept could be domestic and international due to the differences between paper work required. Don't automatically abstract concepts that are related. Continuing with the fulfillment domain, don't try and create a common abstraction for courier gateways; they don't represent domain behavior, they are infrastructural concerns. Instead keep these implementations concrete, explicit, and out of the domain model. When we talk about domain concepts we are really talking about domain behavior. Create abstract classes or interfaces based on behavior; keep them small and focused. Ask yourself how much variation is there in domain behavior? Don't force abstraction; use it only when it will help to express concepts clearer in you model.

Just as design patterns emerge when you refactor code so will domain concepts. When they do and you find variations of the concept then you can introduce abstractions in the form of interfaces or abstract classes. Also be mindful of premature refactoring. If you don't know the domain well enough then you may not know the best way to refactor. Instead of painting yourself into a corner let the code grow for a few iterations then look to see natural patterns appear around related behavior. With this clarity you will be in a much better place to start to refactor and introduce abstractions.

Look at all of the abstractions in your system. What do your interfaces and base classes tell you about the domain of your application? They should reveal the major concepts in your system and not just be abstractions of each implementation.

Implement the Model in Code Early and Often

It is vitally important to test your design in code against domain scenarios to ensure your white board thinking can work as well as discovering any technical constraints that require an alteration or compromises to the model. Technical implementations will reveal any problems with the design and will help cement your understanding of a problem domain.

Don't Stop at the First Good Idea

Only stop modelling when you have run out of ideas and not when you get the first good idea. Once you have a useful model start again. Challenge yourself to create a model in a different way, experiment with your thinking and design skills. Try to solve the problem with a completely different model. If you don't get it right the first time, refactor to a better solution. Constantly refactor to your understanding of the problem domain to produce a more express model. Models will change with more knowledge.

Remember a model is only useful for a moment in time; don't get attached to elegant designs. Rip up parts of your model that are no longer useful, and be willing to change when new use cases and scenarios are thrown at your design.

WHEN TO APPLY MODEL-DRIVEN DESIGN

Simple problems don't require complex solutions. You don't need to create a UL for your entire application. Focus your efforts with domain experts on the complex or important core domain. For generic/supporting domains don't waste your efforts, epically if there is no domain logic; doing so will frustrate your busy domain experts and leave them reluctant to help out with the complex areas of your application.

When you come across an area of complexity, you're having trouble communicating with the stakeholder, or your team is working in part of the domain that you don't have much experience with, this is the time to break out, model, and work on the UL.

Always challenge yourself and ask the questions, "Am I working within the core subdomain? Does this problem require a rich domain? Does the business care about this area of the application? Will it make a difference? Is it important to the business and do they have high expectations of it or do they just want it to work?"

If It's Not Worth the Effort Don't Try and Model It

If you have a particularly nasty and complex edge case that is in an area of the system that is not core then you should think about making it a manual process. Not handling edge cases and making them an explicit manual process instead can save valuable time and give you more

resources to work on the core domain. Humans and manual processes are great at edge cases and can often make decisions based on data that would take a considerable amount of time to replicate.

Focus on the Core Domain

The core domain of your application is why it is being built rather than bought. It is what your stakeholders are most passionate about, and where you can have interesting conversations and valuable knowledge-crunching sessions. This is the area where your UL gives you the most value, and demands your focus. Try not to create a rich language for your entire domain because many of your supporting and generic domains do not require one and are a waste of effort. Focus your efforts on what gives you value. Try not to create a UL for everything. Areas and subdomains that are not complex will not benefit from a UL, so don't spread yourself too thin. A core domain is small; focus on it. Creating a UL is costly.

THE SALIENT POINTS

- The domain is the reality of the problem. The domain model is a set of abstractions based on a projection of the domain designed to handle specific business use cases.
- A model is represented as an analysis model and a code model. They are one and the same.
- A domain model exists as an analysis model and a code model. A Model-Driven Design binds the analysis model and a code model through the use of the shared language.
- An analysis model is only useful if it stays in synergy with the code model.
- If you are shaping the analysis or code models, you have to be hands-on and contribute to code. There is a place for architects, but they must be coders as well.
- Code is the primary form of expression of the model and needs to be bound using the ubiquitous language.
- The process of developing a UL is the most important of Domain-Driven Design (DDD) because it enables communication and learning.
- Domain jargon must be explicitly defined to ensure accuracy in meaning because the terminology used in communication is baked into the code implementation.
- Implicit ideas in the domain that the team needs to understand are made explicit and given names that form the shared ubiquitous language.
- Domains are full of specialist terms and language that describe complex concepts in a clean, concise manner.
- Feature stories and scenarios can help you understand the behavior of a system, but a domain expert will help you build a model that can support the specified behavior.
- The ubiquitous language should be used in tests, namespaces, class names, and methods.

- It's important to care about the conversation; a ubiquitous language is about collaboration and not the development team just adopting the language of the business.
- Use domain scenarios to prove the usefulness of the model and to validate the team's understanding of the domain.
- Only apply Model-Driven Design and create a UL for a core domain that will make a difference. Don't apply these practices to the entire application.

5

Domain Model Implementation Patterns

WHAT'S IN THIS CHAPTER?

- The role of the domain layer in an application
- Patterns to implement your domain model in code
- How to select the right design pattern to represent your model

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 5 download and individually named according to the names throughout the chapter.

The focus of DDD is to manage complexity. As you have read, this is achieved by placing a model of the domain at the center of your software to fulfill the behaviors of your application. There are various patterns at your disposal to represent the model in code form. In Chapter 3, "Focusing on the Core Domain," you were introduced to subdomains and the reality that more than one model may exist in large applications. However, not all models will be of equal complexity or importance. Some will contain complex domain logic, while others will simply be responsible for the management of data, therefore it is wise to choose the most appropriate design pattern to represent the model in code.

It is important to understand that there is no best practice when it comes to selecting a pattern to represent your domain logic. As long as you isolate domain logic from technical concerns you can implement Model-Driven Design and hence Domain-Driven Design.

This chapter presents the design patterns at your disposal when implementing a domain model. Along with an explanation of each pattern, advice will be given on when it's most appropriate to use the pattern and when it is best to be avoided.

THE DOMAIN LAYER

The domain layer, at the heart of your application, is the area of code that contains your domain model. It isolates the complexities of the domain model from the accidental technical complexities of the application. It is responsible for ensuring that infrastructure concerns, such as managing transactions and persisting state, don't bleed into the business concerns and blur the rules that exist in the domain. In most cases, the domain layer makes up only a small part of your application. The rest is filled with infrastructure and presentation responsibilities, as you can see in Figure 5-1.

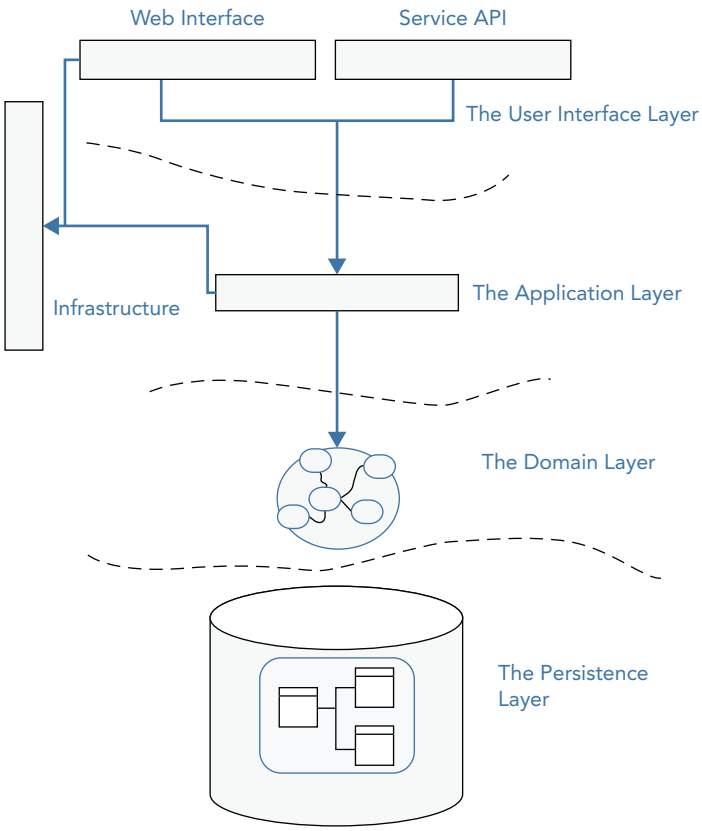


FIGURE 5-1: The code that represents the domain model makes up only a small portion of the overall codebase.

DOMAIN MODEL IMPLEMENTATION PATTERNS

There are various patterns at your disposal to implement a domain model in code. Large systems are not all built in the same way. Some parts are less important than others, and multiple models exist to serve different contexts. Figure 5-2 shows multiple models coexisting in an application. This is because different models are required for different contexts or different teams working on separate

models. The boundaries around the model are explored in more detail in Chapter 6, "Maintaining the Integrity of Domain Models with Bounded Contexts." For now, understand that multiple models can be at play, and you can implement those models in different manners. Figure 5-2 shows an example of how a large application can be segmented into contexts with different patterns used to represent the domain model. The rest of this chapter explores the design patterns that you can follow when modeling a domain. The following three were first presented in the book *Patterns of Enterprise Application Architecture*, by Martin Fowler.

- Domain model
- Transaction script
- Table module

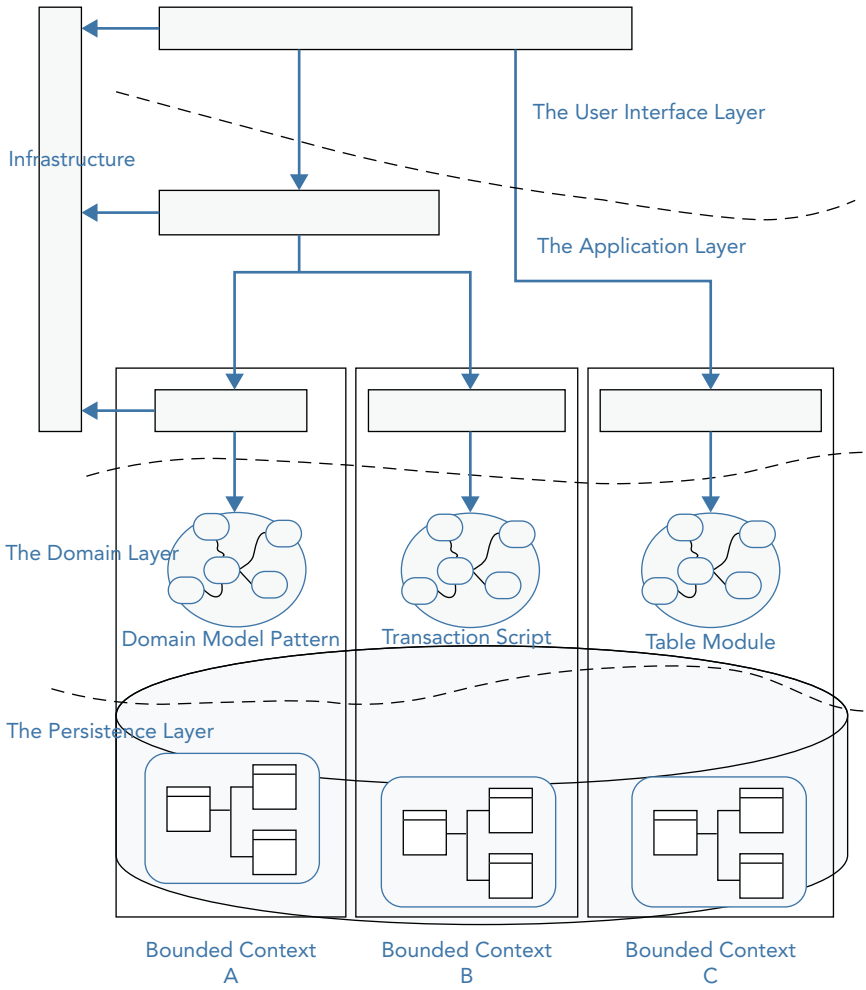


FIGURE 5-2: Multiple domain models implemented in various patterns inside an application.

In addition to Fowler's patterns, you will be introduced to the active record pattern, the anemic domain model pattern, as well as functional patterns for implementing a model in code. Each of the patterns presented in this chapter are useful depending on the complexities of each model in your application.

Domain Model

The domain model pattern, catalogued in Martin Fowler's *Patterns of Enterprise Application Architecture,* is synonymous with DDD because it is a good fit for complex domains with rich business logic. The domain model is an object-oriented model that incorporates both behavior and data. At first glance, it may mirror the data persistence model (data schema if you are using a relational database). Although both contain data, the domain model also incorporates business process and associations, rules, and rich domain logic. DDD offers a number of building block patterns, which are covered in Part III, which will enable you to implement Fowler's domain model pattern more effectively.

The domain model pattern is based on the premise that there is no database; therefore, it can evolve and be created in a completely persistence-ignorant manner. When designing the model, you don't start with a data model; instead, you start with the code model—model-driven as opposed to data-driven design. Only when you have to think about persisting the model can you compromise on the design. Domain objects within the model are known as Plain Old C# Objects (POCO). These classes are free from infrastructure concerns and are completely persistence ignorant. Figure 5-3 shows how the domain model pattern and technical infrastructure are kept separated.

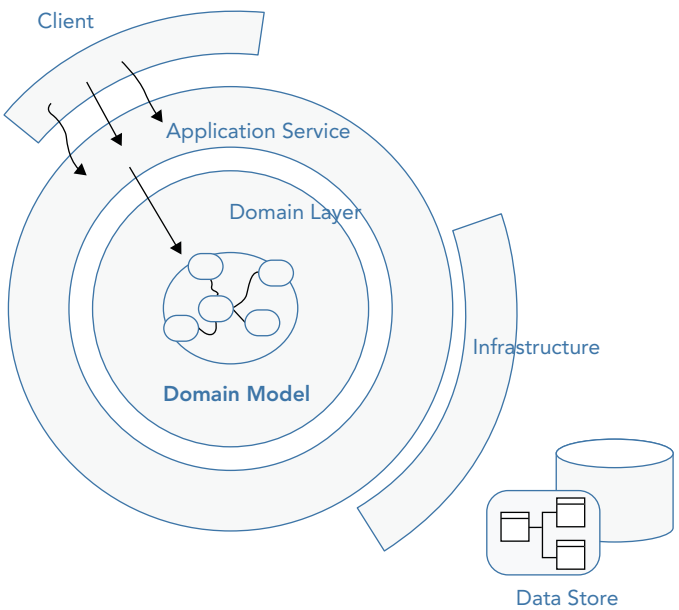


FIGURE 5-3: The domain model pattern.

This ability to focus only on the domain model enables the design of the domain logic to be driven by the abstractions of the domain—hence, DDD. By not thinking about persistence needs, you can build an expressive model purely focused on the domain problem at hand. Of course you will need to persist it and may need to compromise, but you should not think about this when modelling. This keeps the domain model free of infrastructural code and focused only on domain logic.

You can think of a domain model as a conceptual layer that represents the domain you are working in. Things exist in this model and have relationships to other things. For example, if you were building an e-commerce store, the "things" that would live in the model would represent a Basket, Order, Order Item, and the like. These things have data and, more importantly, they have behavior. Not only would an order have properties that represent a creation date, status, and order number, but it would contain the business logic to apply a discount coupon, including all the domain rules that surround it: Is the coupon valid? Can the coupon be used with the products in the basket? Are there any other offers in place that would render the coupon invalid?

Figure 5-4 shows part of the domain model for an online auction site. The objects in the model represent concepts of the problem domain that are used to fulfill the behaviors of an auction. As you can see, the model aligns with the nouns of the auction domain but this is not always the case. In fact you should focus on the verbs and actions of a problem domain when modeling as this will help you concentrate on behavior rather than state, which could end up with your creating an object-oriented representation of the data model.

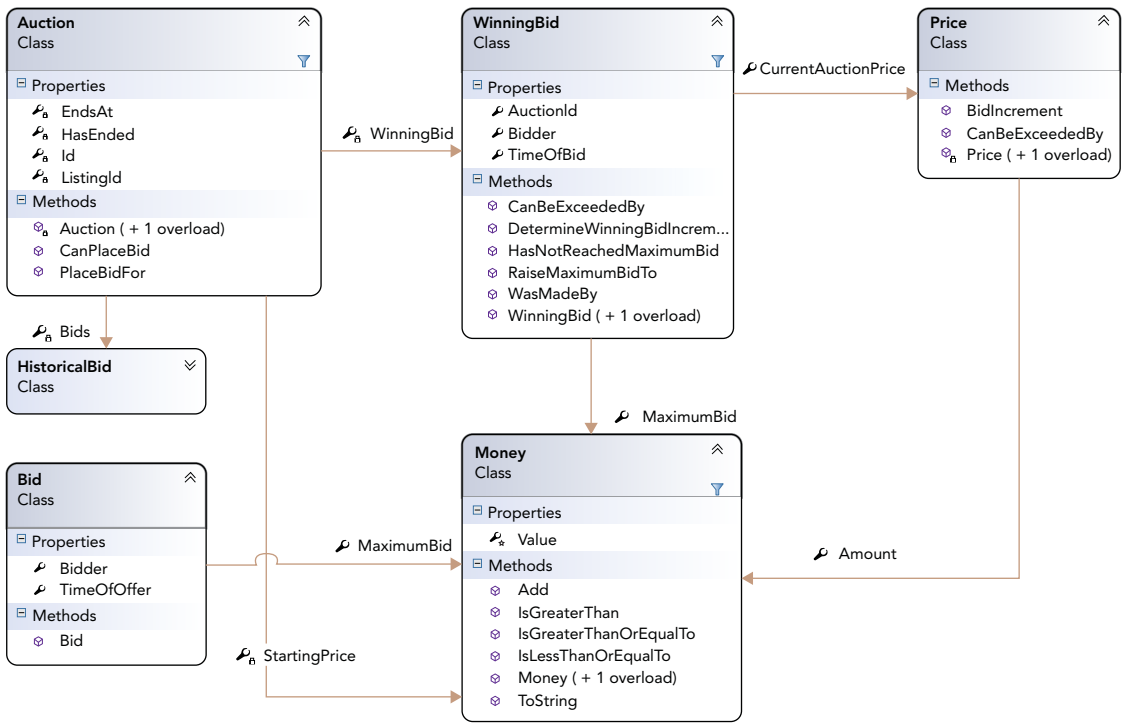


FIGURE 5-4: The domain model of an auction site.

In a domain model each object is responsible for a specific task. Objects work together to fulfill business use cases by delegating to each other. In Listing 5-1 you can see how the `Auction` class delegates to a `WinningBid` to determine the next price of the bid increment.

LISTING 5-1: An Auction Class from a Rich Domain Model

```
public class Auction
{
    ...

    public void PlaceBidFor(Bid bid, DateTime currentTime)
    {
        if (StillInProgress(currentTime))
        {
            if (FirstOffer())
                PlaceABidForTheFirst(bid);
            else if (BidderIsIncreasingMaximumBid(bid))
                WinningBid = WinningBid.RaiseMaximumBidTo(bid.MaximumBid);
            else if (WinningBid.CanBeExceededBy(bid.MaximumBid))
            {
                Place(WinningBid.DetermineWinningBidIncrement(bid));
            }
        }
    }


    ...
}
```

The domain model excels when you have an involved, rich, complex business domain to model. It's a pure object-oriented approach that involves creating an abstract model of the real business domain and is useful when dealing with complex logic and workflow. The domain model is persistence ignorant and relies on mapper classes and other abstraction patterns to persist and retrieve business entities. If you have to model complex logic or part of the problem domain that requires clarity because it's important, or will change often due to continued investment, it's a good candidate for the domain model pattern.

The domain model pattern is no silver bullet as it can be costly to implement. It's the most technically challenging and requires developers with a good grasp of object-oriented programming. The majority of sub systems are CRUD based, with only the core domain requiring the domain model implementation pattern to ensure clarity or to manage complex logic. What you should not do is try to apply the domain model pattern for everything. Some parts of your application will simply be forms over data and will require just basic validation instead of rich business logic. Trying to model everything and apply object-oriented practices would be a waste of effort that would be better spent on your core domain. Software development is all about making things simpler, so if you have complex logic, apply the domain model pattern; otherwise, look for a pattern that fits the problem you have, like the anemic domain model or the table module pattern.

If the portion of the application you are working on does not have frequently changing logic and is merely a form of data, it is best not to try to apply the domain model pattern. At best, incorrectly

using the domain model pattern can lead to a waste of effort; at worst, you introduce needless complexity where a simpler implementation method would have sufficed.

Transaction Script

Of all the domain logic patterns you will read about in this chapter, transaction script is by far the easiest to understand and get up and running with. The transaction script pattern follows a procedural style of development rather than an object-oriented approach. Typically a single procedure is created for each of your business transactions, and it is grouped in some kind of static manager or service class. Each procedure contains all the business logic that is required to complete the business transaction from the workflow, business rules, and validation checks to persistence in the database.

Figure 5-5 shows a graphical representation of the transaction script pattern.

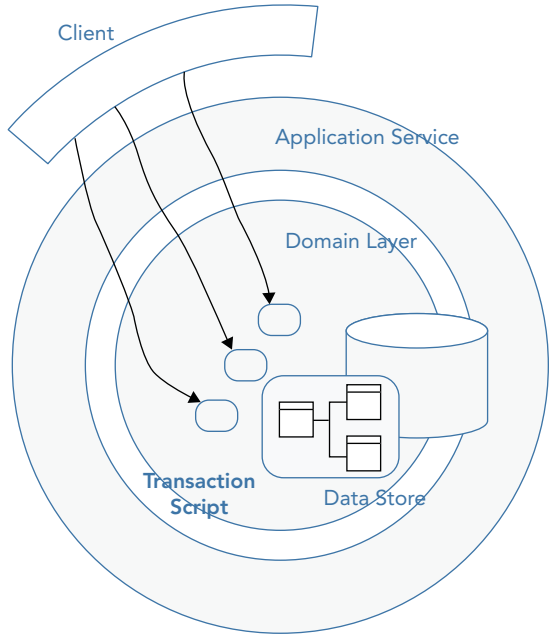


FIGURE 5-5: The transaction script pattern.

Figure 5-6 shows the example signature of an interface that is implementing the transaction script pattern. The two implementations contain all of the logic they require to handle the business cases of creating an auction and bidding on an auction, respectively, including data access and persistence logic, authorization, transactional concurrency, and consistency concerns.

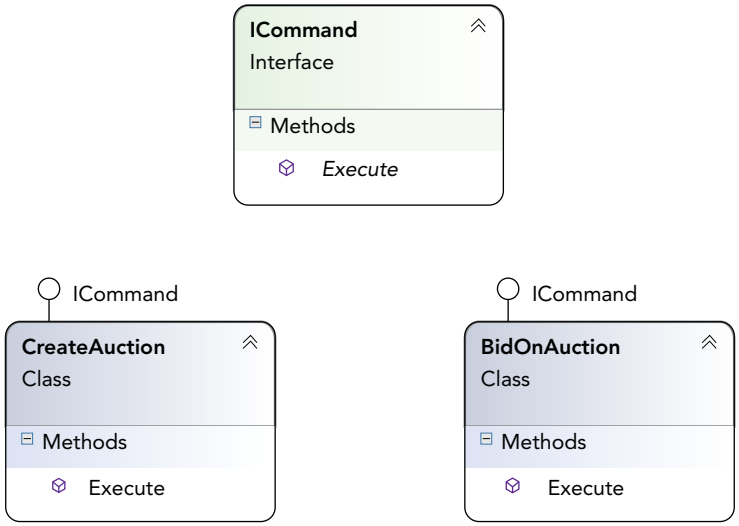


FIGURE 5-6: The transaction script pattern UML.

One of the strengths of the transaction script pattern is that it's simple to understand and can be fast to get new team members up to speed without prior knowledge of the pattern. As new requirements arise, it is easy to add more methods to the class without fear of impacting or breaking existing functionality.

Continuing with the online auction domain, consider Listing 5-2. The transaction script represents the use case of bidding on an auction.

LISTING 5-2: A Use Case Modeled Using The Transaction Script Pattern

```
public class BidOnAuction: ICommand
{
    public BidOnAuction(Guid auctionId, Guid bidderId,
                                decimal amount, DateTime timeOfBid)
    {
        // ...
    }

    public void Execute()
    {
        using (TransactionScope scope = new TransactionScope())
        {

            ThrowExceptionIfNotValid(auctionId, bidderId, amount, timeOfBid);

            ThrowExceptionIfAuctionHasEnded(auctionId);

            if (IsFirstBid(auctionId))
                PlaceFirstBid(auctionId, bidderId, amount, timeOfBid);
            else if (IsIncreasingMaximimBid(auctionId, amount, bidderId))
                IncreaseMaximumBidTo(amount);
            else if (CanMeetOrExceedBidIncrement(amount))
                UpdatePrice(auctionId, bidderId, amount, timeOfBid);
        }
    }

    ...

}
```

As you can see, the entire business case is encapsulated within a single method. The class is dealing with many responsibilities such as data retrieval and persistence, transaction management, as well business logic to fulfill the place of a bid.

Transaction script is a simple procedural pattern that is useful for the parts of your domain that have little or no logic. All logic for an operation is contained within a single service method. Any developer can quickly come to grips with the architecture used to model domain logic. Therefore it is a helpful pattern for teams with junior developers who are not comfortable with object-oriented programming concepts. However, if logic becomes complex, the transaction script pattern can quickly become hard to manage because, by its nature, duplication can occur quickly. If excessive duplication occurs, refactor the code toward the domain model pattern.

The problems with the transaction script pattern are revealed when an application grows and the business logic complexities increase. As an application is extended, so is the mass of methods, making for an unhelpful API full of fine-grained methods that overlap in terms of functionality. You can use sub methods to avoid repetitive code such as the validation and business rules, but duplication in the workflow cannot be avoided, and the code base can quickly become unwieldy and unmanageable as the application grows.

Table Module

The table module pattern maps the object model to the database model. A single object represents a table or view in the database. The object is responsible for all persistence needs along with business logic behavior. The benefit of this pattern is that there is no mismatch between the object model and the database model. The table module pattern is a great fit for Database-Driven Design, so on first glance it might not be a good fit for DDD. However, for simpler parts of the domain that are isolated by a bounded context and that are simply forms over data, this pattern is a good fit and easier to come to grips with than the domain model pattern. If, however, the object model and database model start to diverge, you need to refactor toward the domain model pattern.

Active Record

Active record is a variation of the table module pattern that maps objects to rows of a table as opposed to having objects represent the tables themselves. An object represents a database row (record) in a transient state or under modification.

The active record pattern is a popular pattern that is especially effective when your underlying database model matches your business model. Typically, a business object exists for each table in your database. The business object represents a single row in that table and contains data and behavior, as well as a means to persist it, and methods to add new instances and find collections of objects.

In the active record pattern, each business object is responsible for its own persistence and related business logic.

The active record pattern is great for simple applications that have one-to-one mapping between the data model and the business model, such as with a blogging or a forum engine; it's also a good pattern to use if you have an existing database model or tend to build applications with a "data first" approach. Because the business objects have a one-to-one mapping to the tables in the database and all have the same create, read, update, and delete (CRUD) methods, it's possible to use code generation tools to auto-generate your business model for you. Good code generation tools also build in all the database validation logic to ensure that you are allowing only valid data to be persisted.

Anemic Domain Model

The anemic domain model is sometimes referred to as an anti-pattern. At first glance, the pattern is very similar to the domain model in that you will still find domain objects that represent the business domain. Any behavior, however, is not contained within the domain objects. Instead, it is found outside of the model, leaving domain objects as simple data transfer classes. The major disadvantage of this pattern is that the domain services take on the role of a more procedural style of code rather like the transaction script pattern that you saw at the beginning of the chapter, which brings along the

issues associated with it. One such issue is the violation of the "Tell, Don't Ask" principle, which states that objects should tell the client what they can or can't do rather than exposing properties and leaving it up to the client to determine if an object is in a particular state for a given action to take place. Domain objects are stripped of their logic and are simply data containers.

The anemic domain model pattern is a good candidate for parts of your domain model that have little logic or for teams not very experienced with object-oriented programming techniques. The anemic domain model can incorporate the UL and be a good first step when trying to create a rich domain model.

Anemic Domain Model and Functional Programming

Domain-Driven Design is fully accessible to developers who prefer functional to object-oriented programming. Domain models can easily be built using functional concepts like immutability and referential transparency. Behavior-rich objects are not a necessity, nor is isolating state behind behavioral interfaces. Accordingly, the anemic domain model pattern is actually a fundamentally useful concept when using functional programming as opposed to being an anti-pattern.

It may seem contradictory that domain models are there to facilitate conversations with domain experts, and yet the anemic domain model pattern precludes the ability to represent domain concepts as objects. However, as many modern DDD practitioners assert, the most important domain concepts are verbs—not the nouns like a bank account, but the verbs like transferring funds. With functional programming and the anemic domain model, you still have the power to fully express domain verbs, and consequently to have meaningful conversations with domain experts.

When building functional domain models, it is still possible to have structures that represent domain concepts, even when using the anemic domain model pattern. Significantly, though, they are just data structures with no behavior—so a behavior-rich, object-oriented `BankAccount` entity, as shown in Listing 5-3.

LISTING 5-3: A Behavior-Rich, Object-Oriented BankAccount Entity

```
public class BankAccount
{
    ...

    public Guid Id { get; private set; }

    public Money Balance { get; private set; }

    public Money OverdraftLimit { get; private set; }

    public void Withdraw(Money amount)
    {
        ...
    }

    public void Deposit(Money amount)
    {
        ...
```

```
    }

    public void IncreaseOverdraft(Money amount)
    {
        ...
    }
}
```

Would be modeled as a pure, immutable date structure, as in Listing 5-4:

LISTING 5-4: A Data Transfer BankAccount Object with No Behavior

```
public class BankAccount
{
    public Guid Id { get; private set; }

    public Money Balance { get; private set; }

    public Money OverdraftLimit { get; private set; }
}
```

Having reduced objects into pure data structures, behavior then exists as pure functions, and the challenge is to cohesively group and combine them aligned with the conceptual domain model. One effective option is to group functions into aggregates. The other big divergence with your functions is their structure and responsibility. Since functional programming necessitates immutability, your functions need to return updated data structures rather than mutating the state of existing objects. For example, an object-oriented `ShoppingBasket` may directly update its `Items` collection each time the customer adds a product, as seen in Listing 5-5.

LISTING 5-5: An Object-Oriented ShoppingBasket Class

```
public class ShoppingBasket
{
    ...

    public Guid Id { get; private set; }

    // encapsulated mutable state
    private List<BasketItem> Items { get; set; }

    public void Add(BasketItem item)
    {
        if (this.Items.Contains(item))
            throw new DuplicateItemSelected();
        else
            this.Items.Add(item); // mutating state
    }

    ...
}
```

Taking the functional approach, instead of updating the `Items` collection, a copy of the `ShoppingBasket` is returned that contains an updated, immutable `Items` collection, as seen in Listing 5-6.

LISTING 5-6: A Functional ShoppingBasket Class

```
// pure immutable data structure
public struct ShoppingBasket
{
    public ShoppingBasket(Guid id, IImmutableList<BasketItem> items)
    {
        this.Id = id;
        this.Items = items;
    }

    public Guid Id { get;  private set; }

    public IImmutableList<BasketItem> Items { get; private set; }
}

// all functions for the Basket aggregate
public static class Basket
{
    // pure function, does not belong to any object instance
    public static ShoppingBasket AddItem(
        BasketItem item, ShoppingBasket basket)
    {
        if (basket.Items.Contains(item))
            throw new DuplicateItemSelected();

        // adding creates a new immutable collection
        IImmutableList<BasketItem> items = basket.Items.Add(item);

        // create a new immutable basket
        return new ShoppingBasket(basket.Id, items);
    }
}
```

Notice the `Items` collection can no longer be encapsulated since functions cannot access the `ShoppingBasket`'s private state.

> **WARNING** *Be careful when creating copies of objects and collections. In some languages, you may have a new object reference that points to the existing object (aka a shallow copy). For instance, when copying the* `ShoppingBasket` *in the object-oriented example, both the original and copy may point directly to the same* `Items` *collection. So updating the copy, will actually update the original. To easily solve this problem ensure you create and use only immutable data structures and collections.*
>
> *Most modern languages have native support for immutable collections. C# has immutable equivalents of most mutable collections (*`https://msdn.microsoft.com/en-us/library/dn385366%28v=vs.110%29.aspx`*) and Scala has both a* `collections.mutable` *and* `collections.immutable` *module, for example.*

> **NOTE** *Entities and aggregates are tactical building blocks used by DDD practitioners to represent domain concepts like a bank account, shopping basket, or online date. Part III of this book covers the DDD building blocks in-depth.*

Some programming languages, including Haskell, Scala, and Clojure, make functional programming a first-class feature. But it is still possible to build functional domain models in traditionally object-oriented languages like C# and Java.

> **NOTE** *If you are unfamiliar with functional programming it is definitely worth your time to at least learn the fundamental concepts. The Haskell wiki (*`https://www.haskell.org/haskellwiki/Functional_programming`*) is widely regarded as an excellent beginner's resource, even if you intend to use a language other than Haskell.*

THE SALIENT POINTS

- The domain layer contains the model of the domain and is isolated from infrastructure and presentation concerns.
- The domain model can be implemented with multiple domain logic patterns.
- There may be more than one model at play on a large project and therefore more than a single pattern to represent domain logic.
- As long as the pattern isolates code representing domain logic from technical code then it is a good fit for DDD.
- The domain model pattern is a good fit for a complex problem domain. Concepts in the domain are encapsulated as objects containing both data and behavior.
- The transaction script pattern organizes all domain logic to fulfill a business transaction or use case in a procedural module.
- The table module pattern represents your data model in object form. The Table Module is useful for data-driven models that closely reflect the underlying data schema.
- The active record pattern is like the table module pattern in that it is data-driven but it represents rows in tables as opposed to the tables themselves. It's a good fit for low complexity logic but high CRUD-based models.
- An anemic model is similar to the domain model pattern; however, the model is devoid of any behavior. It is purely a model of the state of an object all behavior resides in service classes that modify.
- Functional programming is an equally valid approach to building domain models.
- When using functional programming, behaviors can be grouped into aggregates (that represent domain concepts) and applied to pure, immutable data structures (that also represent domain concepts).

6

Maintaining the Integrity of Domain Models with Bounded Contexts

WHAT'S IN THIS CHAPTER?

- The challenges of a single model
- The importance of the bounded context
- Carving out and defining boundaries of responsibility in code
- Protecting the integrity of core parts of the domain
- Where to define boundaries

In large and complex applications you will find multiple models at play. Each model will be built to represent a distinct area of the problem domain, with each implementation using an appropriate code design pattern suitable for the complexity of the problem. Ideally you will have a model for each subdomain; however, this might not always be the case as some complex subdomains could contain more than a single model and some models could span two or more subdomains. No matter how many models you have you will find that they will need to interact to fulfill the behaviors of a system. It is when models are combined by teams without a clear understanding of what context they apply to that they are prone to become blurred and lose explicitness, as concepts and logic are intermingled.

Therefore it is vital to protect the integrity of each model and clearly define the boundaries of their responsibility in code. This is achieved by binding a model to a specific context, known as a bounded context. A bounded context is defined based on team's language, and physical artifacts. Bounded contexts enable a model to stay consistent and meaningful, which is vital in managing complexity in the solution space. Diligent use of bounded contexts is essential to being successful with Domain-Driven Design.

THE CHALLENGES OF A SINGLE MODEL

At the core of Domain-Driven Design is the need to create explicit, evolvable models in code that align with the shared conceptual models. As new domain insights are gained, they can be incorporated into the model efficiently. However, if a single model is used for an entire system, concepts from one area of the model can be confused with similar-sounding concepts from another area of the system—and even become coupled to them. Therefore, DDD advocates that you break up a large complex system into multiple code models.

A Model Can Grow in Complexity

Large models accommodate many domain concepts and carry out many business use cases. As a consequence, it is easy to make mistakes and group the wrong concepts together. It can also be very difficult to find what you are looking for. The more the system grows, the more severe these problems become, slowing down the speed at which new features and improvements can be added.

As Figure 6-1 highlights, with each new use case and insight incorporated into the model, the number of concepts and dependencies in the model grows, resulting in increased complexity.

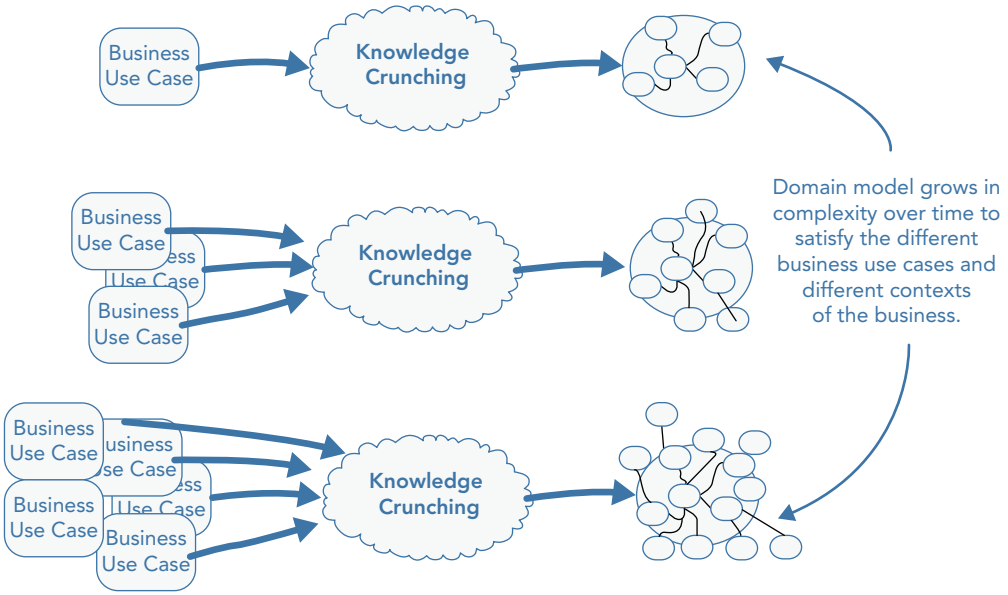


FIGURE 6-1: A model will grow in complexity.

Multiple Teams Working on a Single Model

Complex code is just one of the problems arising from a single model. Collaboration overhead and organizational inefficiencies are also major problems a monolithic model is likely to cause.

As one team wants to release a new feature, they have to check with other teams that their changes can also be deployed. Either the first team will have to wait, or complex branching strategies will be used. As Chapter 11, "Introduction to Bounded Context Integration," explains in more detail, complex branching strategies can be a big hindrance to an organization's ability to frequently and efficiently deliver business value and learn about their customers.

Continually requiring teams to collaborate on the design of new features or to plan releases is an unnecessary inefficiency. As each team works with their own domain expert and tries to drive their model in different directions, dragging other teams along with them is mutually wasteful. The more teams, the more expensive the collaboration overhead, and the more complex the codebase, as Figure 6-2 illustrates.

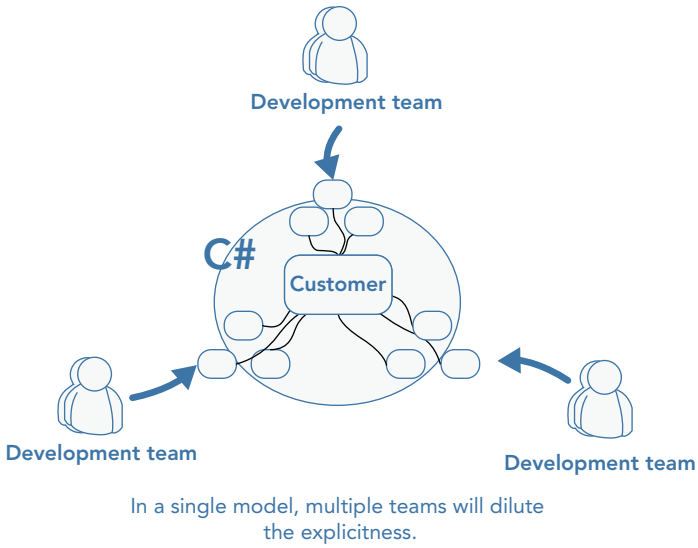


FIGURE 6-2: Complexity in a model increases with multiple teams.

If multiple models are used instead, teams can iterate on their models and deliver new value frequently and efficiently because they do not have to synchronize with other teams or concern themselves with concepts from other teams' models.

You may be concerned about a team's duplicating code in each of their models. But focus on the benefits that arise by removing dependencies between teams. Essentially, it is Ok to duplicate code between models because the concepts are not the same.

Ambiguity in the Language of the Model

One of the epiphanies that DDD practitioners have is the realization that some concepts in a system are very similar—they might even have the same name. Yet actually, they mean very different things to different parts of the business. As Figure 6-3 illustrates, the "Ticket" concept means different things to the Sales and Customer Service departments.

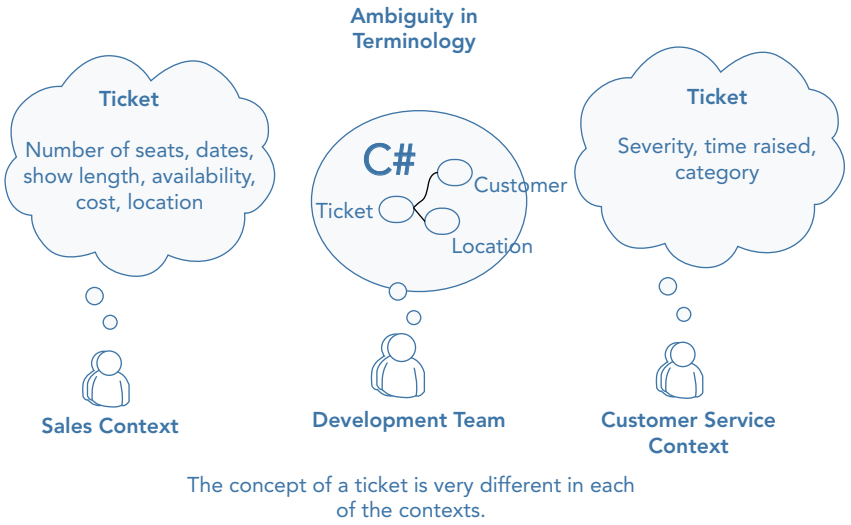


FIGURE 6-3: Domain terms mean different things in different contexts.

Once you accept that names can have different meanings in different contexts, it's easier to accept that multiple smaller models are more effective than a single large one. You can also have more meaningful discussions with domain experts. For example, based on Figure 6-3, when talking to the Sales manager about tickets, you know she cares about the cost and location of an event; whereas discussion about tickets with the Customer Service manager will be focused on the severity and category of problems raised by customers.

The Applicability of a Domain Concept

Sometimes, a single physical entity in the problem domain can mistakenly be classified as a single concept in code. This is problematic when the physical entity actually represents multiple concepts, that each mean different things in different contexts. The classic example is a product.

Figure 6-4 shows how products mean different things in different contexts. It is a concept that must be acquired with a profitable margin and acceptable lead time to the Procurement team. Yet to the Sales team a product is a concept with images, size guides, and belongs to a selling category—none of which are relevant to the Procurement team, even though it is the same physical entity in the problem domain.

> **NOTE** *You learn in Part II how correlation IDs are used to join up the lifecycle of a physical entity that exists in multiple contexts (like the product example).*

When a physical entity, such as a product, actually represents multiple domain concepts, it is often modeled as a single concept by developers. Unfortunately, it's very easy to fall into the trap of thinking that because a product can be a physical item that it should be modeled as a single class in code. This leads to coupling, as each model shares the same product class, as shown in Figure 6-5.

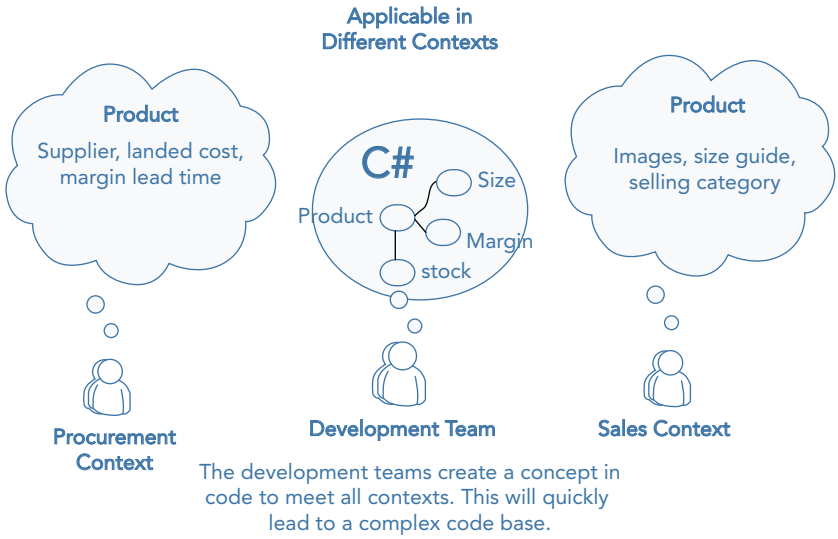


FIGURE 6-4: The same concept should be understood within different contexts.

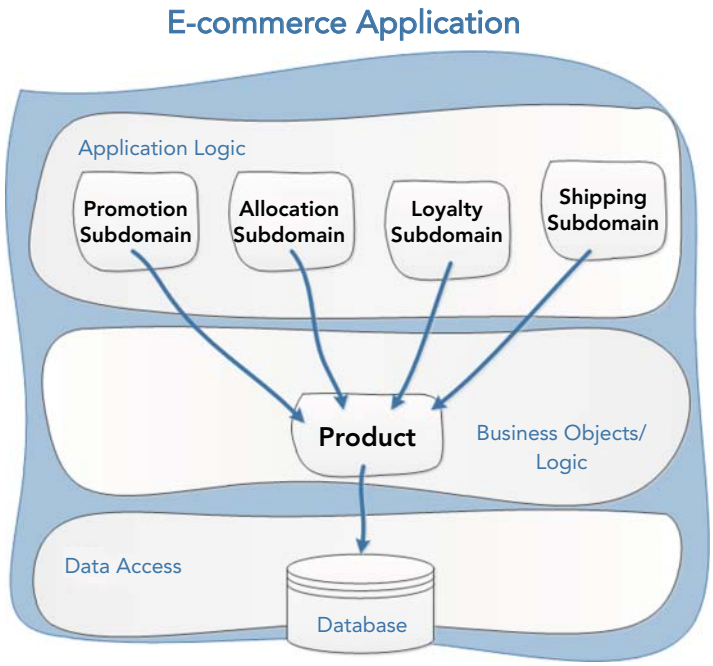


FIGURE 6-5: A single view of an entity in the domain for all subdomains can quickly become a problem.

As discussed previously, when multiple contexts are coupled, code can become excessively complex and the collaboration overhead between teams can become excessively costly. The shared class, in this example product, also violates the Single Responsibility Principle (SRP), since there are four contexts that all want it to change for completely different reasons.

When there are no boundaries in the code, it is too easy for coupling to occur, as with the product shown in Figure 6-5. A better solution that reduces the coupling would be for each context—Promotion, Allocation, Loyalty, and Shipping—to have its own model. Each model would then contain a unique representation of a product that only satisfies the needs of the model's context. Figure 6-6 shows the multiple responsibilities of the shared `Product` class, indicating which model each of them should really belong in.

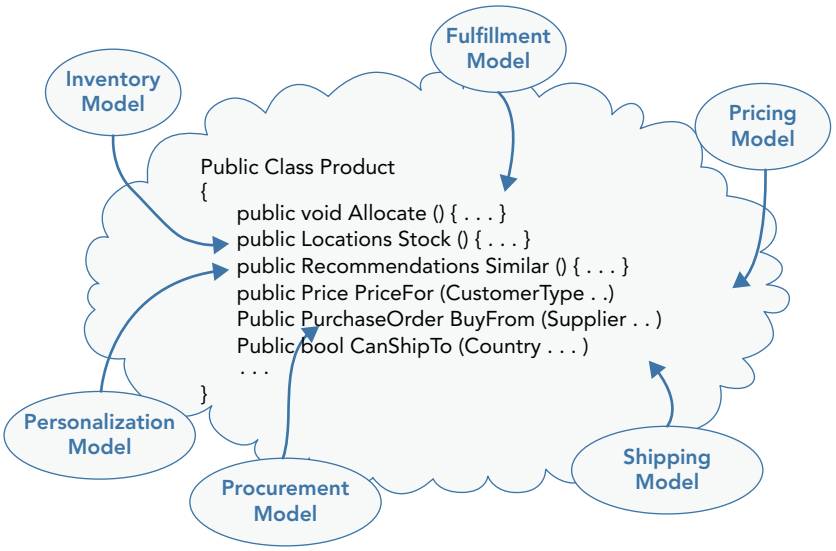


FIGURE 6-6: The product, an implementation of the god object antipattern.

> **NOTE** *The* `Product` *class in Figure 6-6 is a good example of the BBoM pattern discussed earlier. A change to logic in one of the subdomains has an undesired ripple effect to unrelated subdomains because of the interwoven code and the lack of clearly defined boundaries of responsibility.*

Integration with Legacy Code or Third Party Code

Another reason to prefer smaller models is that integrating with legacy code or third parties can be less problematic. Adding new features to a monolithic codebase can be painful when there is lots of legacy code. You want to add clean, new, insightful models that you created with domain experts, but the limitations of legacy code can constrain the expressiveness of your design. But if you have smaller models, not all of them will need to touch the legacy code.

A number of patterns, discussed in Chapter 11, show how it is easier to apply DDD to legacy systems when you have multiple smaller models to work with.

Your Domain Model Is not Your Enterprise Model

Having a single model of the entire system is useful in some scenarios, including business information (BI) and reporting. However, the enterprise model is not the best solution for creating an evolvable domain model that explicitly expresses domain concepts. Nor is an enterprise model suitable for iterative development processes that aim to deliver business value frequently.

Figure 6-7 shows how you can have the best of both worlds—a unique model for each context and an enterprise model for BI.

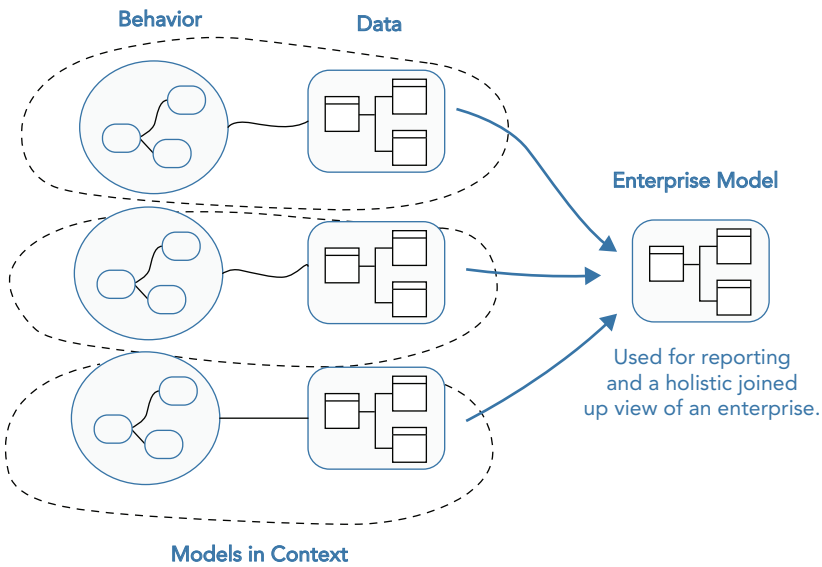


FIGURE 6-7: The difference between an enterprise model and a domain model.

> **NOTE** *Part II and III of this book show strategies, like publish-subscribe, that can be used to transport data from bounded contexts to a data warehouse so that you can create an enterprise model.*

USE BOUNDED CONTEXTS TO DIVIDE AND CONQUER A LARGE MODEL

A bounded context defines the applicability of a model. It gives clarity on what a model is used for, where it should be consistent, and what it should ignore. A bounded context ensures that domain concepts outside a model's context do not distract from the problem it was designed to

solve. A bounded context makes it explicit to teams what the model is responsible for and what it is not.

Context is an important term in Domain-Driven Design. Each model has a context implicitly defined within a subdomain. When you talk about a product in the context of the fulfillment subdomain, you don't need to call it a product that can be fulfilled; likewise, when talking in the context of shopping, it's not a saleable product. It's simply a product in a defined context.

When communicating with domain experts or other members of the development team, you should ensure that everyone is aware of the context you are talking in. The context defines the scope of the model, limiting the boundaries of the problem space, enabling the team to focus without distractions.

In Chapter 4, "Model-Driven Design," you are introduced to the concept of the ubiquitous language (UL) and the importance of models defined in a context that are free from linguistic ambiguity. The context refers to the specific responsibility of the model, which helps to decompose and organize the problem space. A bounded context takes the idea of a model in context further by encapsulating it within a boundary of responsibility. This boundary is a concrete technical implementation, as opposed to the context that is more abstract. The bounded context enforces communication in such a manner as to not lessen the purity of the model.

A bounded context is first and foremost a linguistic boundary. When talking with domain experts, if you feel a sentence requires a context, this is a big hint that you need to isolate that model within a bounded context.

BOUNDED CONTEXTS = BORDER CONTROL

Treat bounded contexts like the borders of a country. Nothing should pass into the bounded context unless it goes through the border control and is valid. Just like countries where people speak a different language, so does the code within your bounded context. Be on your guard in case people try to bypass your borders and don't adhere to your rules and language. One of the most important parts of DDD is the protection of boundaries. A model is defined in a context. This should be followed through to the implementation in the code; otherwise, you will find yourself in a BBoM. Figure 6-8 continues the example of an ambiguous product concept; you see the concept of a product existing without an explicitly defined context. It has been distorted to satisfy the needs of many different scenarios. Without enforcing a boundary around the model and defining it within a specific context, you end up with a mass of sprawling code.

Figure 6-9 shows how a product can be a smaller more focused concept when applied to a specific context. It is important when developing the application that you isolate models within bounded contexts to avoid the blurring of responsibilities that can lead to code that resembles a BBoM.

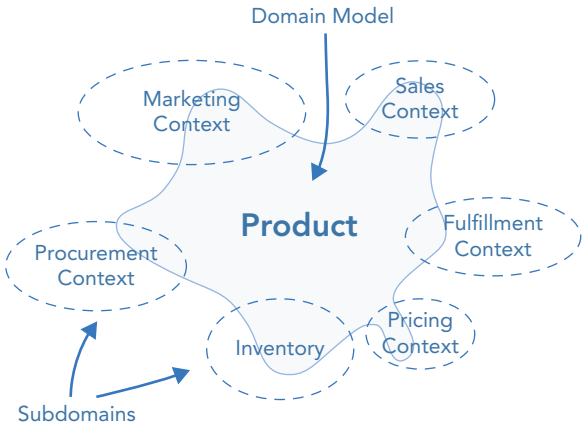


FIGURE 6-8: Putting terms into context and identifying multiple models.

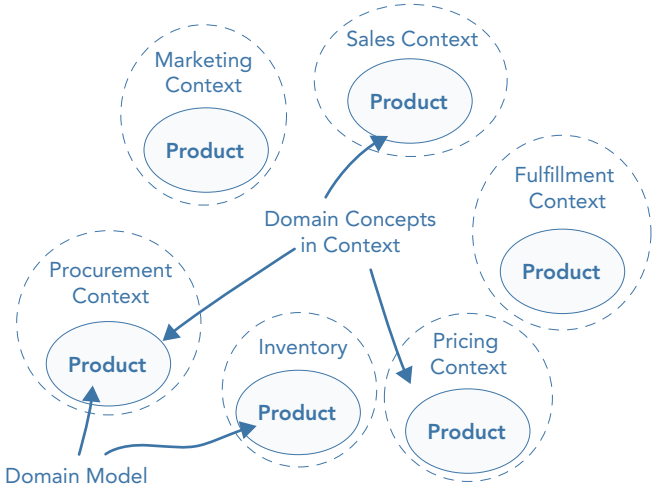


FIGURE 6-9: Define each model within its own context.

CODE ORGANIZATION IS WHAT MATTERS

As a developer, your focus should be on organizing code so that you can manage solutions for complex problem domains. Bounded contexts help to organize code at a macro level—a skill that you should rank high in importance.

Defining a Model's Boundary

The need for bounded contexts is clear in larger systems, but the process of identifying bounded contexts and their boundaries is challenging. Fortunately, it's not an up-front decision you have to get perfectly correct. As you learn more about the domain, you can adjust the boundaries of your bounded contexts.

There are two aspects of a problem domain that you can use as a guide to identifying bounded contexts—terminology and business capabilities. As you've seen previously in this chapter, the same term can have different semantics in different contexts. If you can delineate a domain model based on a change in the meaning of a word or phrase, you will very likely have identified the boundary of a bounded context. Business capabilities are often easy to discern but can be misleading. For example, if a business has a Sales department and a Customer Service department, there is very likely to be a sales and customer bounded context. But that's not always true, so it's important not to blindly model business capabilities.

Outside of the problem domain, team structure and location can also be a big influence on context boundaries, as can integrating with legacy or third-party systems.

Size, though, is not a guideline for delineating bounded contexts. No absolute or relative value can tell you how many classes or lines of code you need. A bounded context's size is dependent mostly on aspects of the problem domain. Some bounded contexts may, therefore, be large while others are small.

To summarize, context boundaries can be influenced by the following:

- Ambiguity in terminology and concepts of the domain
- Alignment to subdomains and business capabilities
- Team organization and physical location
- Legacy code base
- Third party integration

> **NOTE** *In Part II, you see practical examples of how individual bounded contexts can be broken down into smaller modules or components while retaining a faithful representation of the domain.*

Define Boundaries around Language

It's important to be explicit about what context you're using when talking with domain experts, because terminology can have different meanings in different contexts. As repeated throughout this chapter, multiple models will be at play in your domain. You need to enforce linguistic boundaries to protect the validity of a domain term. Therefore, linguistic boundaries are bounded context boundaries. If the concept of a product has multiple meanings inside the same model, then the model should be split into at least two bounded contexts, each having a single definition of the product concept. This was discussed previously and illustrated in Figure 6-9. Equally, the same term can refer to multiple concepts, as was the case with the ticket example illustrated in Figure 6-3. That is also an example of a linguistic boundary that should be the boundary of a bounded context.

Align to Business Capabilities

An organization is an ecosystem of interdependent services, each with its own vocabulary. Hence business capabilities are often strong indicators of linguistic boundaries. As mentioned previously, a Sales department and Customer Service department can have completely different definitions of a ticket concept. Accordingly, you should look to business capabilities as potential context boundaries.

> **NOTE** *Part II shows how Service Oriented Architecture (SOA) can be used to create services that are aligned with business capabilities.*

Be careful when using business capabilities to delineate bounded contexts. Sometimes business capabilities do not align perfectly with the problem domain. You can end up with a system that mirrors an organization's communication structure, but does not faithfully represent the domain. Conway's Law even implies that a system will inevitably reflect an organization's communication structure:

> *"Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure."*

You can use Conway's Law as a guide in two ways. Firstly, you can be cognizant of Conway's Law so that you don't just model the organization's structure. Alternatively, you can remodel your organization based on the desired architecture of your system. Either approach is going to require a big effort, so it's not a decision you should take without careful planning.

> **NOTE** *What actually is a business capability? A business capability is a grouping of people in an organization that collaborate on business processes made up of lower-level capabilities. Consider a fulfilment business capability; it may be compromised of a manger(s) who manages staff working in a warehouse. The warehouse staff will carry out low-level processes like packing and dispatching goods, thereby contributing to the business processes of fulfilling an order.*

Create Contexts around Teams

A single team should be responsible for a bounded context, whether that crosses one or many applications or departments. So structure teams around bounded contexts; form product and services groups rather than trying to mirror the departmental structure of the business. Ensure that teams are responsible for a bounded context from presentation through domain logic and to persistence.

AMAZON'S PIZZA TEAMS

Amazon has a policy of ensuring no development team is so big that it cannot be fed by two pizzas (`http://highscalability.com/amazon-architecture`). It is important to keep teams small and focused and make them responsible for a bounded context or a set of bounded contexts. As highlighted by the context map, not all bounded contexts work in isolation. Furthermore, just as there are patterns to communicate in code between bounded contexts, so too are there patterns for team collaboration.

The main rationale for aligning teams with bounded contexts is that independence allows teams to both move faster and make better decisions. Teams can move faster if they are in full control of product and technical decisions. They can also iterate much more rapidly if they don't have to worry about affecting other teams.

A single team can stay focused on its business priorities; therefore, when a decision needs to be made or someone has a suggestion, everyone can quickly huddle together and decide on the best way forward. Conversely, different teams might have different business priorities and arrangements that affect their ability to work together efficiently. For example, one team might have to wait until the other team becomes available before they can schedule a meeting and start making decisions or iterating on concepts.

Remember, communication between teams is sometimes a good thing, so don't completely avoid it; just limit it to when it's useful. One example of useful cross-team communication is knowledge and skill sharing.

Try to Retain Some Communication between Teams

Although having completely independent teams is a productivity win, it's important to ensure that communication between teams still occurs for knowledge and skill-sharing benefits. Ultimately, bounded contexts combine at run time to carry out full use cases, so teams need a big-picture understanding of how their bounded context(s) fit into the wider system. Established patterns for this problem involve having regular sessions in which teams share with other development teams what they are working on, how they have implemented it, or any technologies that have helped them achieve their goals. Another excellent pattern is cross-team pair programming. This involves moving a developer to a different team for a few days to learn about that part of the domain. You can spawn many novel approaches based on these two concepts of having group sessions and moving people around.

Making an effort to ensure that teams communicate efficiently really pays off when breaking changes need to occur. And in every system, you do always get them. At some point, the contract between bounded contexts needs to change to meet the needs of the business. Having teams communicate to work out the best overall solution can sometimes be the most efficient option.

> **NOTE** *Backwards compatibility is also an effective approach that avoids the need for breaking changes (and thus cross-team coordination). You learn about backwards-compatible versioning in Part II, including messaging and REST-based examples.*

Diagrams and lightweight documentation help teams share knowledge quickly, especially when new members join. You'll see how context maps and other types of diagrams facilitate knowledge sharing in Chapter 7, "Context Mapping."

Context Game

To demonstrate the importance of modeling in context and to reveal multiple models within the domain, you can employ another facilitating game. The Context Game, pioneered by Greg Young (`http://codebetter.com/gregyoung/2012/02/29/the-context-game-2/`), helps to make it clear where an additional model is required to map the problem space effectively.

You can introduce the game into knowledge-crunching sessions when you think you have an overloaded or ambiguous term. Split the group into smaller groups of developers and business experts. You should split the business experts by department or business responsibility. Give them 20 minutes to come up with a definition of what the term or concept means to them in their part of the business, using the developers to capture the knowledge. Then bring the whole team together to present their views on the concept.

You will find that different parts of the business have different views on the shared terminology. Where the business functions have a difference of opinion is where you need to draw your context lines and create a new model. This was shown in Figure 6-8 with the product concept existing in many different contexts.

The Difference between a Subdomain and a Bounded Context

Subdomains, introduced in Chapter 3, "Focusing on the Core Domain," represent the logical areas of a problem domain, typically reflecting the business capabilities of the business organizational structure. They are used to distinguish the areas of importance in an application, the core domain, from the less important areas, the supporting and generic domains. Subdomains exist to distill the problem space and break down complexity.

Domain models are built to fulfill the uses cases of each of the subdomains. Ideally there would be a one-to-one mapping between models and subdomains, but this is not always the case. Models are defined based on team structure, ambiguity in language, business process alignment, or physical deployment. Therefore a subdomain could contain more than a single model and a model could span more than a single subdomain. This is often the case within legacy environments.

Models need to be isolated and defined within an explicit context in order to stay pure and focused. As you've learned, this context is known as the bounded context. Unlike a subdomain, a bounded context is a concrete technical implementation that enforces boundaries between models within an application. Bounded contexts exist in the solution space and are represented as explicit domain models in a context.

IMPLEMENTING BOUNDED CONTEXTS

A bounded context owns the vertical slice of functionality from the presentation layer, through the domain logic layer, on to the persistence, and even to the data storage.

Applying the concept of bounded contexts to the system shown previously in Figure 6-5 results in a system with each bounded context looking after its own presentation, domain logic, and persistence responsibilities, as shown in Figure 6-10. In this improved architecture, the concept of a product can exist in each bounded context and only contain attributes and logic prevalent to that context alone. Changes in any bounded context no longer have undesired effects on others because the subdomains are now isolated.

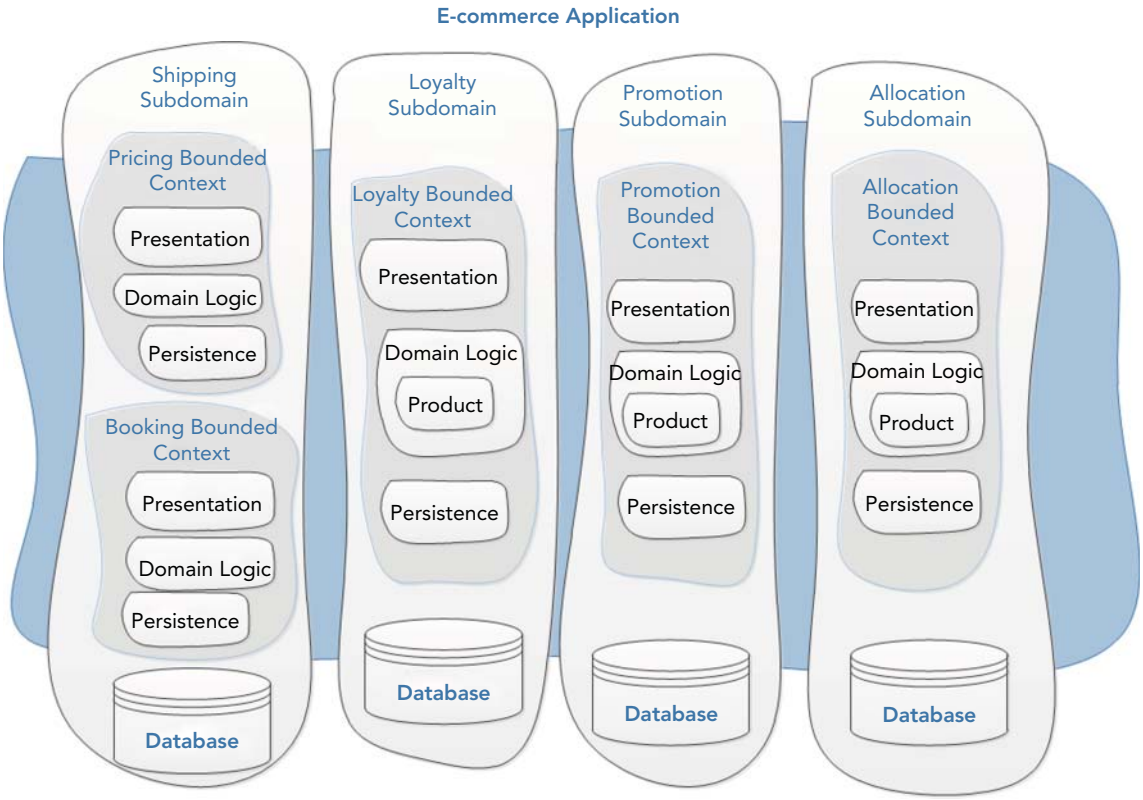


FIGURE 6-10: A layered architecture pattern per bounded context and not per application.

A closer inspection, as shown in Figure 6-11, shows the product concept existing in two models but defined by the context that it is within.

Not all bounded contexts need to share the same architectural pattern. If a bounded context contains a supporting or generic domain with a low logic complexity, you might want to favor a more create, read, update, and delete (CRUD) style of development. If, however, the domain logic is sufficiently complex, it's best to create a rich object-oriented model of the domain. Once bounded contexts are separated you can go a step further and apply different architectural patterns, as shown in Figure 6-12.

Figure 6-12 shows how you can use different architectural patterns within each bounded context of an application. The various bounded contexts are pulled together using a composite UI to display to the user. Figure 6-13 shows that the bounded context encapsulates the infrastructure, data store and user interface as well as the domain model.

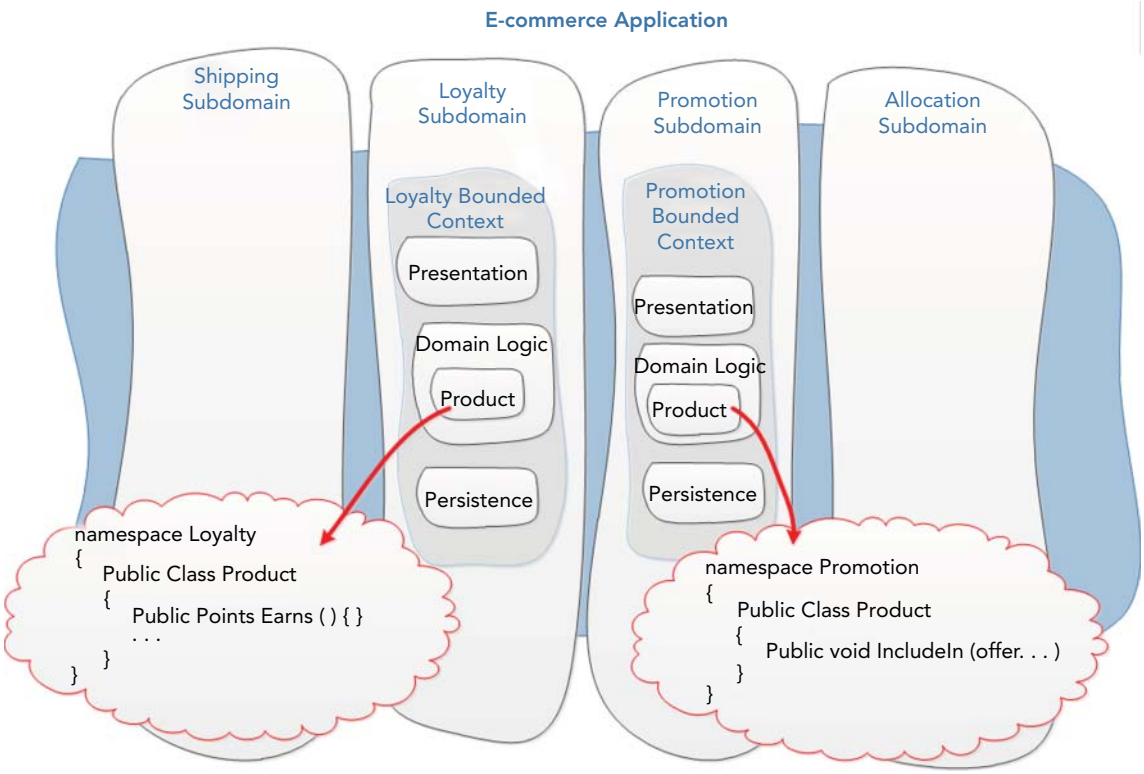


FIGURE 6-11: The Product class in different contexts.

WHAT IS CRUD?

CRUD is an acronym for create, read, update, and delete. It often describes a system with little logic that is merely formed over a data model. To many developers, CRUD is a four-letter word (okay, it really is), and they think a simple solution is beneath them. Don't be frightened of applying a CRUD-style architecture to an application. If you are dealing with a bounded context that contains no logic, don't add lots of layers of abstraction. Remember to KISS (keep it simple, stupid).

WHAT IS CQRS?

CQRS stands for Command Query Responsibility Segregation. It is an architectural pattern that separates the querying from command processing by providing two models instead of one. One model is built to handle and process commands, the other is built for presentation needs. Chapter 24, "CQRS: An Architecture of a Bounded Context," goes into more detail on the pattern.

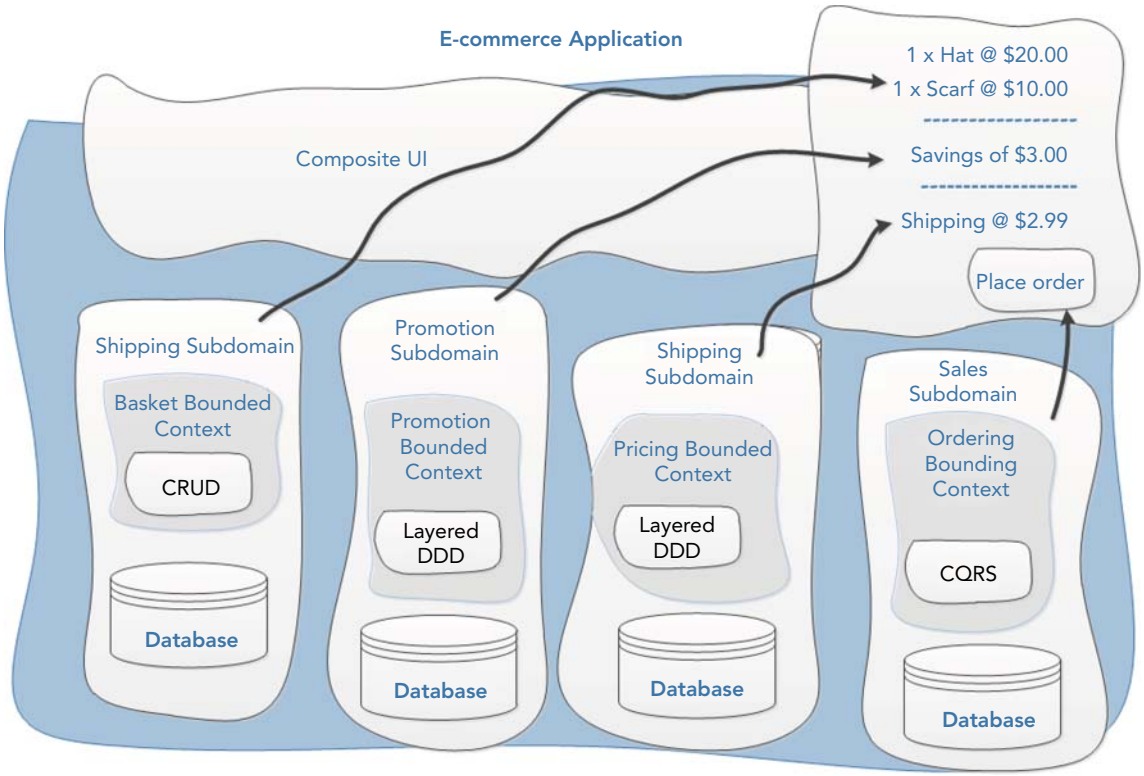


FIGURE 6-12: You can apply different architectural patterns to the different bounded contexts.

WHAT IS A COMPOSITE UI?

A composite UI is a user interface made up of loosely coupled components. A composite UI shows data from multiple bounded contexts on the same web page. This can be done with multiple Ajax calls, for example. In order to protect a model's integrity, a bounded context can employ application services to expose coarse grained methods that encapsulate the details of the underlying domain model, as shown in Figure 6-12. This endpoint can take the input of a different model and transform it into a language that the model inside understands. This protects the integrity of the model and helps to prevent blurred lines of responsibility between models.

NOTE *Application Services are covered in more detail in Chapter 25, "Commands: Application Service Patterns for Processing Business Use Cases."*

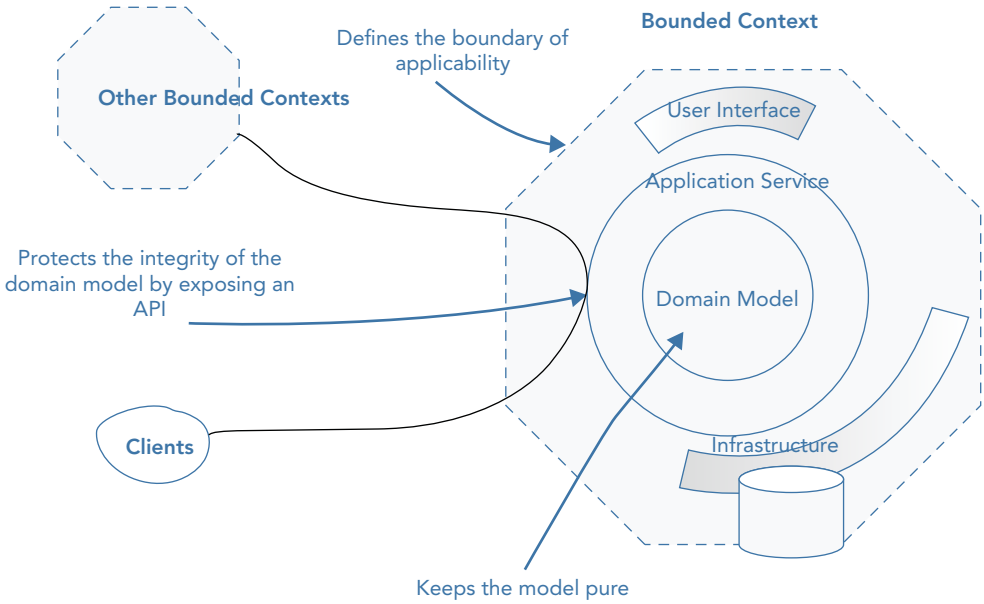


FIGURE 6-13: The anatomy of a bounded context.

Fundamentally, autonomy is a key characteristic of bounded contexts that isolates teams from external distractions, and isolates models from unrelated concepts. In Part II you will see practical examples of implementing and integrating autonomous bounded contexts using scalable integration approaches, including event-driven architecture with messaging and REST.

THE SALIENT POINTS

- Trying to use a single model for a complex problem domain will often cause code to result in a Big Ball of Mud.
- A monolithic model increases collaboration overhead amongst teams and reduces their efficiency at delivering business value.
- For each model at play within an application, you must explicitly define its context and communicate it to other teams.
- A bounded context is a linguistic boundary. It isolates models to remove ambiguity within UL.
- A bounded context protects the integrity of the domain model.
- Identifying and creating bounded contexts is one of the most important aspects of Domain-Driven Design.

- There are no rules for defining the boundaries of a model and therefore bounded contexts. Instead you should base bounded contexts around linguistic boundaries, team organization, subdomains and physical deployments.
- Subdomains are used in the problem space to partition the problem domain. Bounded contexts are used to define the applicability of a model in the solution space.
- A single team should own a bounded context.
- Architectural patterns apply at the bounded context level, not at the application level. If you don't have complex logic in a bounded context, use a simple create, read, update, and delete (CRUD) architecture.
- Speak a ubiquitous language within an explicitly bounded context.
- A bounded context should be autonomous—owning the entire code stack from presentation through domain logic and onto the database and schema.

7

Context Mapping

WHAT'S IN THIS CHAPTER?

- Why the context map is vital for strategic design
- Understanding model relationships between bounded contexts
- The organizational relationship patterns between teams and contexts
- Effective ways to communicate a context map

In large and complex applications, multiple models in context collaborate to fulfill the requirements and behaviors of a system. A single team may not own all of the various sub components of a system, some will be existing legacy code that is the responsibility of a different team, and other components will be provided by third parties that will have no knowledge of the clients that will consume its functionality. Teams that don't have a good understanding of the different contexts within a system, and their relationships to one another, run the risk of compromising the models at play when integrating bounded contexts. Lines between models can become blurred resulting in a Big Ball of Mud if teams don't explicitly map and understand relationships between contexts.

The technical details of contexts within systems are not the only force that can hamper the success of a project. Organizational relationships between the teams that are responsible for contexts can also have a big impact on the outcome of a project. Often, teams that manage other contexts are not motivated by the same forces, or they have different priorities. For projects to succeed, teams usually need to manage changes in these situations at a political rather than technical level.

Other nontechnical challenges can appear during development. These are issues that arise from the areas of the problem domain that sit between bounded contexts that have not been explicitly defined. These important business processes can often be devoid of responsibility from development teams and business ownership, but paradoxically are immensely important to business workflows and processes.

To combat these challenges, teams can create context maps to capture the technical and organizational relationships between various bounded contexts. The greatest strength of a context map is that it is used to capture the reality of the landscape, warts and all, as opposed to an outdated high-level design document. The context map, ever evolving, ensures that teams are informed of the holistic view of the system, both technical and organizational, enabling them to have the best possible chance of overcoming issues early and to avoid accidentally weakening the usefulness of the models by violating their integrity.

A REALITY MAP

A context map, as shown in Figure 7-1, is an important artifact; its responsibility is to ensure that boundaries between various contexts of the system are defined explicitly and that each team understands the contact points between them. A context map is not a highly detailed document created in some kind of enterprise architecture tool, it is a high-level, hand drawn diagram that communicates a holistic picture of the contexts in play. A context map should be simple enough to be understood by domain experts and development teams alike. As well as clearly labelling the contexts the teams understand, the diagram should also show areas of the system that are not well understood to reflect the messy and often unintelligible reality of the codebase.

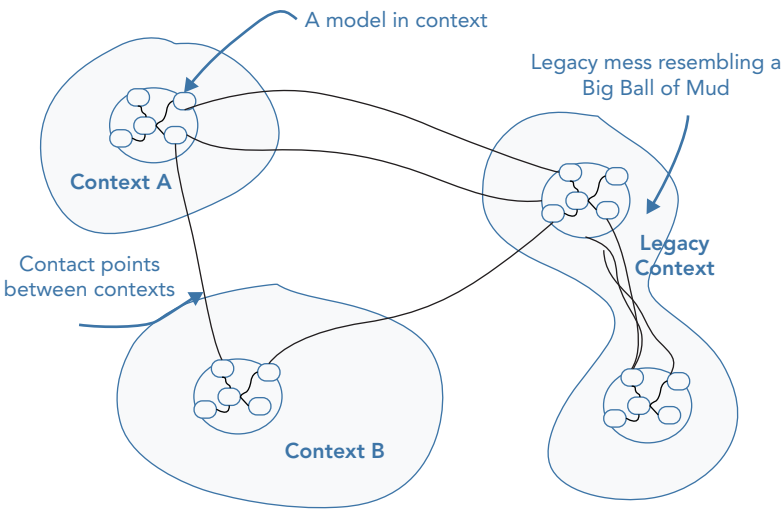


FIGURE 7-1: A context map.

The Technical Reality

The technical details of the map, as shown in Figure 7-2, demonstrate the integration points between contexts. This heat map is vital for teams to understand the technical implications of their changes. It shows the boundaries that exist and any translations that are used to retain the integrity of bounded contexts.

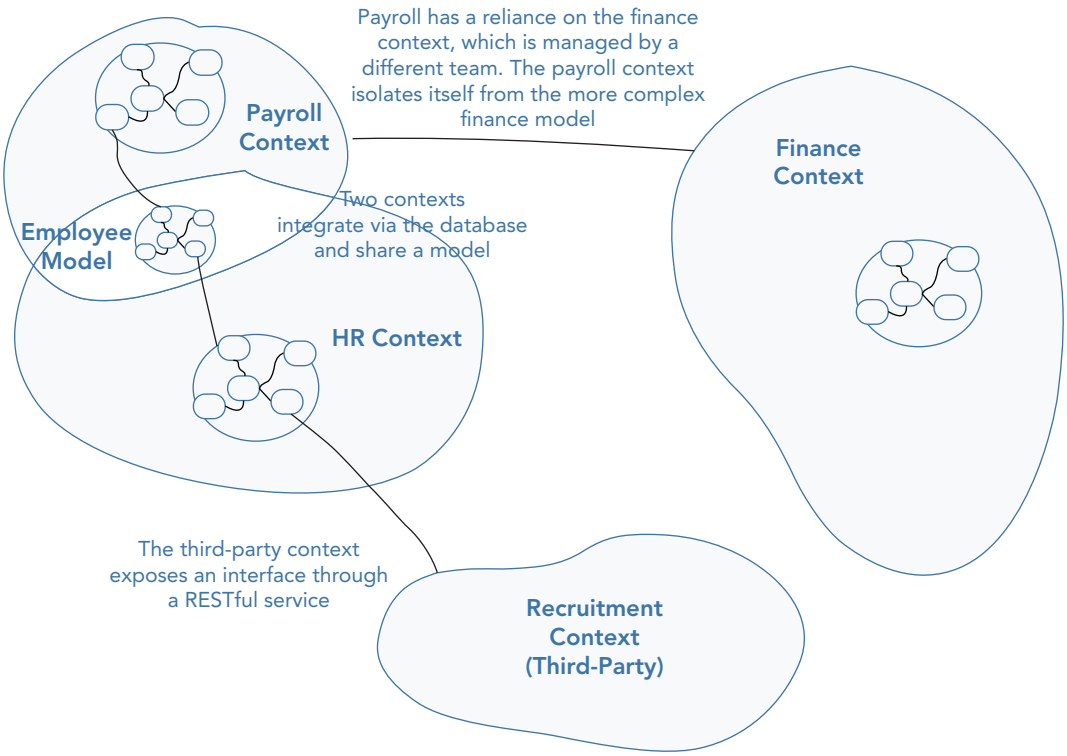


FIGURE 7-2: The technical integration on a context map.

It is extremely important that context maps reflect reality, showing the code in the present state rather than an ideal future state. Context maps need not show the detail of a model; instead, they must demonstrate the integration points and the flow of data between bounded contexts. Like the code model and analysis model, the context map should change only when code changes so it does not give a false impression of the landscape. The map should show the stark reality; only then will it be useful.

The Organizational Reality

Changes to business processes or the creation of new work flows can often span many bounded contexts and reach across various parts of the domain. Coordinating change on this scale often requires as much management of teams as it does technical change. It is vital to understand who is responsible for each context that is required to change and how this change will take place. If the process of coordination and prioritization of changes is not understood, it can be a massive stumbling block and stifle development as teams wait on others to act on requests for change. Showing the direction of team relationships is one of the strengths a context map has over traditional UML or architectural diagrams. Having this knowledge at the start of a project is essential to resolving nontechnical challenges before they block progress.

Figure 7-3 shows the direction of relationships between bounded contexts. Teams that are not on the same project might find release schedules and development priorities need to be aligned if a change is required to a bounded context outside their ownership. Technical change can be fairly straightforward, but if the political situation is not understood, changes to other contexts may be delayed or not implemented at all.

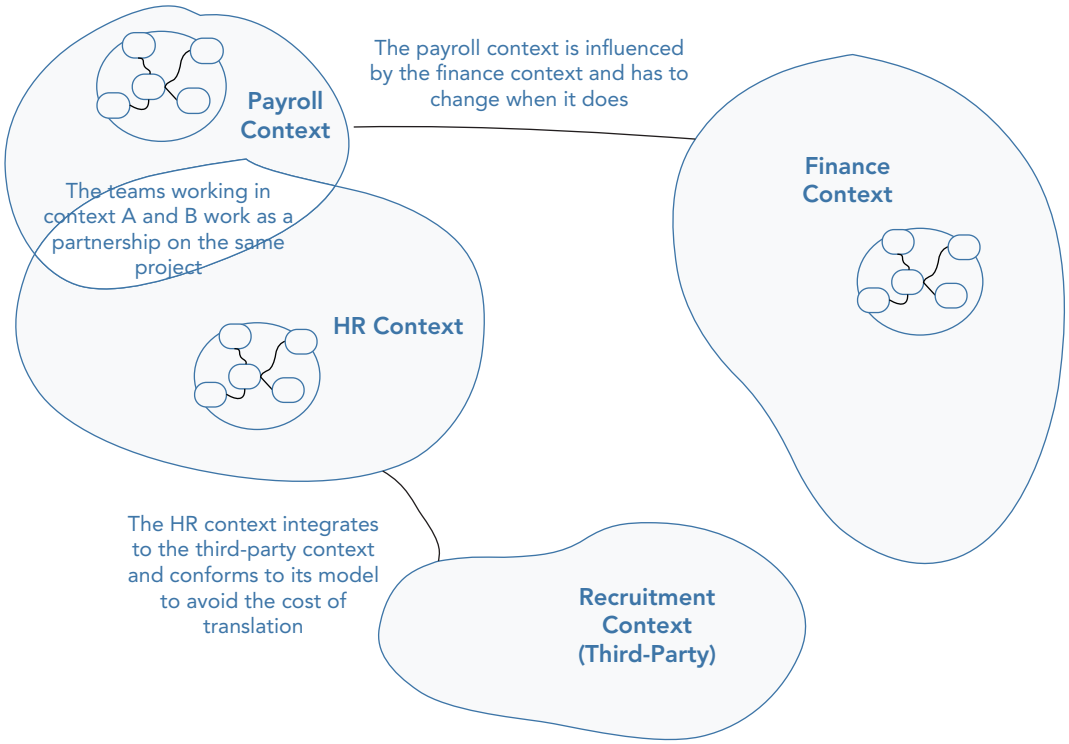


FIGURE 7-3: The organizational relationships on a context map.

Mapping a Relevant Reality

When creating a context map, try to focus on your immediate problem area; you need to understand the landscape that will affect the success of your project and not the entire enterprise. A focus only on the contexts that you will be directly integrating with helps you get going with context mapping and prevents you from losing focus.

X Marks the Spot of the Core Domain

When mapping out the contexts and identifying the models in play, it is a good idea to work with your domain experts and label the core domain. Marking the core domain on the map and discovering the relationships between it and other contexts can provide insight into its clarity in context to the enterprise landscape.

RECOGNISING THE RELATIONSHIPS BETWEEN BOUNDED CONTEXTS

Models in context work together in large applications to provide system behavior. It is important to understand the relationships between the contexts to have a clear understanding as to the lay of the land. The following patterns describe common relationships between bounded contexts. Note that these patterns show how the models relate to each other and how teams relate. They are no technical integration patterns on communicating across contexts. Part II of the book covers the technicalities of how to integrate bounded contexts.

Anticorruption Layer

If you are creating a model for a sub system that communicates with other sub systems as part of a larger system you may need to interface with models created by different teams. Other models, even though created for the same domain, can be expressed with a different ubiquitous language and modelled in a completely different manner to your own. If you are not careful integrating with these models, adapting to their interfaces can lead to a corruption of your model.

In order to avoid corruption and protect your model from external influences you can create an isolation layer that contains an interface written in terms of your model. The interface adapts and translates to the interface of the other context. This isolation layer is known as an anticorruption layer.

As shown in Figure 7-4, you can use an anticorruption layer to wrap communication with legacy or third-party code to protect the integrity of a bounded context. An anticorruption layer manages the transformation of one context's model to another.

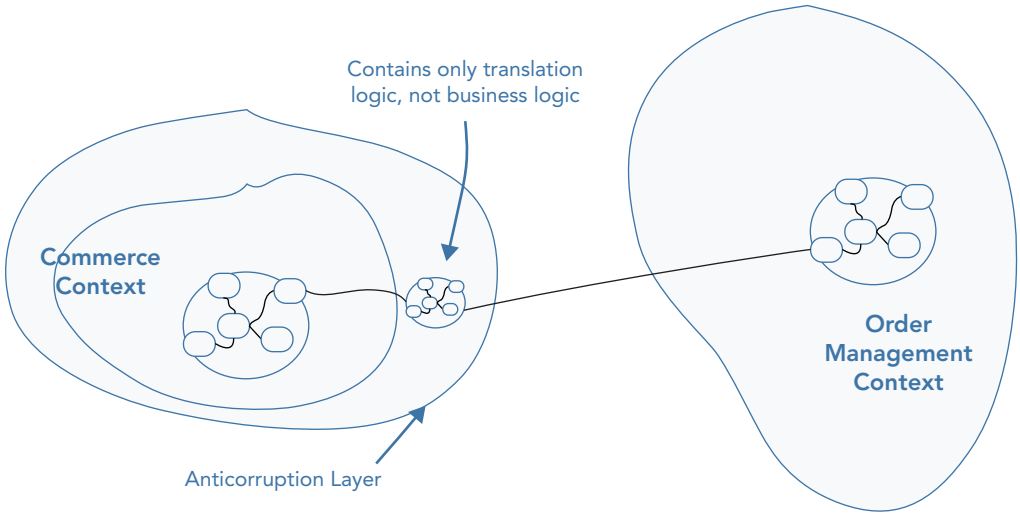


FIGURE 7-4: Use an anticorruption layer to integrate with code you don't own or can't change.

The anticorruption layer's translation map works in a similar manner to the adapter pattern in that it transforms the API of another context into an API that you can work against. Chapter 11 gives an example of the transformation that occurs between bounded contexts using the anticorruption layer.

You won't always be working on greenfield developments, so you will often need to integrate with third-party or legacy contexts. Because you can't change the API of the contexts you don't own or those that can't be changed easily, it's important not to compromise the integrity of your bounded context to match the API of another.

If you have a system that resembles a BBoM and you need to introduce additional functionality it is tempting to simply add code to it and in turn add to the mess; alternatively you can request to rewrite the entire system at the same time as adding the new feature. Neither of these two options is practical as it can be time consuming and risky to rewrite a large application, and simply adding to the mess can increase the maintenance nightmare. A more pragmatic option is to lean on the anticorruption layer, which can be used to isolate the new context from the existing code mess. Using an anticorruption layer in this context is a great refactoring practice because you are able to create clear boundaries without needing to update the mess of code that lives within a context.

Shared Kernel

If two teams are working closely in the same application, on two separate bounded contexts that have a lot of crossover in terms of domain concepts and logic, the overhead of keeping the teams isolated and using translation maps to translate from one context to another can be too much. In this instance, it may be better to collaborate and to share part of the model to ease integration. This shared model is known as a shared kernel. The pattern is of particular use if you have two bounded contexts in the same subdomain that share a subset of domain logic.

Figure 7-5 shows part of an ERP system that contains a payroll context and an HR context that shares the employee model.

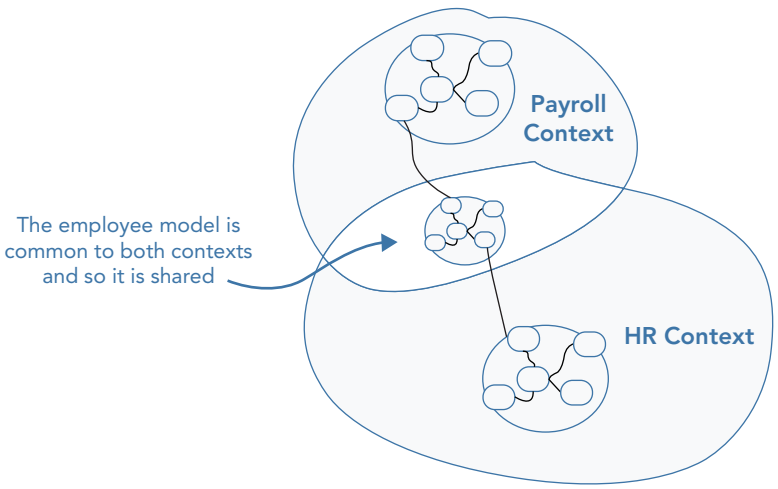


FIGURE 7-5: Integration with a shared kernel.

Because there is a shared code dependency, a shared kernel can be more risky due to the tighter coupling that leads to one team being able to break another team's system. It's important that everyone on both teams understands this and that a continuous integrated test system verifies the behavior of both models when the common model is modified.

Open Host Service

Other systems or components that communicate with you will employ some type of transformation layer in order to translate your model into terms of their own, similar to the anticorruption layer. If multiple consumers share the same transformation logic it can be more useful to provide a set of services that exposes the functionality of a context via a clearly defined, explicit contract known as an open host service.

Consider the example in Figure 7-6. The order management system provides information about customer orders to the commerce system, procurement system, and the CRM system. Each system is required to translate the complicated order management system's order model for use within their own system. To avoid this duplication of efforts the order management system can expose a simplified version of a sales order using a published language via an open host service, as shown in Figure 7-7.

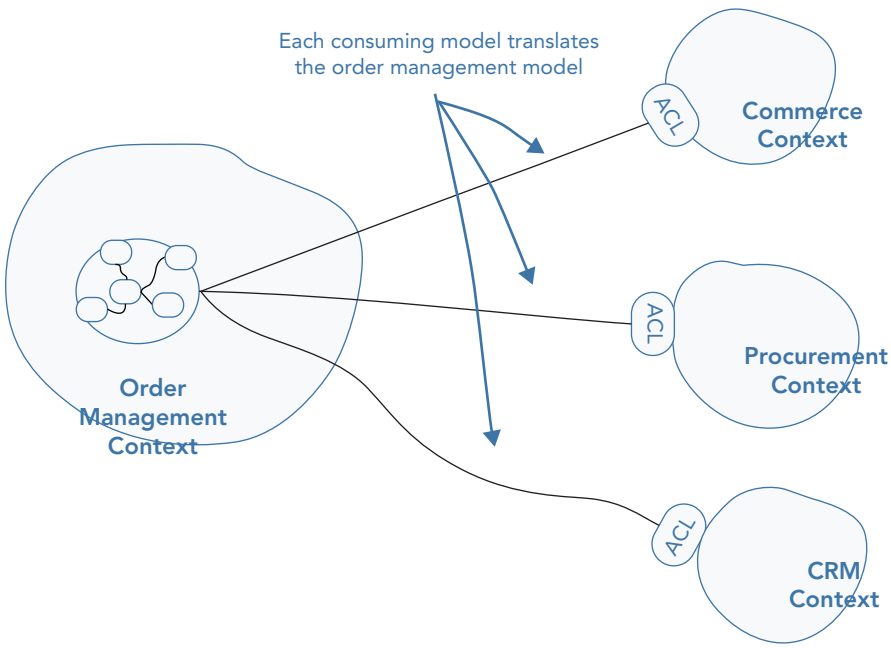


FIGURE 7-6: Multiple subsystems integrating with similar transformation efforts.

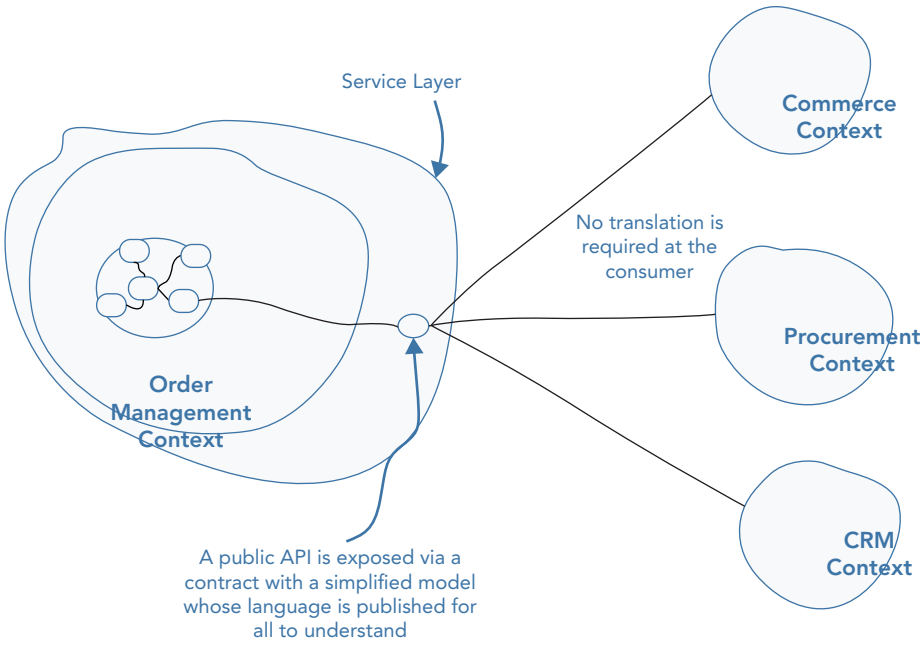


FIGURE 7-7: Integration with an open host service.

Separate Ways

If the cost of integration between contexts is too great due to technical complexities or political ones, a decision can be made to not integrate contexts at all and simply have teams implement separately from one another. Integration can instead be achieved via user interfaces or manual processes. For example, it may be useful for a customer service application that manages contacts with customers to also show users the orders a customer has outstanding when dealing with a query. However, if the effort of integration between an order management system is too great it may be more practical to simply include a menu link that enables the Order Management system to be opened in a separate screen, thus giving users the information they need without the complexities of fully integrating albeit for a small de-scope in feature request.

Partnership

If two teams are responsible for different contexts but are working toward a common goal a partnership can be formed to ensure that cooperation on integration between the two contexts can be made. Cooperation can cover the technical interfaces so that they accommodate both teams' interests. From a political standpoint releases can be aligned between the teams so that necessary interfaces and contact points are released at a time that they are required. If teams are using a shared kernel between two bounded contexts it is recommended that they do so as a partnership.

An Upstream/Downstream Relationship

The relationships between bounded contexts can be defined in terms of a direction; one end will be upstream and the other downstream. If you are the downstream end of the relationship you are

dependent on data or behavior of the upstream end. The upstream end will influence the downstream context. For instance, if an upstream interface changes so must the consuming side downstream. Likewise the release plan of the upstream part of the relationship will influence the downstream context as it may be dependent on a particular API method. The following patterns show how the upstream and downstream relationships can be classified.

Customer-Supplier

In situations where teams are not working toward a common goal to avoid the upstream team making all the decisions and potentially compromising the downstream team to the detriment of the project as a whole, a more collaborative customer-supplier relationship can be formed. In this pattern, the teams work together to create an agreed-upon interface that satisfies both from a technical and scheduling standpoint. The customer part of the relationship is the downstream context. The customer will join the supplier's (upstream context) planning meeting to ensure its needs are understood and that it can have visibility when upstream changes are occurring.

Due to the increased collaboration of the customer/supplier relationship, decisions can take longer. Teams need to have meetings or online discussions to progress. With careful planning, it might be possible to make agreements in advance so that no team is blocked waiting for the other to make a decision or deploy its new system with the updated interface. On the other hand, for remote teams, teams in different time zones, or teams with busy schedules, the collaboration overhead could cause lengthy delays.

The customer-supplier relationship emphasizes that the customer team's bounded context relies on the supplier team's bounded context, but not vice versa. Sometimes there is no opportunity to form a collaborative relationship with an upstream context and so the downstream context must conform to the upstream context's integration points.

For example, consider Figure 7-8; the commerce context requires more information on a sales order than is currently supplied from the order management context. The team that is responsible for the commerce context can act as a customer during the order management context team's planning sessions to ensure their needs are understood and accounted for.

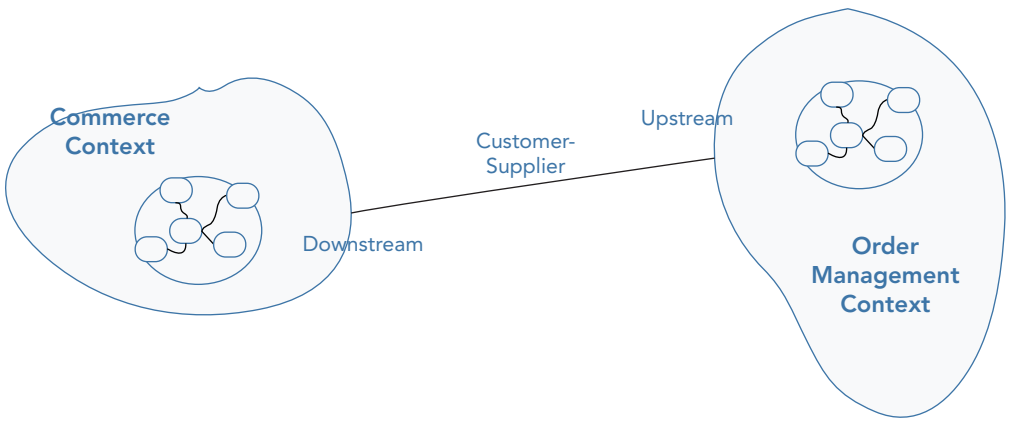


FIGURE 7-8: A customer-supplier relationship between bounded contexts.

Conformist

If an upstream context is not able to collaborate then the downstream context will need to conform to the upstream context when integrating. The most common occurrence of the conformist relationship is integrating with external suppliers. It's almost certain that a payment provider will not change its API for you and give you extra information unless you are an influential client, which would make you upstream of them. Instead, if you are downstream and are unable to form a customer-supplier relationship and it is too costly to create an anticorruption layer you should conform to the model of the provider to simplify integration. The most obvious downside to the conformist relationship is that the downstream team, which works to the requirements of the upstream team, may have to sacrifice clarity of its domain model because it must align to the model of the upstream context even though it may be conceptually different than your own view. Alternatively, an anticorruption layer can be used to retain the integrity so that changes to a contact point don't affect the underlying model.

COMMUNICATING THE CONTEXT MAP

When drawing up your context maps, you can add the type of organizational relationship that exists as well as the type of technical integration between two bounded contexts on the line that joins them. You can also indicate which side of the line is upstream and which is downstream using letters or symbols, if applicable. Figure 7-9 shows an example of a context map with these features.

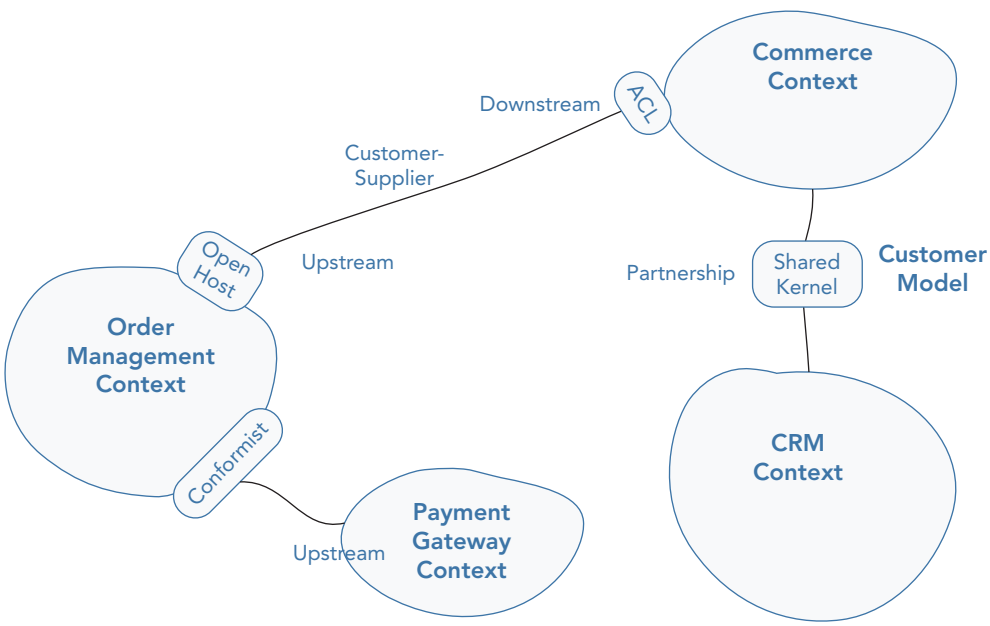


FIGURE 7-9: A context map showing the types of integration between bounded contexts.

THE STRATEGIC IMPORTANCE OF CONTEXT MAPS

In many ways, the communication between bounded contexts, both technical and organizational, is more important for teams starting out on a project than the bounded contexts themselves. Information that context maps provide can enable teams to make important strategic decisions that improve the success of a project. A context map is a powerful artifact that can bring new team members up to speed quickly and provide an early warning for potential trouble hot spots. Context maps can also reveal issues with communication and work flows within the business.

Retaining Integrity

All development teams in the organization need to understand the context map. Teams don't need to understand the inner workings of each bounded context; instead, they need to be aware of those other contexts—the application programming interface (API) they expose, the relationships they have, and, most importantly, the conceptual models they are responsible for. With this information, teams can prevent blurring the lines of responsibility and ensure that all contexts retain their integrity.

Retaining integrity is important to keep your codebase focused on a single model. This enables the code to become supple because any change affects only a single bounded context and doesn't have a rippling effect across multiple areas of your domain. It's this suppleness that enables you to alter code and to adapt quickly and confidently when the business needs a change to process or logic.

The Basis for a Plan of Attack

A context map highlights areas of confusion and mess, and, more importantly, where the core domain is. Teams can use this information to identify areas they need to clear up first to improve the success of a project:

- Areas that are too far gone and where the effort to improve is far greater than the payback can be isolated and left.
- Areas of no strategic advantage or that are of low complexity need not incur the cost of creating a ubiquitous language or follow the model-driven development methodology.
- Areas that are core to the success of the project or are complex are candidates for the tactical side of Domain-Driven Design, and should be kept isolated from poorly designed bounded contexts to retain integrity.

Understanding Ownership and Responsibility

Accountability and responsibility are other nontechnical areas that can affect a project. Defining team ownership and management for subsystems that you need to integrate with is essential for ensuring changes are made on time and in line with what you expect. Context mapping is about investigation and clarification; you may not be able to draw a clear context map straight away, but the process of clarifying responsibility, explicitly defining blurred lines, and understanding communication flow while mapping contexts is as important as the finished artifact.

Revealing Areas of Confusion in Business Work Flow

The business processes that happen between and take advantage of bounded contexts are often left in no-man's-land without clear responsibility and clarity regarding their boundaries and integration methods. A context map, focusing on the nontechnical aspects of the relationships, can reveal broken business process flow and communication between systems and business capabilities that have degraded over time. This revelation is often more useful to the businesses that are able to better understand and improve process that spans across departments and capabilities. The insight can be used to reduce risk of project failure by tackling ambiguity early and asking powerful questions that help the success of the project.

The often gray area between contexts that govern business process is also void of accountability when changes are being made, and is only discovered later on in a project's life cycle.

Identifying Nontechnical Obstacles

Context maps reveal the departmental boundaries involved in a project. If your team does not own all the contexts in play, coordination with other teams and other lines of management and prioritization needs to take place. Understanding these obstacles up front gives you a much greater probability of success on a project and enables you to tackle nontechnical problems such as release scheduling before they become blockers.

In a similar manner, changes that require integration with third-party contexts can expose requirements on testing environments and coordination with outside teams or at least access to sandbox accounts and documentation.

Encourages Good Communication

When a diagram shows you that a relationship exists between your bounded context and another bounded context, you should have a pretty good idea that you need to be communicating with the team responsible for it. When the diagram also indicates that you are the upstream team, you understand that your responsibility is usually to lead decision making, and accordingly, you may often need to initiate communication.

Helps On-Board New Starters

Have you ever started working for a new company and not understood how your system fits into the system as a whole? Have you ever felt uneasy about answering questions from domain experts because they came to you with problems that also touched on parts of the system you barely knew existed? Having a concise, yet informative, context map that the whole team regularly views and keeps up to date is a fantastic way to ensure all team members understand the bigger picture—or at least have an idea of which parts of the system they don't know enough about. If a domain expert approaches you with a problem that you're unfamiliar with, you can turn to the context map for suggestions about who it would be best to talk to.

THE SALIENT POINTS

- A context map reflects the way things are right now. It provides a holistic view of the technical integration methods and relationships between bounded contexts. Without them, models can be compromised, and bounded contexts can quickly change to balls of mud as integration blurs the lines of a model's applicability.
- An anticorruption layer provides isolation for a model when interfacing with another context. The layer ensures integrity is not compromised by providing translation from one context to another.
- Integration using the shared kernel pattern is for contexts that have an overlap and shared a common model.
- Integration via an open host service exposes an external API instead of requiring clients to transform from one model to another. Typically, this creates a published language for clients to work with.
- Relationships between bounded contexts can be understood in terms of being upstream or downstream of one another. Upstream context have influence over downstream contexts.
- Collaboration between two teams not working to a common goal or not on the same project is known as a customer-supplier relationship. Downstream customers can collaborate with their upstream suppliers to integrate contexts.
- The conformist pattern describes the relationship between an upstream and downstream team where the upstream team will not collaborate with the downstream team. This is often the case when the upstream team is a third-party.
- The partnership relationship pattern describes two teams that have a common goal and a usually on the same project but working on two different contexts.
- Separate ways should be followed if bounded contexts are too costly to integrate and other nontechnical methods can be found.

8

Application Architecture

WHAT'S IN THIS CHAPTER?

- Application architecture patterns that protect the integrity of your domain model
- The difference between application and bounded context architectures
- The role and responsibilities of application services
- How to support various application clients

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 8 download and individually named according to the names throughout the chapter.

Domain-Driven Design (DDD) focuses on managing the challenges of building applications with complex domain logic by isolating the business complexities from the technical concerns. Up until now, this book has only looked at techniques to enable teams to model a useful conceptual abstraction of the problem domain. This chapter, however, looks at patterns that enable the domain model to be utilized in the context of an application, taking into consideration persistence, presentation, and other technical requirements.

APPLICATION ARCHITECTURE

Developing software while following the principles of DDD does not require you to use any particular application architecture style. But one thing that your architecture must support is the isolation of your domain logic.

Separating the Concerns of Your Application

To avoid turning your codebase into a Big Ball of Mud (BBoM) and thus weakening the integrity and ultimately the usefulness of a domain model, it is vital that the structure of an application supports the separation of technical complexities from the complexities of the domain. Presentation, persistence, and domain logic concerns of an application will change at different rates and for different reasons; an architecture that separates these concerns can accommodate change without causing an undesired effect to unrelated areas of the codebase.

Abstraction from the Complexities of the Domain

In addition to a separation of concerns, an application architecture must abstract away from the intricacies of a complex domain by exposing a coarse-grained set of use cases that encapsulate and hide the low-level domain details. Abstracting at a higher level prevents changes in domain logic from affecting the presentation layer and vice versa because the clients of the application communicate through application services acting as use cases rather than directly with domain objects.

A Layered Architecture

To support the separation of concerns, you can layer the different responsibilities of an application, as shown in Figure 8-1. Fowler catalogued the ubiquitous layered architecture in his book *Patterns of Enterprise Application Architecture*. However, many other architectures support the separation of concerns by dividing an application into areas that change together, such as Uncle Bob's Clean Architecture, the Hexagonal Architecture (also known as the Ports and Adapters Architecture), and the Onion Architecture.

Unlike typical views of a layered architecture, Figure 8-1 shows that at the heart of the architecture is the domain layer containing all the logic pertaining to the business. Surrounding the domain layer is an application layer that abstracts the low-level details of the domain behind a coarse-grained application programming interface (API) representing the business use cases of the application. The domain logic and application layers are isolated and protected from the accidental complexities of any clients, frameworks, and infrastructural concerns.

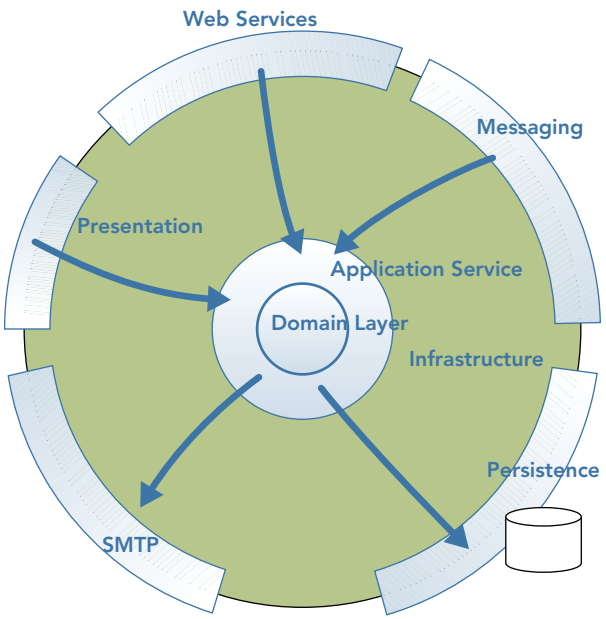


FIGURE 8-1: A layered architecture.

Dependency Inversion

To enforce a separation of concerns, the domain layer and application layers at the center of the architecture should not depend on any other layers. All dependencies face inward, in that the domain layer, at the heart of the application, is dependent on nothing else, enabling it to focus distraction-free on domain concerns. The application layer is dependent only on the domain layer; it orchestrates the handling of the use cases by delegating to the domain layer.

Of course, the state of domain objects needs to be saved to some kind of persistence store. To achieve this without coupling the domain layer to technical code, the application layer defines an interface that enables domain objects to be hydrated and persisted. This interface is written from the perspective of the application layer and in a language and style it understands free from specific frameworks or technical jargon. The infrastructural layers then implement and adapt to these interfaces, thus giving the dependency that the lower layers need without coupling. Transaction management along with cross-cutting concerns such as security and logging are provided in the same manner. Figure 8-2 shows the direction of dependencies and the direction of interfaces that describe the relationship between the application layer and technical layers.

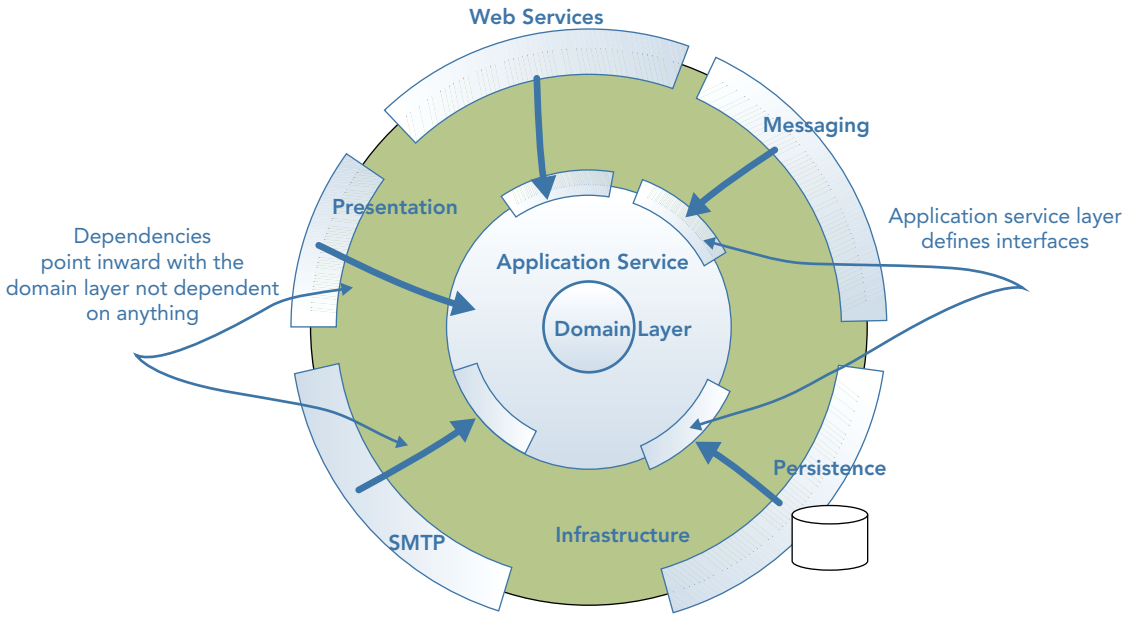


FIGURE 8-2: Dependency inversion within a layered architecture.

The Domain Layer

As discussed in Chapter 4, "Model-Driven Design," a domain model represents a conceptual abstract view of the problem domain created to fulfill the needs of the business use cases. The domain layer containing the abstract model does not depend on anything else and is agnostic to the technicalities of the clients it serves and data stores that persist the domain objects.

The Application Service Layer

The application service layer represents the use cases and behavior of the application. Use cases are implemented as application services that contain application logic to coordinate the fulfillment of a use case by delegating to the domain and infrastructural layers. Application services operate at a higher level of abstraction than the domain objects, exposing a coarse-grained set of services while hiding the details of the domain layer—what the system does, but not how it does it. By hiding the complexities of the domain behind a façade, you can evolve the domain model while ensuring that changes do not affect clients.

The client of the domain layer is the application service layer; however, to perform its work, it requires dependencies on external layers. These dependencies are inverted because the application layer exposes the contracts to the interfaces it requires. The external resources must then adapt to the interfaces to ensure the application layer is not tightly coupled to a specific technology.

Coordinating the retrieval of domain objects from a data store, delegating work to them, and then saving the updated state is the responsibility of the application service layer. Application service layers are also responsible for coordinating notifications to other systems when significant events occur within the domain. All these interfaces with external resources are defined within the application service layer but are implemented in the infrastructural layer.

The application service layer enables the support of disparate clients without compromising the domain layer's integrity. New clients must adapt to the input defined by the application's contract—its API. They must also transform the output of the application service into a format that is suitable for them. In this way, the application layer can be thought of as an anticorruption layer, ensuring that the domain layer stays pure and unaffected by external technical details.

The Infrastructural Layers

The infrastructural layers of an application are the technical details that enable it to function. Whereas the application and domain layers are focused on modeling behavior and business logic, respectively, the infrastructural layers are concerned with purely technical capabilities, such as enabling the application to be consumed, whether by humans via a user interface or by applications via a set of web service or message endpoints. The infrastructural layers are also responsible for the technical implementation of storing information on the state of domain objects.

In addition, the infrastructural layer can provide capabilities for logging, security, notification, and integration with other bounded contexts and applications. These are all external details—technical concerns that should not directly affect the use case exposed and the domain logic of an application.

Communication Across Layers

When communicating across layers, to prevent exposing the details of the domain model to the outside world, you don't pass domain objects across boundaries. For the same reasons, you don't send raw unsolicited data or user input straight into the domain layer. Instead, you use simple data transfer objects (DTOs), presentation models, and application event objects to communicate changes or actions in the domain.

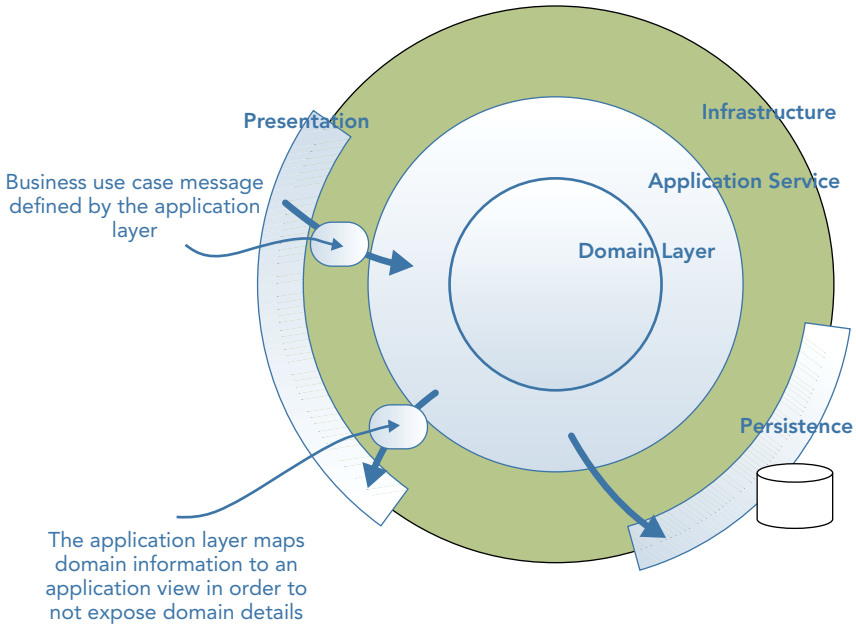


FIGURE 8-3: Domain objects are hidden from clients of the application.

To avoid tight coupling of layers, higher layers must communicate with lower layers by adapting to their message types. This again keeps the lower layers isolated and loosely coupled to any external layers. Figure 8-3 shows the communication across the layers and how data is transformed to protect the integrity of the domain model.

Testing in Isolation

Separating the different concerns in your application and ensuring your domain logic is not dependent on any technically focused code such as presentation or data persistence frameworks enables you to test domain and application logic in isolation, independent of any infrastructural frameworks.

As shown in Figure 8-4, you can use unit tests to confirm the logic within the domain layer. You can use mocks and stubs to give the application layer the fake implementations it requires to confirm the correctness of business task coordination with the domain layer and external resources.

Don't Share Data Schema between Bounded Contexts

In addition to separating the concerns within the codebase of an application, an architecture must include the separation of the persistence of the domain object state from other applications' data requirements. Figure 8-5 shows applications integrated via a shared database and shared schema.

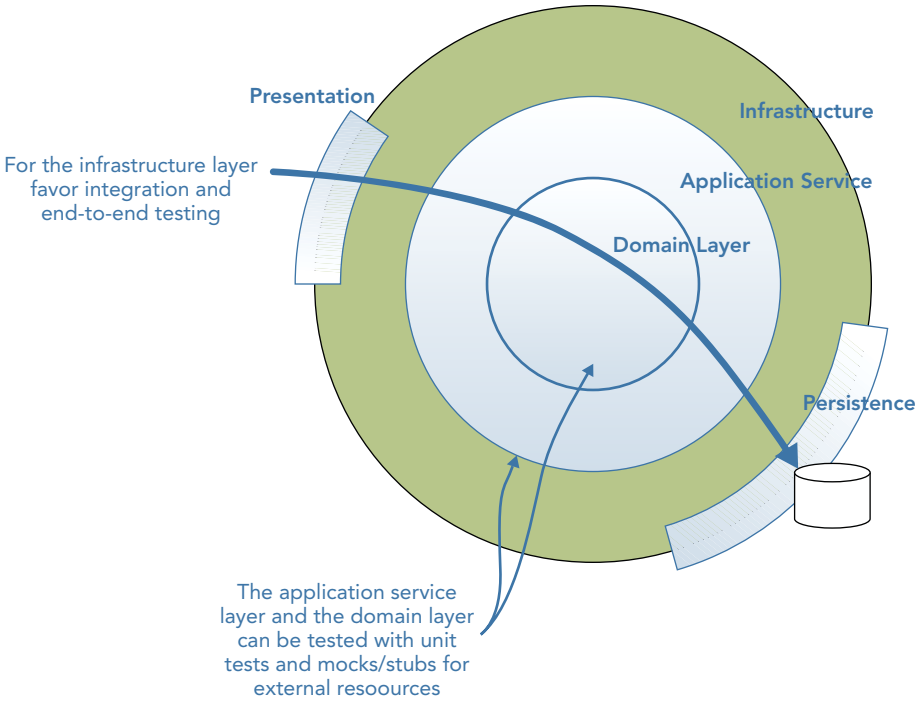


FIGURE 8-4: Testing layers in isolation.

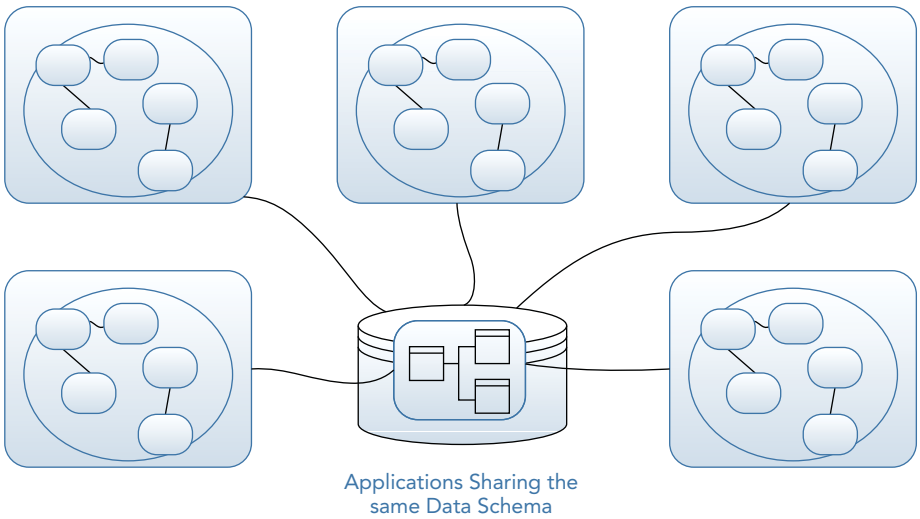


FIGURE 8-5: Bounded contexts integrating via a shared data schema.

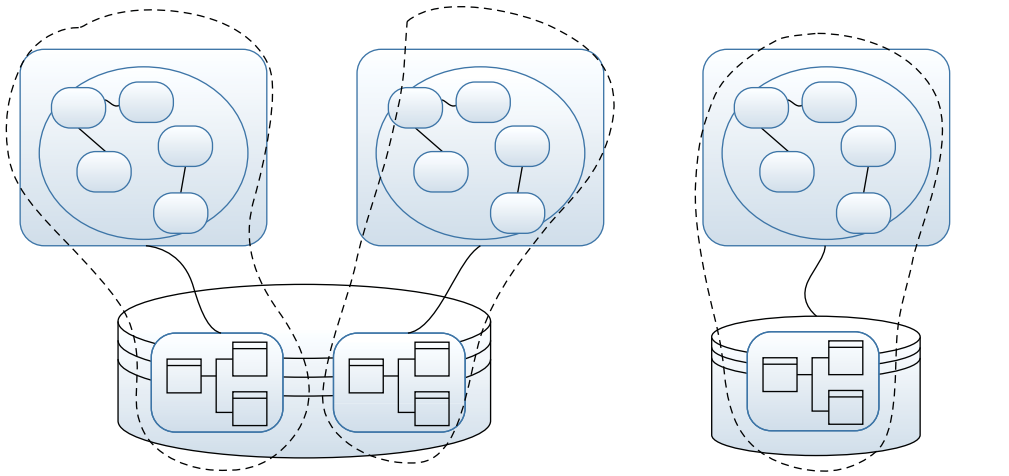


FIGURE 8-6: Bounded contexts with their own data schema.

Although this is an easy integration method, it can complicate and blur the lines of a model by acting as the catalyst to your codebase growing into a BBoM. Sharing data makes it easy for client code to bypass the protection of a bounded context and interact with a domain object state without the protection of domain logic. It is also easy to interpret logic and schema incorrectly, resulting in changes to the state that invalidate invariants.

As shown in Figure 8-6, you should favor application or bounded context databases over integration databases. Just as you apply context boundaries within the domain model, you must do the same for the persistence model. This helps to force clients to integrate through the well-defined application service layer, protecting the integrity of your model and ensuring invariants are met.

Application Architectures versus Architectures for Bounded Contexts

Applications can be composed of more than one bounded context. Architectures apply to bounded contexts and applications in different ways. An application that is composed of two or more bounded contexts may have an architectural style for the user interfaces and different architectures for each of the bounded contexts. Figure 8-7 shows an application composed of three bounded contexts; here the presentation layer contains its own application layer to facilitate the coordination with the bounded contexts.

However, some people believe that the boundary of a bounded context should extend to the presentation layer. Udi Dahan's business component gives the bounded context the responsibility for owning specific regions of the user interface. This architecture can be seen in Figure 8-8.

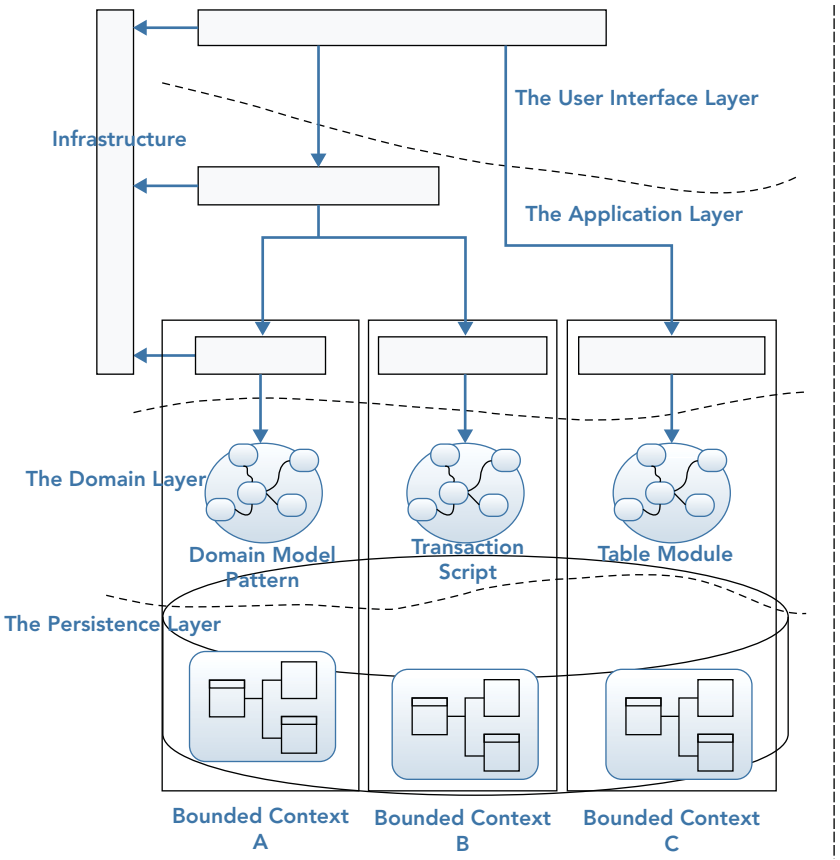


FIGURE 8-7: Bounded contexts integrating via a separate application layer.

In this architecture style, the infrastructure takes care of ensuring communication and the sharing of correlation IDs.

There does not need to be consistency in architectural styles or data stores across bounded contexts, but within a single bounded context, you should strive to follow one method of representing domain logic.

APPLICATION SERVICES

The application service layer, cataloged as the service layer in Fowler's *Patterns of Enterprise Application Architecture* book, can be used to define the boundary of your domain model and can also be thought of as the implementation of the bounded context concept, isolating and protecting the integrity of your domain model.

As mentioned earlier in this chapter the responsibility of the application service layer is to expose the capabilities and operations available to the application while abstracting the low-level complexities

of the domain model. Capabilities are defined by the business use cases that the system must satisfy. The application services fulfill the use cases by coordinating the execution of domain logic. They deal with technical concerns such as handling input and shaping reporting information on the state of the domain, as well as transactional, logging, and persistence concerns.

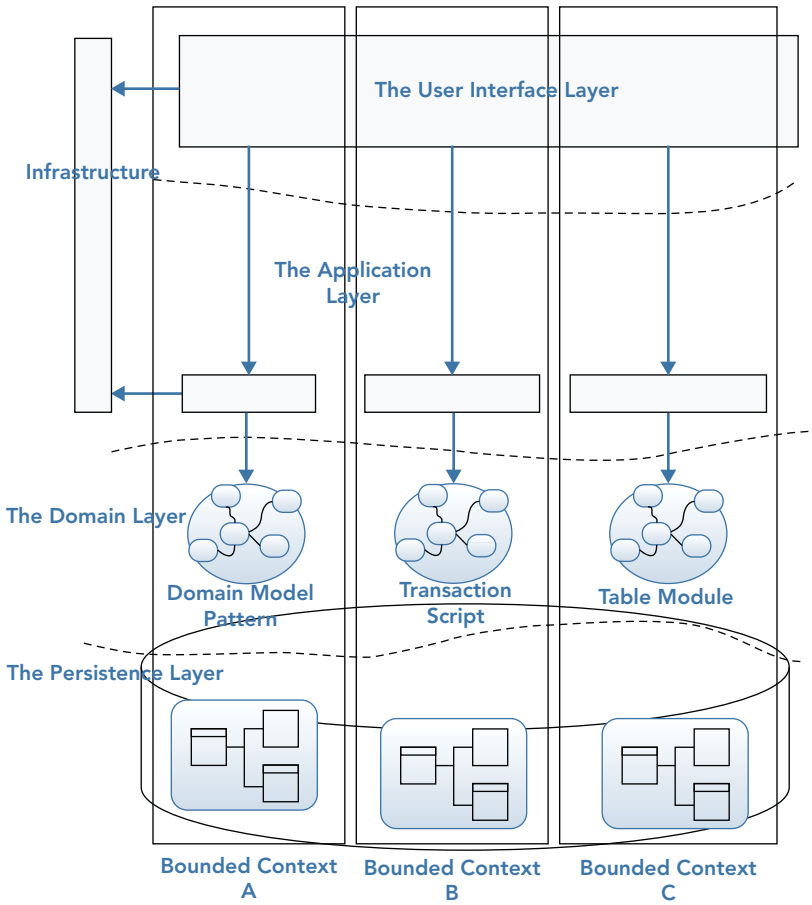


FIGURE 8-8: Presentation layer composed of bounded contexts.

Application services contain application logic only. This logic covers security, transaction management, and communication with other technical capabilities such as e-mail and web services. They are the clients of the domain layer and delegate all work to that layer. No domain logic should be found within the application services; instead, the application services should be procedural in style and thin. The application layer is not dependent on any frameworks or technology that consumes the application service, such as UI or service frameworks. It does, however, define interfaces that it depends on to hydrate domain objects and manage nondomain tasks.

Application Logic versus Domain Logic

Application logic contains the workflow steps required to fulfill a business use case. Steps can include the hydrating of domain objects from a database, the mapping of user input to objects that the domain layer understands, and ultimately the delegating to domain objects or a collection of them to make a business decision. Other steps may include delegating to infrastructural services, such as notifying other systems of changes in domain state via messaging systems or web calls, authorization, and logging.

Application logic is all about coordination and orchestration through delegation to domain and infrastructural services. The application services don't do any work, but they understand who to talk to to complete the task. Domain logic, on the other hand, is focused only on domain rules, concepts, information, and work flows. Domain logic is free from technical details, including persistence.

As an example, consider Figure 8-9, which models the use case of applying a promotion coupon to an e-commerce basket. The ASP.NET MVC framework presentation layer transforms the Hypertext Transport Protocol (HTTP) request into a form that the application service layer expects and calls the service method. The application service delegates to the persistence layer to retrieve the coupon object. It then checks whether the coupon is still valid. If it is not, it responds with an appropriate result. If it is valid, it again delegates to the persistence layer to retrieve the basket and passes the basket to the coupon to generate a discount. The changes to the discount on the basket domain object are persisted, and an event is published to notify that the coupon was redeemed.

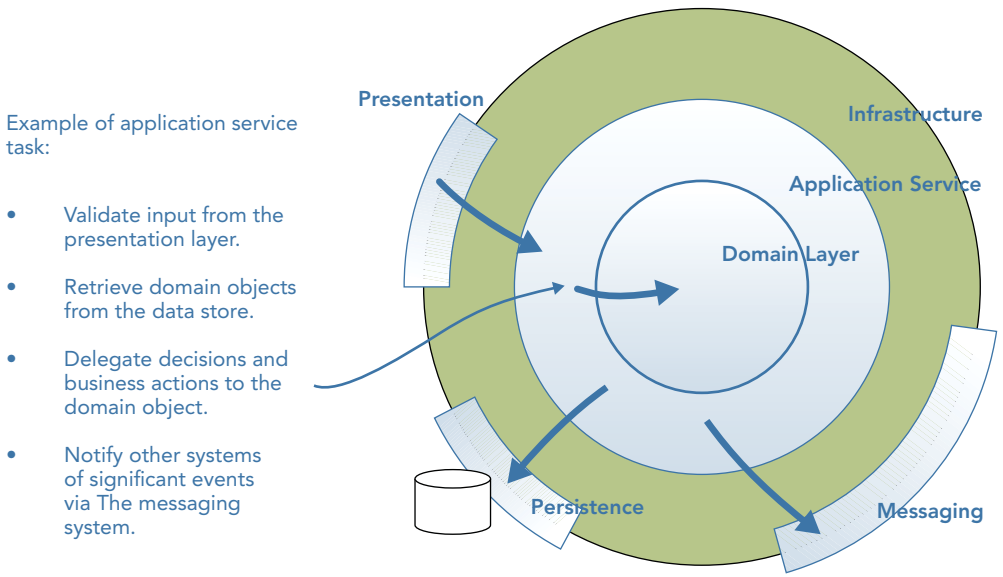


FIGURE 8-9: Application logic versus domain logic.

Defining and Exposing Capabilities

Because the application services are exposing capabilities of the system, they should not have to adapt to new clients. Instead, new clients, such as presentation layers, should adapt to the contracts exposed by the services. In other words, the capabilities of the system should not change for clients. Rather,

they should change only when the business use case changes. The use cases exposed by the application services change at a different rate and for different reasons than the domain logic that is used to fulfill them. This enables clients consuming the services to be protected from frequent changes to domain logic.

Take, for example, the business use case of risk assessing an order for fraud. The system exposes the capability to take details of an order and return a score based on domain logic. Over time, the domain logic may change, but the application service that is the implementation of the use case to score an order for risk will largely remain constant, changing only to alter its contract to provide additional information.

The stakeholders may not know the complexities of the domain layer; however, the business tasks that the application layer is responsible for are meaningful to the business. Even if they are not domain experts, the stakeholders should understand them.

Business Use Case Coordination

Whereas the domain model is object-oriented in its nature, the application services are procedural, as they are focused on task orchestration as opposed to modeling domain logic and concepts. Application services are somewhat similar to ASP.NET MVC controller actions. Controller actions contain logic to control the user interface interactions in the same manner that application services contain logic that represents business tasks or use cases that coordinate communication with services and objects within the domain layer. Both controller actions and application services are stateless and procedural in nature. An exception is that both controller actions and application services can store state, but the state should only be to store the status of the customer journey or the progress of the business task.

Application services also share more coordination logic with controller actions in the form of transforming and mapping input and output. Controller actions map HTTP post variables to objects that application services require and map application services query responses to view models for presentation needs. Application services, in the same way, map requests into structures that domain objects understand, and respond with presentation models that hide the real form of the domain objects and that are specific to user interface views.

Application Services Represent Use Cases, Not Create, Read, Update, and Delete

Behavior-Driven Design (BDD) helps you understand the behaviors of an application. With the behaviors you capture using BDD, you can use the language expressed in the BDD specifications as the name for you application services use/cases. This is similar to the way you use the ubiquitous language (UL) of the domain within the code of the domain layer. Application services are not simply create, read, update, and delete (CRUD) methods; they should reveal user intent and communicate the capabilities of a system. Examples of this can be seen in Chapter 25, “Commands: Application Service Patterns for Processing Business Use Cases,” along with patterns on how to implement the application service layer.

Domain Layer As an Implementation Detail

Application services are powerful and can be helpful for any application complexity, be it a core subdomain with rich logic or a generic subdomain that is merely a façade for access to the data

store. Having the application services decouple clients from the logic enables the domain layer to evolve cleanly without having a ripple effect across layers.

Your application service methods can reveal whether a domain model is required at all. If you find that all your business use cases are simply updating, adding, or deleting data, then it's a good bet that the domain is lacking any real logic and can be kept simple by employing a transaction script or data wrapper pattern, as discussed in Chapter 2, "Distilling the Problem Domain," instead of a full-blown rich domain model. However, if the application services and behaviors of your system are rich in language, this may suggest the need for a domain model pattern in your domain logic layer.

Domain Reporting

Besides coordinating business tasks, the application service layer needs to provide information on the state of domain objects in the form of reports. You don't want to expose the inner workings of your domain model to the outside world, so the application services transform domain objects into presentation models that give specific views of domain state without revealing the structure of the domain model. You can see this transformation in Figure 8-10.

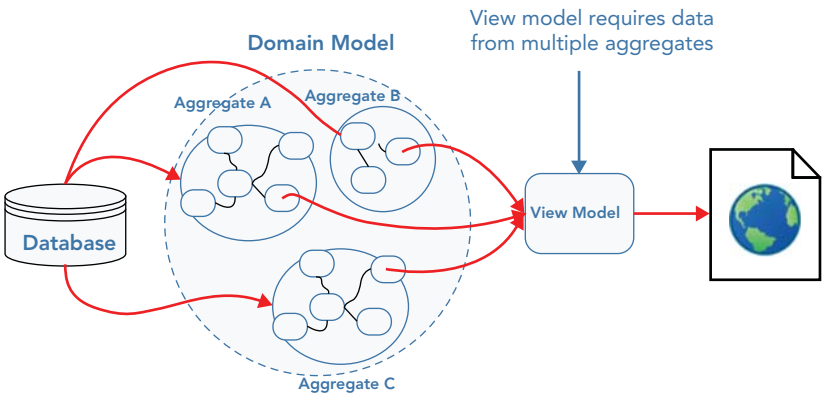


FIGURE 8-10: A view model mapping to many domain objects.

Read Models versus Transactional Models

Sometimes a user interface requires information that spans across many domain objects. It would be inefficient and costly for the application service to hydrate all the rich domain objects to simply provide a subset of information for a view. In these cases, it is preferable for the application service layer to provide a specific view of domain state directly from the data source, as shown in Figure 8-11. This way, you can construct views in an efficient manner without having to construct large object graphs of the domain objects and expose details within them.

There is, however, a drawback to providing read and write capabilities from the same conceptual model, albeit the data model. The transactional model stores logic in domain objects and simple state in the data store. To support both reporting and transactional needs, the views might require extra information that will affect the structure of domain objects. To prevent the model from having to change because of presentation needs, you can store the view data separately in a data schema that is best suited to querying. To achieve this, you can

store changes that occur within the domain model and use these as the basis for reporting requirements.

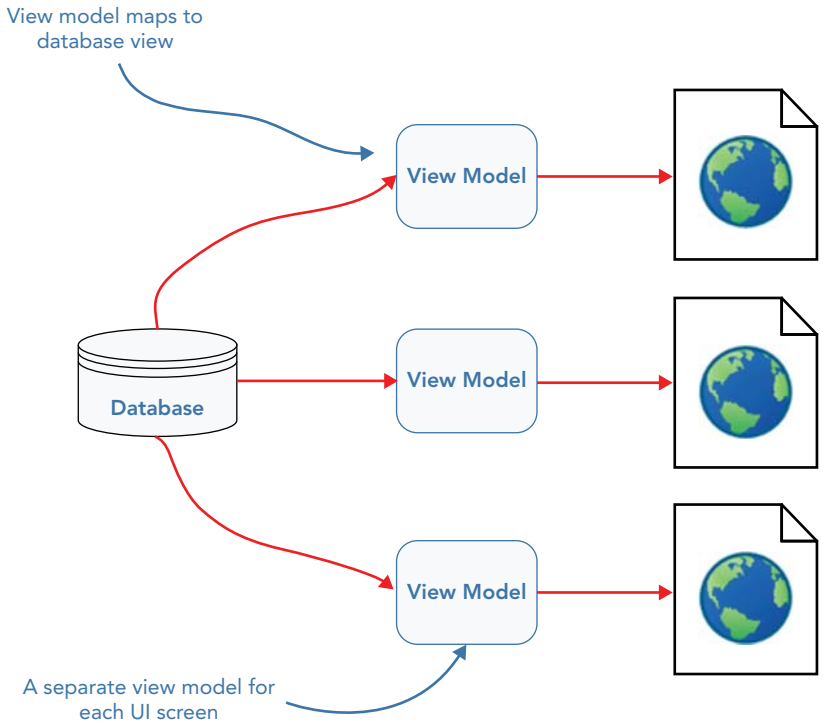


FIGURE 8-11: View models queried directly from the data source.

Figure 8-12 shows how the transactional model handles a write request from a client and then raises events that are stored for querying. You can store these events and their data in the same database or a completely different storage mechanism. This pattern is called Command Query Responsibility Segregation (CQRS) and is covered in greater detail in Chapter 24, "CQRS: An Architecture of a Bounded Context." Further patterns for reporting on a domain model are presented in Chapter 26, "Queries: Domain Reporting."

APPLICATION CLIENTS

The role of the clients of the application service layer is to expose the capabilities of the system. Many applications have some form of presentation or user interface that will give users access to the system behaviors. Other applications instead expose their functionality via RESTful or web services. Regardless of the type of client application, a service should be ignorant to what consumes its functionality. Application services should not bend to meet the needs of a client but instead should expose use cases of an application and force a client to adapt to its API.

It is entirely possible to build a system without an application service layer, relying on the clients to perform all the tasks that the application service layer is responsible for. However, by creating a specific set of services, you are modeling use cases explicitly and keeping them separate from

presentation requirements. These application services help to focus on the behaviors of the systems and enable you to separate domain logic from the other concerns of your application.

Figure 8-13 shows how multiple clients can consume the behaviors of an application via the application service layer. Also shown is how the application service layer can itself consume external contexts and third party services.

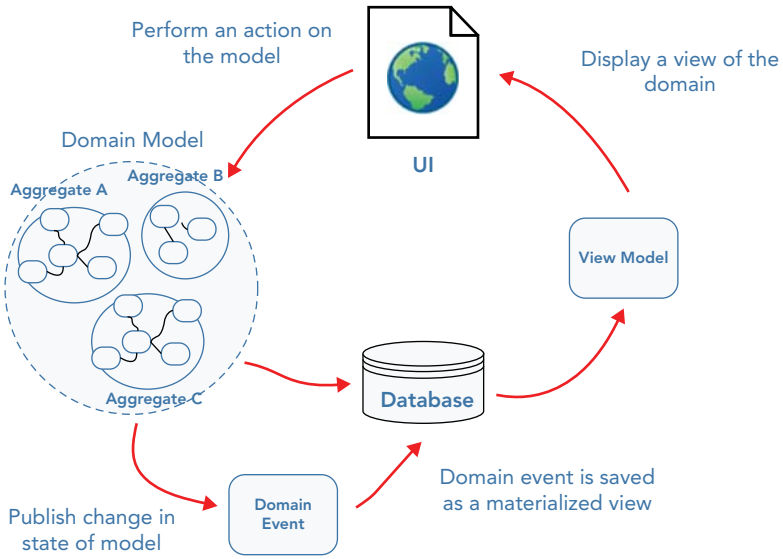


FIGURE 8-12: View store separated from transactional storage.

Bounded contexts can form large systems by communicating via technical infrastructure. Figure 8-14 shows various clients working together to define a larger system. The methods of integrating bounded contexts feature in Part II of this book, with the user interface needs being covered in Part IV.

Sometimes business processes span multiple bounded contexts. For these cases, you employ the use of a process manager to coordinate business tasks. Figure 8-15 shows a process manager that contains business task logic to coordinate larger processes. Similar to the application services, the process managers will be stateless apart from the state used to track task progression, and will delegate back to applications to carry out any work. This pattern is explored in Chapter 25, "Commands: Application Service Patterns for Processing Business Use Cases."

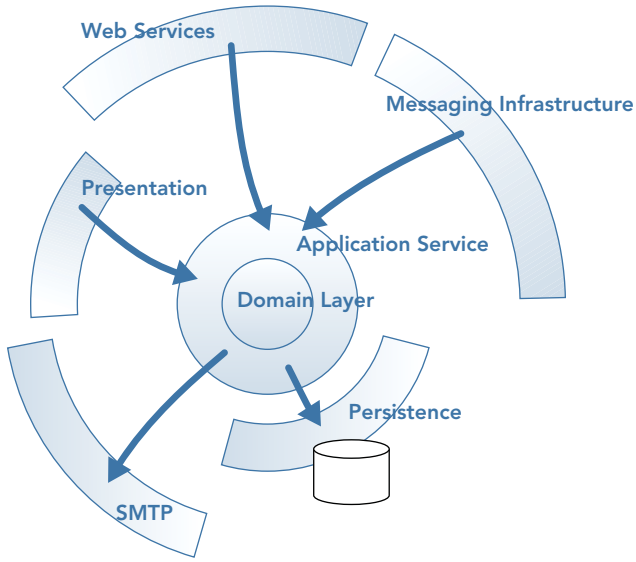


FIGURE 8-13: Various clients of an application.

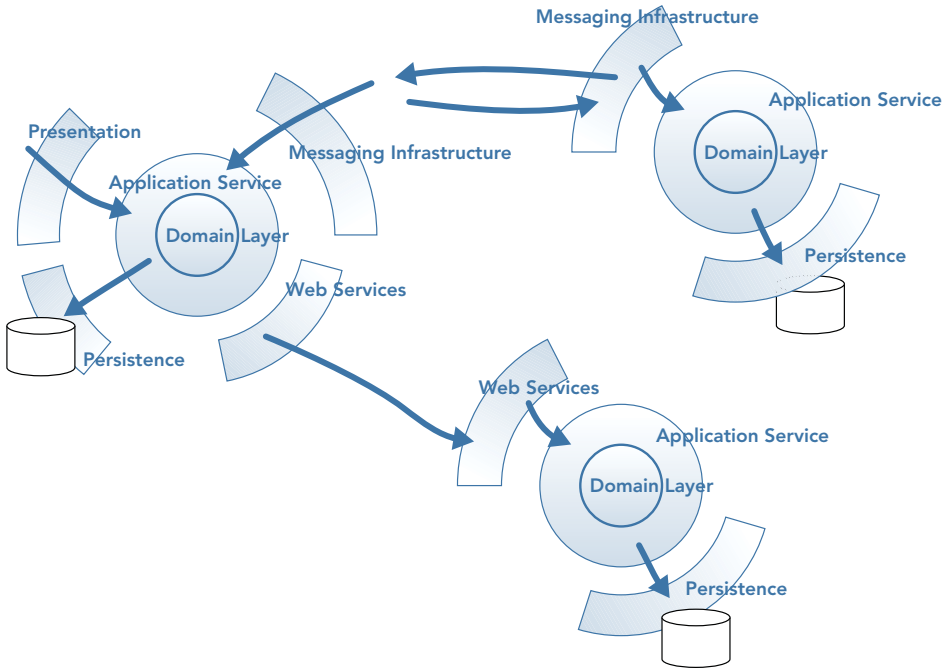


FIGURE 8-14: A system composed of multiple bounded contexts.

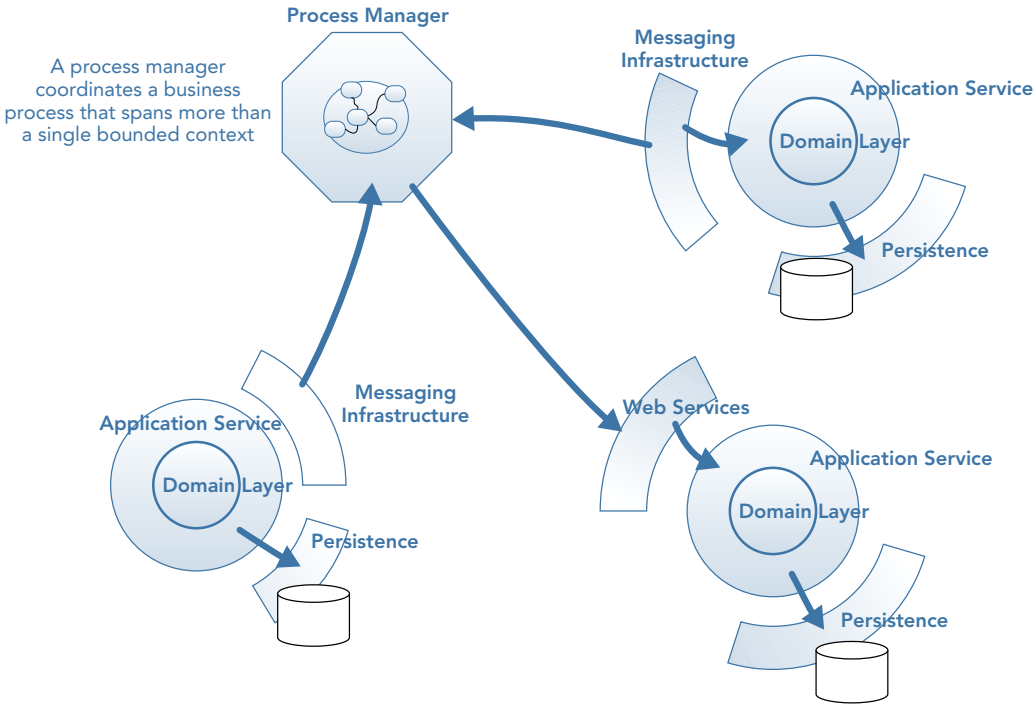


FIGURE 8-15: A process manager.

THE SALIENT POINTS

- DDD does not require a specific architecture—only one that can separate the technical concerns from the business concerns.
- Separate the concerns of your application, and isolate business complexity from technical complexity by layering your application.
- Outer layers depend on inner layers. Inner layers expose interfaces that outer layers must adapt to and implement. This form of dependency inversion protects the integrity of the domain and application layers.
- The domain layer is at the heart of your application. It is isolated from technical complexities by the application layer.
- Application services expose the capabilities of a system by abstracting the domain logic to a higher level.
- Application services are based around business use cases; they are the clients of the domain layer. They delegate to the domain layer to fulfill the use cases.
- Application services should remain ignorant to the clients that consume them. Clients should adapt to the API of the application, which enables the support of discrepant clients.
- The application service layer is the concrete implementation of the bounded context boundary.

9

Common Problems for Teams Starting Out with Domain-Driven Design

WHAT'S IN THIS CHAPTER?

- Understanding why Domain-Driven Design is about more than just writing code
- Avoiding the trap of overemphasizing the tactical patterns
- Why incorrectly applying Domain-Driven Design will make simple problems more complex and frustrate teams
- Realizing that strategic design, collaboration, and communication are more important than the Domain-Driven Design pattern language
- The wasted effort of teams not focusing on the core domain
- The pitfalls teams fall into when applying Domain-Driven Design without a domain expert or an iterative development methodology
- The antipattern of striving for Domain-Driven Design perfection

Domain-Driven Design (DDD) is a philosophy born out of a need to realign the focus of development teams writing software for complex domains. It is not a framework or a set of predefined templates that you can apply to a project. Although there is value in design patterns and frameworks, DDD emphasizes the value in collaboration, exploration, and learning between the business and development team to produce useful and relevant software. Teams should embrace the problem domain they are working within and look outside of their technical tunnel vision. The most fundamental point to being a great software craftsman is to

understand the problem domain you work within and value this as much as you value your technical expertise.

Teams new to DDD or those that do not understand the core concepts can experience problems when applying the philosophy to their projects. These common problems are presented here with explanations as to why teams are finding it difficult to adopt the philosophy.

OVEREMPHASIZING THE IMPORTANCE OF TACTICAL PATTERNS

DDD presents a selection of tactical patterns to help with Model-Driven Design and to aid in the creation of a domain model. A quick Google search on DDD shows you that these building blocks have been overemphasized and are often mistakenly thought of as the most important part of DDD. Evans himself often remarks that he wished he had put the more strategic patterns of DDD rather than the building block patterns at the beginning of the book because most people seem to stop reading after this set of patterns.

A focus on the tactical coding patterns of DDD highlights a bigger problem: technical people who are only focused on technical patterns and writing code. When designing software for systems with complex logic, typing code will never become a bottleneck. The code is an artifact of developers and domain experts working together and modeling the problem domain. The code represents the end of the process of collaboration and discovery. A developer's job is to problem solve, and problem solving is sometimes easier done away from the keyboard in collaboration with domain experts. In the end, working code is ultimately the result of learning and understanding the domain.

Using the Same Architecture for All Bounded Contexts

DDD does not dictate a framework, tool set, or application architecture. You don't have to use CQRS, event sourcing, event-driven, RESTful services, messaging, or object-relational mappers to apply the principles of DDD. What it does insist on, though, is that the complexity of your domain model is kept isolated from the complexities of your technical code. Any architecture that supports this is a good fit for DDD. Domain logic and technical complexity change at different rates; as a result, the organization of these different contexts is key to managing complexity.

Architectures are bounded context and not application specific. The architect for a simple bounded context with low complexity could be composed of a combination of a layered design with the transaction script pattern for the domain layer. A more collaborative domain could employ CQRS, and a complex domain would favor the rich domain model pattern.

Striving for Tactical Pattern Perfection

Teams concerned only with writing code focus on the tactical patterns of DDD. They treat the building block patterns as a bible rather than a guide, with no understanding of when it's okay to break the rules. They spend wasted effort adhering to the rules of the patterns. This energy is better spent on understanding why it needs to be written in the first place. DDD is about discovering what you need to write, why you need to write it, and how much effort you should use. As mentioned before, the tactical patterns of DDD are the elements that have evolved the most since Eric's book was written, with the strategic side of DDD remaining faithful to Eric Evan's original text. How

development teams create domain models is not nearly as important as understanding what models to write in the first place and how to develop them in a bounded context. Understanding the what and the why of problem solving is a more important process than how you are going to implement it in code.

Mistaking the Building Blocks for the Value of DDD

Many DDD projects fail because the tactical patterns of DDD are picked, but the strategic and collaborative sides of DDD are neglected. Teams do not take the time to knowledge-crunch with the business. They do not concentrate on the domain model and on careful abstractions. They don't establish a ubiquitous language (UL). Using only the tactical pattern language of DDD is known as DDD lite. Following a DDD lite approach is fine, but this is not embracing the DDD philosophy. Teams mistakenly thinking that they are applying DDD will be missing out on much of where the value of DDD lies: collaboration, UL, and bounded contexts. Focusing only on the patterns to aid the modeling design omits the bigger picture of problem solving.

In contrast, many of the strategic patterns of DDD can be used in the creation of any medium-to-large-scale business system regardless of the underlying complexity. In fact, all the strategic patterns have many benefits, including identifying whether a UL should be defined and whether the tactical patterns should be used at all. Subdomains can help break down complex problem domains to aid communication and identify what is important. Context maps reveal integration points between different contexts along with the relationships between teams. However, it is sometimes difficult to justify the tactical patterns of applying the domain model pattern to anything other than a complex or constantly evolving domain.

Focusing on Code Rather Than the Principles of DDD

One of the most often-asked questions on software development forums is this: Can I see a sample application? There are probably many good solutions that show the result of a product developed under a DDD process, but much of the benefit of DDD is not revealed when you only examine the code artifact. DDD is performed on whiteboards, over coffee, and in the corridors with business experts; it manifests itself when a handful of small refactorings suddenly reveal a hidden domain concept that provides the key to deeper insight. A sample application does not reveal the many conversations and collaborations between domain experts and the development team.

The code artifact is the product of months and months of hard work, but it only represents the last iteration. The code itself would have been through a number of guises before it reached what it resembles today. Over time, the code will continue to evolve to support the changing business requirements; a model that is useful today may look vastly different to the model used in future iterations of the product.

If you were to view a solution that had been built following a DDD approach hoping to emulate the philosophy, a lot of the principles and practices would not be experienced, and too much emphasis would be placed on the building blocks of the code. Indeed, if you were not familiar with the domain, you would not find the underlying domain model very expressive.

DDD does prescribe a set of design best practices, patterns, and building blocks that are often mistakenly thought to be core to applying DDD to a product. Instead, think of these design artifacts

as merely a means to an end used to represent the conceptual model. The heart of DDD lies deep in the collaboration between the development team and domain experts to produce a useful model.

MISSING THE REAL VALUE OF DDD: COLLABORATION, COMMUNICATION, AND CONTEXT

A team focusing too much on the tactical patterns is missing the point of DDD. The true value of DDD lies in the creation of a shared language, specific to a context that enables developers and domain experts to collaborate on solutions effectively. Code is a by-product of this collaboration. The removal of ambiguity in conversations and effortless communication is the goal. These foundations must be in place before any coding takes place to give teams the best chance of solving problems. When development does start to focus on language, context and collaboration enable code to be well organized and bound to the mental models of the business.

Problems are solved not only in code but through collaboration, communication, and exploration with domain experts. Developers should not be judged on how quickly they can churn out code; they must be judged on how they solve problems.

Producing a Big Ball of Mud Due to Underestimating the Importance of Context

Context, context, context, and more context. It's a fundamental concept in DDD. Context helps you organize solutions for large problem domains. All problems cannot be solved using the same model. Various models need to be created to solve different problems. Creating models within defined context boundaries is essential to keep your code in a manageable state and avoid it turning into a Big Ball of Mud (BBoM). Understanding where contexts end and begin is the responsibility of a context map. Without the notion of context and a context map to guide you, teams cannot deliver value because they are constantly fighting the unorganized mess of their codebase.

If teams don't understand other contexts, changes they make may bleed into those other contexts. Teams without a clear understanding of the boundaries of a model risk violating its conceptual integrity. Blurred or no lines of applicability between models often results in a merging of models, which quickly leads to a BBoM. A context map is vital to understanding boundary lines and how to uphold the integrity of models.

The biggest issue that contributes to legacy code and technical debt is how it's organized. Code is easy to write, but without due care and attention to how it is structured, it can become extremely hard to read. Understanding about context enables you to isolate unrelated concepts so that models are more pure and focused. Think about it like applying the Single Responsibility Principle (SRP) but at an architectural level. Code that is easier to maintain and read will allow teams to deliver value faster, which is the essence of DDD.

Recognize when domain experts are talking in a different context but still using the same terms. If the same terms are used within the business, it is easy to fall into the trap of thinking that the models can be reused. Beware implicit models that are used for more than one context. It's better to create explicit models. Apply the principle of Don't Repeat Yourself (DRY) to a bounded context

only and not a system. Don't be afraid to use the same concepts and names in different contexts. The most important thing teams need to know about is that they should protect their boundaries.

Causing Ambiguity and Misinterpretations by Failing to Create a UL

Effective communication is essential for understanding and solving challenges within the problem domain. Without strong communication, collaboration between the development team and domain expert cannot flourish. Teams that do not value the need for a shared language are likely to employ technical abstractions and build a model using their own shared technical language. When the teams seek help or validation on their model, they are required to translate it for domain experts to understand. At best, this translation is a bottleneck for development; at worst, it can end up with the team building around concepts and themes that are not important or that the domain experts do not understand.

Without a shared language, you cannot create a shared model. This shared vision of the problem enables the capturing of implicit concepts and collaborative problem solving. The process of creating a language is a direct result of collaboration between the development team and domain experts. Being able to easily solve problems and understand the problem domain is where the payoff comes from.

Without a UL, contexts are hard to discover, because a bounded context is primarily defined by the applicability of language. Models created without context and explicit language quickly turn into a BBoM as concepts with the same name are modeled as one.

The formulation of a language can have a big impact on a business and product development. It helps to explicitly define common concepts, and just like a pattern language, remove ambiguity when talking through complex domain logic and business capabilities.

With a UL, domain experts can offer solutions to software problems when implementing the domain model as much as the development teams themselves.

Designing Technical-Focused Solutions Due to a Lack of Collaboration

Without collaboration, key concepts of the problem domain can be missed, and easy-to-understand concepts can be overemphasized. Within an organization, important facets of a domain are often implicit; teams not working with domain experts can overlook these, instead focusing on the lower-hanging fruit like the nouns of a domain. Without collaboration to validate understanding and reveal hinted-at concepts, development teams will abstract around technical terms, and business users will require translation to understand how the problems in the solution space relate to the problem space.

Collaboration is all about getting lots of people with different points of view working on creating a model of the problem domain that can be used to solve problems. No one has the authority on a good idea, and no suggestion is stupid.

Trying to collaborate on knowledge crunching with anyone other than a domain expert can be a wasted effort. A business analyst who may act as a proxy for a domain expert will be able to

give you requirements and communicate inputs and outputs, but he will not be able to assist with shaping a model to fulfill the use cases.

SPENDING TOO MUCH TIME ON WHAT'S NOT IMPORTANT

Teams must understand the underlying reason why they are developing software instead of integrating an off-the-shelf solution. Understanding the strategic vision and the choice of build over buy helps teams concentrate effort. Without a focus on what is core to the success of a project, teams with a limited resource will not apply that resource in the most important areas — the core domains. Spreading resources too thin in the most important areas of a project is an antipattern.

The design of software will suffer without a clear and shared vision of the goal of the product, often captured using a domain vision statement. Well-informed and educated design decisions can be made when developers understand the intent behind business users' requirements. Missing the intent of the business and blindly developing leads to poor results.

Accepting that not all of a system *will be* well designed and that not all of a system *needs* to be well designed is a big step forward for a team. Without a focus on the key aspects of a system, talented members of a team may be distracted by frameworks and instead want to work on the latest JavaScript framework at the presentation layer instead of the core aspects of a product.

In addition to teams, it's important that domain experts have a clear understanding of the core domain. A domain expert who does not share the vision of a stakeholder or is unsure why the software is being written will not be effective during knowledge-crunching sessions due to negativity or confusion.

MAKING SIMPLE PROBLEMS COMPLEX

Applying techniques designed to manage complex problems to domains with little or no complexity will result in at best wasted effort and at worst needlessly complicated solutions that are difficult to maintain due to the multiple layers of abstractions. DDD is best used for strategically important applications; otherwise, the deep knowledge gained during DDD provides little value to the organization.

When creating a system, developers should strive for simplicity and clarity. Software that is full of abstractions achieves little more than satisfying developers' egos and obscuring the reality of a simple codebase. Developers who aren't engaged with delivering value and are instead only focused on technical endeavors will invent complexity because they're bored by the business problem. This kind of software design can lead to frustration for teams in the future that need to maintain the mess of technical layers.

Applying DDD Principles to a Trivial Domain with Little Business Expectation

Simple problems require simple solutions. Trivial domains or subdomains that do not hold a strategic advantage for businesses will not benefit from all the principles of DDD. Developers who are keen to apply the principles of DDD to any project regardless of the complexity of the problem domain will likely be met with frustrated business colleagues who are not concerned with the less important areas of a business.

DDD is a set of principles to help you manage complex problem domains that hold significant advantage for a business. Problems with a low business expectation should be tackled in a no-thrills manner. This is not to say that they should be built in a haphazard manner. On the contrary, they should be built to be performant and maintainable, but problems that have little logic need straightforward solutions.

Large complex systems will have many subdomains, some containing complex logic key to strategic importance of a product, whereas others will simply be forms to manage data with little or no complexity. The tactical patterns of DDD along with collaborating to build a UL to communicate models should be reserved for the core subdomains only. These are the areas that need to be clear to aid rapid change or that model complicated and intrinsic logic. Teams should not waste energy on the generic domains or subdomains beyond keeping them working and isolated from the core domains.

Disregarding CRUD as an Antipattern

Not all of your system will be well designed; trying to perfect an entire codebase can be wasted effort. Your focus and energy should be on the core domain, for anything else good is good enough. For systems with little or no domain logic and with no more than forms over data opt for a simpler form of architecture such as a create, read, update, and delete (CRUD)-based system to decrease time spent and increase availability for the core domain.

Using the Domain Model Pattern for Every Bounded Context

The domain model pattern is useful for complex or frequently changing models. The effort required to employ the domain model pattern for models that are generic or lack domain logic will be far greater than any value that will be gained. Utilize model-driven design and the domain model pattern for the core domain, and use other domain logic patterns for simpler parts of your system.

Ask Yourself: Is It Worth This Extra Complexity?

When junior developers learn about design patterns, they try to apply them to every piece of code they write. This behavior is often seen when teams learn about DDD. They focus only on the tactical patterns of DDD, blindly applying these patterns to every project regardless of whether it is justified. This eagerness to apply a new philosophy without due consideration as to whether the process is a worthwhile endeavor for the software can lead to needless complexity where a simple solution would have sufficed.

Don't develop nonstrategic software if off-the-shelf software will suffice. If the effort to automate a manual process is too great, just leave it manual. Remember: Solutions don't always have to be technical.

UNDERESTIMATING THE COST OF APPLYING DDD

Applying the principles of DDD is hard and costly both in time and resource. It makes sense to only fully apply them to the most important areas of your system: your core domain. The principles hang on a business willing to work with you on solutions rather than have you work in isolation. DDD often is more valuable to the nontechnical parts of product design.

Trying to Succeed Without a Motivated and Focused Team

DDD is not for everyone. It is certainly not the right fit for every project. To gain the most benefit when following DDD, you need a complex core domain that will be invested in over time, an iterative development process, and access to domain experts. However, that is not all that is required. There are a number of other skills that you need to succeed at DDD. To be effective at DDD, you need the following:

- Solid design principles required for refactoring
- Sharp design sense
- A focused, motivated, and experienced team

You need disciplined developers who are willing to work with domain experts and understand the business rather than worry how they can wedge the latest JavaScript framework into a project.

Remember: DDD isn't a silver bullet. Just as switching from an upfront waterfall approach to a more agile/XP project methodology didn't solve all your software development problems, opting to follow DDD won't suddenly produce better software. The common denominator in any successful project is a team of clever people who are passionate about what they are doing and who care about it succeeding.

Attempting Collaboration When a Domain Expert Is Not Behind the Project

Business ownership and investment especially from domain experts is key to successful collaboration. If a development team is working alongside domain experts who are not invested in the project or do not understand the intent or vision, they will unlikely discover a useful model, create a UL, and work as an effective team. Development teams are great at designing systems to handle challenges in the problem domain. However when collaborating with a domain expert, they can go further and work together to remove the need for software solutions by removing issues at the source and redefining business processes.

Learning in a Noniterative Development Methodology

DDD is about brainstorming. It's about collaborative learning. It's about not stopping at your first idea but continuing to experiment so you discover something better or simply to validate your initial idea. All this takes time, and a methodology that doesn't support this can't support DDD.

A useful model will not be created on the first attempt; therefore, an iterative development methodology is required to hone a design. Models evolve. Teams that don't appreciate that the model and language are only valid for a given time will quickly see their useful creation turn into a BBoM. A model needs love. It needs to be refined and refactored as more insight into the domain is gained and as new use cases challenge the model.

It's also worth noting that, to experiment and evolve a model, you need to have the safety of unit tests. That is not to say that you must follow a test-driven process; instead, you must ensure your code can be valid after a series of refactors.

Applying DDD to Every Problem

For every solution, there must be an appropriate problem. DDD is a design philosophy suited to a particular problem space. It is a great tool to have in your toolbox. However, if your business is not complex or isn't changing frequently then don't automatically reach for the DDD hammer: remember there are other better suited tools in your development tool kit. Only focus your modeling efforts and DDD on the most complex bounded contexts that bring the most value to your customer.

Not all subdomains are complex. Some domains or contexts may not even need a fully fledged domain model and may just contain data with no business logic that simply requires the basic CRUD operations. For low-complexity contexts, favor the use of a CRUD/Active Record/Transaction Script-based application, and leave the tactical patterns of DDD for parts of your system that are important to your customer, that are complex, and that change frequently.

Ask yourself: Is this extra effort helping you deliver your solution, or is it slowing you down? Keep things simple but not simplistic. Don't over engineer a solution or try to leverage unhelpful frameworks. Keeping things simple is an art form and takes practice and a pragmatic mind-set.

Sacrificing Pragmatism for Needless Purity

Don't try to strive for perfection in areas that don't need it. For generic or supporting subdomains, keep things simple, straightforward, and uncomplicated. Use CRUD and simple domain logic patterns. Get the code written so it works; then move on to the core domain. The core domain is the area where you can strive for perfection. Small balls of mud are sometimes better in bounded contexts that are unimportant; they get the code written quickly and get it out of the way. If you need to change it, you can overwrite it. For areas of your product that are unimportant, unlikely to change or be invested in over time, favor working code over perfect code. Good is often good enough. Don't worry if you are doing it right or start to seek confirmation; this will be wasted effort and a distraction. Leave purity for the areas that count.

Wasted Effort by Seeking Validation

When you build a system following the principles of DDD, you do not receive a certificate in the post from Eric Evans congratulating you on your achievement. Blindly following any patterns language or methodology without considering your own unique context is foolhardy. Trying to adhere to a set of rules for no other reason than to compile with a methodology is an antipattern. The DDD philosophy is not about following a set of rules or applying coding patterns. It is a process of learning. The journey is far more important than the destination, and the journey is all about exploring your problem domain in collaboration with domain experts rather than how many design patterns you can employ in your solution.

Search forums and read DDD blog posts to discover how teams are collaborating with the business to aid learning and increase discoveries. Don't try to create the perfect repository pattern, and don't seek confirmation from your peers who aren't involved directly in your project because without the full context, you must take any advice lightly.

Always Striving for Beautiful Code

Teams that obsess with applying design patterns and principles regardless of the actual need will likely create overly complex and confusing architectures that miss the goal of the product in the first place. Teams should understand the motivations behind design patterns and use them judiciously. Blindly employing the tactical patterns of DDD does nothing to add value for the business.

A supple design in important areas that frequently change aids a model's ability to be flexible and evolve without having large rippling effects. Painstakingly striving for elegant design in areas that offer little business value and will not be invested in is a waste of efforts. It is far better to have small balls of mud, isolated from other contexts that can easily be replaced, rather than trying to strive for beautiful code everywhere.

When working in the core domain, teams should certainly wait before committing to a pattern and a way of thinking. Delaying refactoring and living with the code to see what causes friction/changes the most can reveal more about the domain and lead to a more informed design choice.

Don't be distracted by patterns, frameworks, or methodologies; they are implementation details. Your goal is to understand your domain at a deeper level to be best equipped to solve problems within it. This is the true value of DDD.

DDD Is About Providing Value

Don't let design patterns and principles get in the way of getting things done and providing value to the business. Patterns and principles are guides for you to produce supple designs. Badges of honor will not be given out the more you use them in an application. DDD is about providing value, not producing elegant code.

THE SALIENT POINTS

- The tactical patterns of DDD can guide you toward creating an effective domain model; however, this area of DDD is evolving, and the implementation details have been overemphasized. The patterns may have value, but this is not where the value of DDD lies.
- DDD is far more than coding. Collaboration with domain experts to knowledge crunch and have a shared understanding of the problem domain expressed in a ubiquitous language are the pillars of DDD.
- Context is everything; context and isolation retain the integrity of your code. It reduces cognitive load and makes a model specific.
- You need a smart dedicated team willing to learn about the domain.
- You need access to a domain expert. Teams can't reveal deeper insights without them.
- Use CRUD for bounded contexts with low complexity. You are not a bad programmer if you don't have a domain model.
- Bounded context and the ubiquitous language are the foundation of DDD.
- DDD is about the process of learning, refining, experimenting, and exploring in the quest to discover a useful model in collaboration.

10

Applying the Principles, Practices, and Patterns of DDD

WHAT'S IN THIS CHAPTER?

- How to sell Domain-Driven Design
- Applying the principles and practices of Domain-Driven Design to your project
- Understanding the importance of exploration and experimentation in the quest to find a useful model
- Why avoiding ambiguity will greatly improve your modeling efforts
- Removing the complexities of technology when problem solving
- Why you shouldn't worry about the perfect domain model
- How you know when you are doing it right

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 10 download and individually named according to the names throughout the chapter.

Over the previous nine chapters, you have gained an overview into the philosophy of Domain-Driven Design (DDD). This chapter brings all of that knowledge together to show you how you can start to apply the principles and practices of DDD to your next project. The remainder of this book focuses on coding patterns to produce an effective domain model in code, integrate bounded contexts, and architect maintainable applications.

SELLING DDD

DDD is not a silver bullet, and it shouldn't be sold as one. In the same way that following an agile methodology won't solve all of your problems, neither will DDD; however, it is a powerful and extremely effective philosophy when used in the correct circumstances, such as these:

- You have a skilled, motivated, and passionate team that is eager to learn.
- You have a nontrivial problem domain that is important to your business.
- You have access to domain experts who are aligned to the vision of the project.
- You are following an iterative development methodology.

Without these key ingredients, applying the principles and practices of DDD will overcomplicate your development effort rather than simplify it. However, if your particular circumstances meet the preceding list, then applying the principles and practices of DDD can greatly increase the value of your development effort.

Don't sell DDD as a project methodology; instead, understand and apply the principles appropriately and where you can gain value. Just as design patterns are best arrived at when you refactor, the principles and practices of DDD should be used only when necessary, and with each on its own merit. Apply any technique judiciously and only when you can gain an advantage and it can give you value.

Educating Your Team

When communicating the philosophy of DDD to your team, focus not on the pattern language but instead on the alignment with the business, the importance of strategic contexts, and the focus on language used to describe a model. The principles and patterns of strategic DDD are far more powerful and useful than the tactical patterns. A team focused solely on the software patterns will miss the true value of DDD and will likely pay the cost of applying the practices without reaping the rewards.

Technology is not the solution to business problems; it is merely an implementation detail. Problem solving is achieved through collaboration with domain experts who hold the key to discovering a useful model. This model is brainstormed on cards or a whiteboard before being implemented in Visual Studio (or your favorite IDE). Remember: Technology can complicate problem solving; focusing on an abstract conceptual model free from the clutter of infrastructure and technical concerns will enable teams to build a solution that communicates the intent of the domain and can be understood by the business.

Speaking to Your Business

Your business stakeholders don't want to hear about the next new development philosophy or methodology; they just want you to deliver value for them. Agile, Service-Oriented Architecture (SOA) and the cloud have all been overhyped, promising to solve all development problems but failing to do so. Instead, talk only of your desire to learn more about the business you work within to give more value. Talk of the need for the development team to be more aligned to the vision and intent of the business. A stakeholder will see the value in the development team spending time with business experts and aligning themselves to the expectations of the business.

DDD does not work without the commitment of domain experts. If you can articulate the importance of having an expert from the business available to your stakeholders, you are setting yourself up to succeed. Of course, you are free to talk to your stakeholders about DDD, but it's best to focus on the need for collaboration. The success of a product falls on the commitment level of the business and its experts; this is how you sell DDD. Empowering a team with a deep understanding of the problem domain logic enables it to produce a better product.

APPLYING THE PRINCIPLES OF DDD

With a business that understands the value of a domain expert's time and a development team that is focused on driving a solution to a complex problem around a model of the domain created in collaboration with a domain expert, you will be in a good position to apply the principles of DDD. This section details the stages of project inception to developing a solution, and which practices to use when. However, the key to applying DDD is to start simple. Do the simplest thing possible until you encounter complexity or ambiguity. When you find ambiguity during conversation, explicitly define it within the ubiquitous language (UL). When your model is becoming too large, decompose the complexity and apply the strategic patterns of code organization and modeling techniques. If you keep things simple and arrive at applying the principles and practices rather than trying to crack a nut with a sledgehammer, you will get immense value from DDD.

Understand the Vision

Before capturing requirements, it's important to align your team with stakeholders' expectations. Start a meeting with a stakeholder by asking open questions that draw out the reason you are opting to build rather than buy a product, and that help to explain the vision of the product. The following questions are powerful:

- What is the business goal/driver for this product?
- What value will this product bring to the business?
- How will you know if this is a success? What does good look like?
- How is this different from what has been done before?

By listening to the stakeholder answering your questions, you can identify the most important part of the product—the area of the software that is the fundamental reason you are building it. Capture this information and create a domain vision statement that aligns your team to a common goal. Your team needs to understand what will make or break the product, what is essential, and where the value is.

During the meeting, you may discover that the product is not special and is in fact a generic solution that just happens to be more cost effective to build in-house. If so, make that explicit, and understand its importance to the business and its future. It could also be the case that this is a fail-fast product—a prototype to test the appetite of customers. These insights will enable you to understand where the value lies and what the business is trying to achieve. With this understanding, you can determine if all the practices of DDD are going to be effective for your project.

Once you understand and share the intent of the product and the stakeholders' goals, you can capture the features of the product while always aligning to the overall vision of the product.

Capture the Required Behaviors

An effective way to engage stakeholders when gathering requirements is to apply the behavior-driven development (BDD) methodology. BDD is a shared language that helps you capture the behaviors of a system. You can think of it as a UL for requirements. It enables your team to understand the requirements from your stakeholder from a nontechnical perspective. The practice of BDD and the focus on capturing requirements in a language the business understands and which explicitly shows value is a great exercise in demonstrating the power of a shared language to aid communication. This process helps to educate the team on the power of language and the evil of ambiguity in meaning. Use these sessions to explicitly define terminology. This exercise will warm up the team members for when they collaborate on a UL to describe the model that will implement the rules, logic, and processes needed to satisfy the system's behaviors.

What you are capturing from the stakeholders are use-cases, inputs, and outputs. These business use cases form your application services. If they are complex, they drive the decision on what domain logic pattern you will implement. During requirements gathering, focus on what the stakeholder wants, when, and why. The why part is essential. Asking the question helps to validate why the stakeholder wants what he says. During the requirements stage, stay in the problem space, and focus on the opportunities that this product is going to bring. How you are going to satisfy the requirements can wait until you understand and share the vision.

I have seen many teams rush through requirements, eager to jump to a solution without fully exploring the problem. Don't jump to a solution too quickly; ensure that you explore the problem space with your stakeholders. Often, stakeholders are unclear on exactly what they want. By exploring the problem space, you can draw out the real business need and often offer better behaviors before wasting time on solutions to needs the business doesn't really have.

As you start to generate story cards full of features, you may find that the size of the product is becoming too big to manage or ambiguity is occurring in the problem space. You might also be losing sight of the bigger picture and the core reason that this product is being built. When this occurs, the problem space needs to be distilled.

Distilling the Problem Space

If you find that the problem domain is becoming too large to manage, you can ease cognitive load by abstracting the problem to a higher level by creating subdomains. It's useful to look for the capabilities that support the product and create subdomains from these. Business capabilities are the activities that support a business process; look for these activities outside of departmental structures or functions. Some will be generic, some supporting. The ones of real interest that will make or break the product are the core domains.

Focus on What Is Important

When you have decomposed your problem space, ensure that you spend the majority of your time with stakeholders understanding required behaviors for the core domain. Your core domain may very well be small, which is fine. Pay particular attention to conversations in this area, because they will be the most interesting and offer the most value to you. The core domain should directly support the overall vision that the stakeholders have; if it does not, you may have incorrectly identified the core domain, or you may need to clarify the vision with your stakeholders.

Understand the Reality of the Landscape

With a thorough understanding of the problem space and an alignment on where the value of the system is, you can start to model a solution. However, before you start creating a solution to any project, it's of utmost importance to understand the environment that you will be working in. Understanding the state of the software solutions already in production is essential to making informed decisions on how you will integrate your product. The best way to capture the landscape is by creating a context map.

The team needs to identify the different bounded contexts in play that will directly affect or be affected by your product. The team needs to identify how these contexts interact, what protocols they integrate through, and what data they manage. To achieve this, follow these steps:

1. Determine what models the team is aware of that directly affect or will be affected by the problem area. Draw these contexts, name them, and show who is responsible for them. If you are not sure where to start, look at the organizational structure of the domain you are working in, because most systems are built around departmental communication. Then look at the development team structures.
2. Map the integration points and methods of integration.
3. Map the data that is exchanged and who owns it.
4. Label the relationship between the contexts. Is your team downstream and reliant on another team? Or is your team upstream and must communicate changes to other teams downstream from it?
5. Rinse and repeat until you have captured all that you can about the landscape you will be developing in.

The whole team must understand the context map. Hang it on the wall for all the team to see and other teams to understand. This is your war map. It should be simple enough for all to draw quickly, so don't spend too much time on UML diagrams. Instead, get a bird's-eye view, and as you start to integrate, zoom in on touch points and then go into detail.

Modeling a Solution

Before starting to model a solution and applying the principles of DDD, you need to ensure the product you are about to provide a solution for meets the following criteria:

- Is a complex problem or has complexity in a subdomain
- Is important to the business and has high expectations of it
- Has accessibility to a domain expert
- Has a motivated and smart team

If you don't have a complex problem or a portion of your problem is not complex, then building a domain model may be overkill. When there is little logic and merely data manipulation, you should follow a simpler method to manage domain logic, such as transaction script or active record. That way you can avoid some of the more costly practices.

If the product is not important to the business and there are low expectations, it's probably not worth the effort of building a solution to stand the test of time. Build good enough and build for replacement rather than investment. You can try and utilize an off-the-shelf solution. If your problem is generic there may well already be an open source solution out there that fits your needs.

If you don't have access to a domain expert, discoveries in the domain will not happen. The team will abstract around technical concerns, and the language in the codebase will not reflect the problem domain. Contexts will not be well defined, and quickly the code will evolve into a ball of mud.

If your team is not motivated or lacks the knowledge of enterprise design patterns and principles, it may be best to favor a simpler pattern to organize domain logic so you don't overcomplicate the development efforts.

All Problems Are Not Created Equal

Not all parts of the problem space require the full spectrum of the principles and practices of DDD. You must pick your battles. For areas of your solution that don't require the collaboration with a domain expert, don't involve her. For areas that don't require the domain model pattern to represent an abstract model in code, don't create one.

If a portion of your problem space is complex, your business has high expectations, you have access to a domain expert and a team up to the challenge, you have the exact criteria for which DDD can help you manage the development of your product.

Engaging with an Expert

A domain expert is a subject matter expert with deep knowledge of the problem domain. Whereas a stakeholder defines what the system needs to do, a domain expert collaborates with the development team, using his insight, expertise, and experience to model a solution that satisfies the behaviors. A domain expert could be a long-term user of a current system that has in-depth knowledge of the processes and logic of the problem space. The domain expert could equally be the systems product owner or simply someone who works in the department and has worked in the domain for many years. The point is that the term domain expert does not refer to a title; it's anyone in the business who can offer expertise in the problem domain.

DDD doesn't work without a domain expert. It's as simple as that. Without a domain expert, much of the insight and rich domain knowledge and language will not be discovered. Without an expert, the development team may seek advice from users of a current system, and while knowledgeable on the current processes, may not be best placed to provide game-changing wisdom or domain intelligence. It cannot be stressed enough the importance of seeking out a domain expert and engaging with that person. A domain expert will have a day job; her time will be precious, so it is vital that you utilize time with her wisely.

Business analysts are not invalid. They hold skills that developers and domain experts may not possess. Business analysts can facilitate conversations with domain experts and can help the team capture terminology that forms the UL.

It is important for the stakeholder to trust the domain expert and regard this person as an expert. It is also important for the expert to understand why the project is being undertaken and what its goals are. A domain expert at odds with the nature of the project may turn out to be more of a

hindrance than a help. However, her concerns and challenge with the project might be justified. In this case, ensure that the stakeholder and domain expert communicate to allow any fears or worries to be alleviated.

Try to collocate your project team with the business. Your team should be able to access the domain experts and users easily and regularly to ensure constant feedback. Domain experts are your primary source of knowledge when validating your domain model. Extract as much information from the heads of your domain experts as possible. By facilitating domain experts' knowledge, you unlock a more useful model.

Select a Behavior and Model Around a Concrete Scenario

When working in the solution space, ensure that you focus on satisfying the behaviors of the product rather than trying to model the entire problem domain. Drive your modeling endeavors by selecting a behavior and defining concrete scenarios to use as examples. From this, the team and the domain expert can produce a model that is appropriate to the problem at hand. This practice helps to prevent overzealous developers from producing a one-model-to-rule-them-all view of the problem domain that isn't really tailored to the needs of the system, and is more a reflection of reality rather than a useful abstraction of it.

As an example, consider this coupon feature for an e-commerce domain:

> To increase customer spending
>
> As a shop
>
> I want to offer money-off coupons

To start to shape a model for this feature, you must model to meet specific concrete scenarios. This feature has several scenarios associated with it. The following is one example:

> Scenario: A customer receives a discount on a basket.
>
> He has a coupon that offers a discount of 10% off the value of a basket.
>
> The coupon is valid for baskets that have an initial total exceeding $50.
>
> When a coupon is applied to a basket with a total of $60, the discount should be $6.

During knowledge crunching, the team should listen to the domain expert's choice of language and capture concepts that are used to fulfill the scenario. If the team spots potential issues or problems with the model, it should challenge them with the domain expert.

Collaborate with the Domain Expert on the Most Interesting Parts

When picking scenarios to model, don't go for the low-hanging fruit; ignore the simple management of data. Instead, go for the hard parts—the interesting areas deep within the core domain. Focus on the parts of the product that make it unique; these will be hard or may need clarification. Time spent in this area will be well served, and this is exactly why collaboration with domain experts is so effective. Using domain experts' time to discuss simple create, read, update, and delete (CRUD) operations will soon become boring, and the domain expert will quickly lose interest and time for you. Modeling in the complicated areas that are at the heart of the product is exactly what the principles of DDD were made for.

Evolve UL to Remove Ambiguity

Ambiguity alongside ignorance of your problem domain is your worst enemy as a developer. Teams must ensure that during modeling and knowledge crunching, all terms and language are defined explicitly and the domain expert is happy with the terminology. Everyone must be on the same page and have a single understanding of a concept. Besides talking to domain experts and each other in a single language, you must write the codebase with the same terms and concepts to ensure the code model reflects the mental models in conversation.

The UL is formed from the knowledge crunching exercise between domain experts and the development team as they start to model a solution to the more important and trickier parts of a product. Clear and unambiguous communication between the development team and the domain expert is vital to enable discoveries and to reduce translation cost between the team's code model and the domain expert's mental model. Teams that talk to domain experts about design patterns, principles, and practices will soon lose their interest due to the painful and costly translation that is required. The model, even though implemented in code, should be discussed at a higher level of abstraction so that the domain expert can lend his expertise to solving challenges with every new scenario that is thrown at it.

As you gain a deeper understanding of the domain you are working in, your UL will evolve. As the language evolves, so must your code. Refactor your code to embrace the evolution by using more intention-revealing method names. If you find a grouping of complex logic starting to form, talk through what the code is doing with your domain expert, and see if you can define a domain concept for it. If you find one, separate the logical grouping of code into a specification or policy class.

WHAT IS A SPECIFICATION?

A specification represents a business rule that needs to be satisfied by at least part of the domain model. You can also use specifications for query criteria. For example, you can query for all objects that satisfy a given specification.

Throw Away Your First Model, and Your Second

When you are starting out on a project, you know little about it, but this is the time when you will be making important decisions. Your initial model will be wrong, but don't get too hung up. The process of learning more about the problem domain is achieved over many iterations. Your knowledge will grow, and with this you will be able to evolve your model into something useful and appropriate.

When arriving at the first useful model, most teams usually stop exploring and jump to their keyboards to implement it. Your first model will unlikely be your best. Once you have a good model, you should park it and explore the problem from a different direction. Exploration and experimentation are vital to enable deep discoveries and to learn more about the problem domain; therefore, make mistakes and validate good ideas by comparing them to bad ones.

Sometimes while modeling, you become stagnant; your solution may have painted you into a corner, and a new scenario cannot be fulfilled with the current model. This is fine. Instead of trying to make the scenario fit the model, make a new model that is useful for the existing and new scenarios.

Try to unlearn everything you gained for the first model and take a new direction. Explore and experiment to reveal insights and offer new solutions.

The result of tackling a problem from various angles is not the creation of a perfect model but instead the learning and discovery of concepts in the problem domain. This is far more valuable and leads to a team able to produce a useful model at each iteration.

Implement the Model in Code

Once you have derived a model for the complex subdomains of your problem domain from sessions with a domain expert, you need to prove it in code. The mental model that was created between you and your domain expert should be reflected in code with the same terminology, language, and concepts. Once it turns the mental model into code, the development team may find that the model does not meet the needs of the scenario, and it needs to make a new concept or change an existing one. Because of the use of the UL and the shared understanding of the model throughout the team, communication with the domain expert is easy, and the new solution can be validated in collaboration and without translation. The update to the code model is reflected in the mental model, and the two models evolve together.

Creating a Domain Model

The creation of a domain model is an iterative exercise, and the quest to discover a useful model will see it constantly evolve as new business problems are tackled with it. It is important not to try to model the whole problem domain but instead select well-thought-through business scenarios that can be used as an example to test any model produced.

A domain model should constantly adhere to these two principles:

> **Be relevant:** Able to answer questions in the problem domain and enforce rules and invariants.
>
> **Be expressive:** Able to communicate to developers exactly *why* it's doing *what* it's doing.

The creation of a useful model is never completed at the first attempt. In fact, often the initial incarnation of a domain model is naive and contains little insight into the rich problem domain. Instead, constant refactoring is required to expose domain knowledge within the codebase.

Evolution and an effective model are discovered through exploration and experimentation. BDD and Test-Driven Development (TDD) allow you to experiment, knowing that the inputs and outputs won't be affected. Start with an anemic domain or simpler patterns, and refactor toward the rich domain model when needed. Model only when the problem requires a complex solution or the team is unsure of or new to the problem domain (for example, the team has never worked in finance).

Keep the Solution Simple and Your Code Boring

Keep your model simple and focused, and strive for boring plain code. Often teams quickly fall into the trap of overcomplicating a problem. Keeping a solution simple does not mean opting for the quick and dirty; it's about avoiding mess and unnecessary complexity. Use simplicity during code review or pair programming. Developers should challenge each other to ensure they are proving a simple solution and that the solution is explicitly focused only on the problem at hand, not just a general solution to a generalized problem.

Carve Out an Area of Safety

If you are working in a legacy codebase or are integrating with a legacy code, it is vital to ensure that your code is not contaminated by the mess that already exists. (If there is mess; remember that legacy doesn't mean bad code!) It may be tempting to clean up the legacy codebase, but this is a task that can quickly become time consuming and distract from your real goal of introducing new functionality. Instead, lean on the anticorruption layer pattern to create a boundary between your new code and the existing code. This protection boundary enables you to create a clean model that is isolated from other teams' influences.

Integrate the Model Early and Often

While modeling, it is important to validate and prove the model in code as early as possible. Whiteboard drawings and cards are good, but working code is the only true measure of progress. Working code means integrated code. Working code connects to databases and to user interfaces; it proves the model from an end-to-end process.

Nontechnical Refactoring

Experienced developers are familiar with employing technical refactorings to increase the quality of the software construction by migrating to well-known code organizing patterns. However, a different type of refactoring is required to ensure a model communicates what it does clearly. It's important to reflect in code any domain knowledge breakthroughs that happen with domain experts or indeed anyone working on the software product. Code within the domain model should be clear and expressive.

A domain model starts out simple based on a shallow view of the domain, usually based on the nouns and verbs of the requirement documentation or from initial discussions with domain experts. However, as the depth of knowledge within the team grows through iterations, so should the richness of the domain model. Even if the technical design of the code has been modified to increase clarity, the names and methods of classes, along with the namespaces, should continue to be updated to reflect the more insightful abstractions that are discovered through knowledge-crunching sessions. This continued investment in the model helps keep it relevant and match the vision and intent of the domain experts.

Decompose Your Solution Space

As your model grows, you will find ambiguity in language or overloaded terms, or you may just find it difficult to manage due to its size. To make large and complex domain models simpler and easier to maintain, divide by context based on natural language and invariants. Focus on minimizing the coupling between contexts. Don't strive for perfect code; strive for perfect boundaries. Bounded context and aggregates are powerful concepts in DDD that enable complexity to be reduced. These patterns help to manage complexity.

You should arrive at the strategic patterns of DDD due to complexity rather than up-front design. Boundaries are hard to remove when they are defined so refactor toward them after several iterations of development and when you are more knowledgeable within the problem domain. Favor small modules of code with a focus on boundaries that you can replace rather than perfection within those boundaries. Isolate and protect data.

Rinse and Repeat

Figure 10-1 visualizes the steps of applying the principles and practices of DDD.

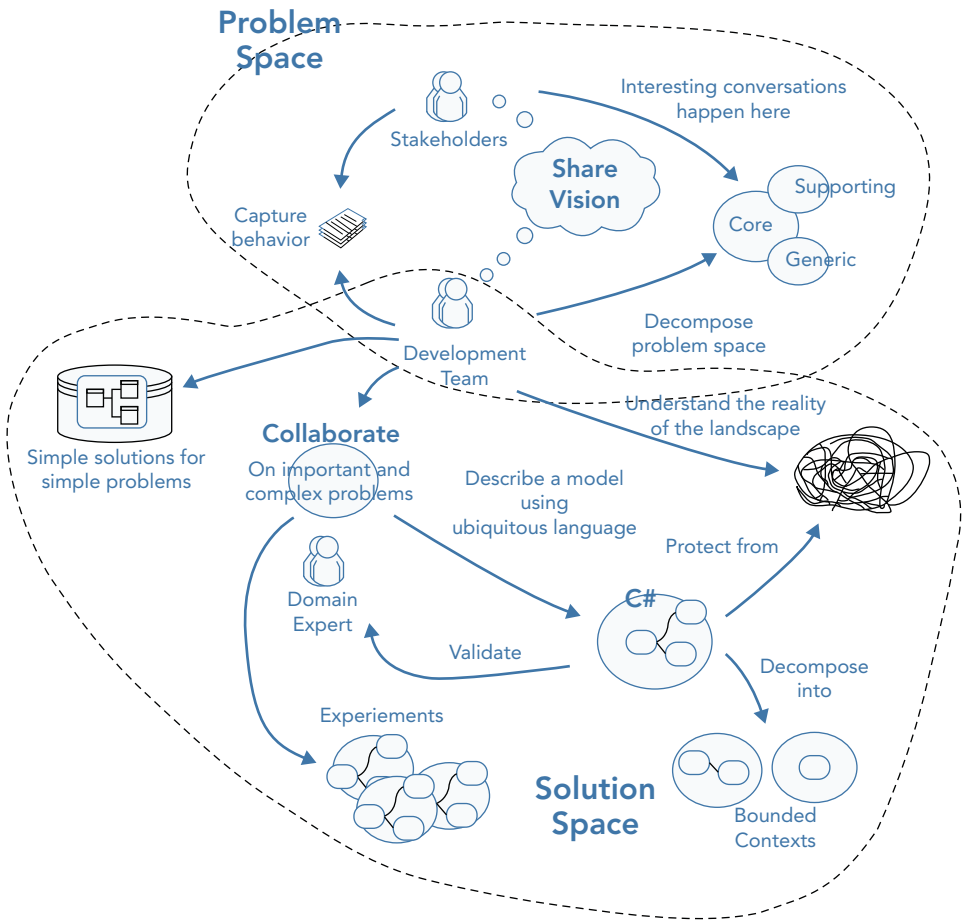


FIGURE 10-1: The process of DDD.

A model is constantly evolving and changing; you cannot effectively utilize the practices and patterns of DDD without embracing evolution. You should perform the steps presented in this section constantly. Keeping the complexity of the software solution as low as possible is the goal of DDD. All the principles and practices are aimed at this overall goal. As a developer, it is your job to continuously challenge the effectiveness and simplicity of your model design as iterations change and evolve it to meet the new behaviors of the stakeholders.

A useful model is arrived at through hundreds of small refactorings. Adjustments to the model are made constantly, with discoveries being unlocked through small transformations.

During development, you may find that your assumption on the core domain may change. The business may discover that it was wrong. Things change. Your boundaries will also change as you realize that new invariants invalidate your design. With new features, you may find that ambiguity creeps in. If it does, split the model and explicitly define the two parts in specific contexts.

Remember to start simple, and move toward the principles and practices when you absolutely need them. Otherwise, you may very well overcomplicate a simple solution to a simple problem.

EXPLORATION AND EXPERIMENTATION

A rich and useful model is a product of exploration and creativity. Experimentation is about looking at the code in a different way. If you find that coding is hard, you are probably doing something wrong. Don't just stop at the first useful model you produce. Because a model evolves over a number of iterations, it is a good idea to delay refactoring until you know enough about the domain. Let the model live a little and evolve. You won't get it right the first time. Experimentation and exploration fuel learning.

Challenge Your Assumptions

During every iteration, the development team must challenge its assumptions because additional features may mean that a previously useful model is no longer fit for purpose. This skill enables your software to be flexible and evolve as the product evolves, thus preventing it from turning into a Big Ball of Mud (BBoM).

A team should not become too attached to a model based on the requirements from a first iteration. Subsequent iterations may see the model become inadequate for the new feature requests. Teams that are inflexible about evolving will soon find that the code works against them. When teams follow a test-driven development methodology, they do not stop when the system is working. Instead, they make a test pass and then refactor their design to make it more expressive. This is by far the most important aspect of test-driven development and one that should be applied to DDD. Refactoring when knowledge is gained helps to produce a model that is more expressive and revealing.

Modeling Is a Continuous Activity

The activity of modeling happens whenever you need it; it is not a step in a project methodology. You break out and collaborate with domain experts whenever it is required. If the area you are working on is well understood, you may find that you don't need to model at all. Don't get too attached to your software; be prepared to throw away your best code when the business model changes. With each iteration comes a new challenge. You need to ensure that you refine and reshape your model to meet the needs of new features and scenarios.

There Are No Wrong Models

There is no such thing as a stupid question or a stupid idea. Wrong models help to validate useful ones, and the process of creating them aids learning. When working in a complex or core area of a

product, teams must be prepared to look at things in a different way, take risks, and not be afraid of turning problems on their head.

For a complex core domain, a team should produce at least three models to give itself a chance at producing something useful. Teams that are not failing often enough and that are not producing many ideas are probably not trying hard enough. When deep in conversation with a domain expert, a team should not stop brainstorming at the first sign of something useful. Once the team gets to a good place, it should wipe the whiteboard and start again from a different angle and try the what-if route of investigation. When a team is in the zone with an expert, it should stay there until it exhausts all its ideas.

LEARN TO UNLEARN

Don't get attached to ideas; trial and error is required to reveal concepts in the domain that will help you solve business problems. Code within the testing namespace alongside the tests until you are happy with the design; you will be a lot happier to spike solutions and throw away a useless model that you haven't committed to the application namespace.

Supple Code Aids Discovery

In Parts II, III, and IV of this book, you will learn about patterns to organize your codebase to enable it to change more effectively with new requirements. This supple code is derived from iterations of small refactors. Constantly refactoring toward deeper insight helps lead to a supple design and flexible code that is able to facilitate change and adapt to new features of the system as they are added in each iteration. If a model is not supple, it is not useful. Martin Fowler states an important modeling principle in his book *Analysis Patterns: Reusable Object Models*: "Design a model so that the most frequent modification of the model causes changes to the least number of types."

However, be careful of premature refactoring. Don't refactor until you know enough about the domain, and don't become preoccupied with applying design patterns. Delaying refactoring can also reveal which areas of the code change most often and why. With this knowledge, you can make more informed design changes to your codebase.

MAKING THE IMPLICIT EXPLICIT

When teams are working deep within a codebase, they often ignore or dismiss logic statements as simple artifacts of programming. These small implicit code blocks hide important details about the domain, often disguising their importance. If these design decisions are not made explicit, they cannot be added to the mental model, and further design discoveries will be harder.

As mentioned previously, delaying refactoring can reveal important details in the code and thus important details of the model. If you find a code grouping that represents some kind of domain logic that doesn't have an explicit name, inform the domain expert, name the logic concept, and wrap the code in the concept. It is vital to make implicit concepts explicit. Any decisions you make in code need to be explicitly fed back to the domain expert and captured as a concept of the model.

Tackling Ambiguity

It's often the things the domain experts don't say or barely hint at that are the key to unlocking deep discoveries within a model. These implicit concepts that may not appear important to domain experts should be made explicit, be named, and be fully understood by the development team. For example, consider an e-commerce site that prevents overseas customers from adding more than 50 of any one item to their basket. Developers can easily implement this business rule, as can be seen in Listing 10-1.

LISTING 10-1: IMPLICIT LOGIC IN CODE

```
public class basket
{
    private BasketItems _items;

    // ....

    public void add(Product product)
    {
        if (basket_contains_an_item_for(product))
        {
            var item_quantity = get_item_for(product).quantity()
                                                .add(new Quantity(1));

            if (item_quantity.contains_more_than(new Quantity(50)))
                throw new ApplicationException(
                        "You can only purchase 50 of a single product.");
            else
                get_item_for(product).increase_item_quantity_by(
                                                        new Quantity(1));
        }
        else
            _items.Add(BasketItemFactory.create_item_for(product, this));
    }
}
```

However, in future months, other developers may not understand why such a rule exists. Instead, you should understand why the rule exists and name the portion of code accordingly. As it transpires, suppliers enforce such a rule to prevent sites acting as wholesalers. With this knowledge, you can make the code explicitly reveal this rule by wrapping it in a class that indicates a deeper insight and understanding of the domain. This refactoring is seen in Listing 10-2.

LISTING 10-2: EXPLICIT LOGIC IN CODE

```
public class basket
{
    private BasketItems _items;

    // ....

    public void add(Product product)
```

```
    {
        if (basket_contains_an_item_for(product))
        {
            var item_quantity = get_item_for(product).quantity()
                                                .add(new Quantity(1));

            if (_over_seas_selling_policy.is_satisfied_by(item_quantity))
                get_item_for(product).increase_item_quantity_by(
                                                        new Quantity(1));
            else
                throw new OverSeasSellingPolicyException(
                    string.format(
                    "You can only purchase {0} of a single product.",
                    OverSeasSellingPolicy.quantity_threshold));
        }
        else
            _items.Add(BasketItemFactory.create_item_for(product, this));
    }
}

public class OverSeasSellingPolicy
{
    public static Quantity quantity_threshold = new Quantity(50);

    public bool is_satisfied_by(Quantity item_quantity, Country country)
    {
        if (item_quantity.contains_more_than(quantity_threshold))
            return false;
        else
            return true;
    }
}
```

Developers should watch for ambiguity or inconsistency in code or in the language of a domain expert. You should also ensure that you pay particular attention to the language of other team members when discussing the domain model. Always validate assumptions about the language and details of the model by talking to domain experts. Validate aloud, and confirm your language and design decisions with linguistic consistency. If a domain expert doesn't say it, it shouldn't be in the language or codebase. If a term in the model no longer makes sense or is not useful, remove it. Keep your language small and focused. Domain experts should object to inadequate terms or structure in the language or model.

Give Things a Name

If domain experts talk about it, make it explicit. If domain experts hint at a concept, make it explicit. If you talk about something that puzzles domain experts, maybe you have misunderstood something they have said and you need to work on your UL. Give things a name, and if you can't think of a good name, defer it and call it the blue policy until you can think of something more meaningful.

A domain model should communicate the intent of the business. Ensure that you take care in naming all methods and properties of your classes. Try to describe the behaviors by involving the

UL. Don't leave your code design up to interpretation. Help yourself and other developers by writing code that is insightful, revealing the rich language of the domain.

A PROBLEM SOLVER FIRST, A TECHNOLOGIST SECOND

A software developer is primarily a problem solver who utilizes technology to implement a solution. Developers are fantastic at educating themselves on technology and project methodologies; however, decomposing a problem and being able to distill what is important from what is not will enable a good developer to become a great one. You should spend as much time in the problem space as you do in the solution space.

Just as a useful model is derived over a series of iterations, so too must a problem space be refined to reveal the true intent behind a stakeholder's vision. Listening and drawing the why as well as the what and when from stakeholders is a skill that developers should practice just as they practice coding katas.

Code is a product of DDD, not the process; you can solve problems without having a technical solution.

Don't Solve All the Problems

All problems are not created equal; some are complex and are of little business value, so it makes no sense to waste effort in finding automated solutions for them. Complex edge cases do not always need automated solutions. Humans can manage by exception. If a problem is complex and forms an edge case, speak to your stakeholder and domain expert about the value of automating it. Your effort could be better served elsewhere, and a human might better handle this exception.

HOW DO I KNOW THAT I AM DOING IT RIGHT?

Unlike becoming a scrum master, there is no certificate awarded when applying the principles and practices of DDD. Your reward from your investment will result in a product that is easily understood, straightforward to maintain, meets the expectations of your stakeholders, and is fun to work on.

You will also find that your team members understand the business better. You will notice that they will be able to talk more fluently with stakeholders and offer solutions to problems that the business didn't know it had or maybe did not have a solution for.

Aligning a team and a business ensures everyone in the organization understands what value means. Teams will no longer spend time on overcomplicated, technical, influenced solutions that use the same architecture and effort, striving for code perfection even for areas that are of little importance to the business. They will instead be able to decompose problems and work with the business to focus on value and spend time in this area, proving good enough, simple solutions to any supporting or generic domains. They will understand where the true value is and where they can make a difference.

Teams will focus on the problem domain, understanding it rather than focusing only on the technical solution. They will spend increased time on the what, why, and when, leaving the how to later.

Good Is Good Enough

Teams that are aligned with the philosophy of DDD focus more on the bigger picture and understand where to put the most effort. They will not apply the same architecture to all parts of a solution, and they will not strive for perfection in areas of little value. They will trade isolated and working software for unnecessary elegance and gold plating.

Only the core domains need to be elegant due to complexity or importance. This is not to say that all other code should be poorly written, but it should be isolated, defined by a boundary, and expose behavior to support the core domain.

Practice, Practice, Practice

Software development is a learning process, and so is DDD. If you want to be good at anything, you need to practice, practice, practice. If you want to be a great developer rather than a good one, you need to show passion for the problem and passion for the solution. To apply the principles of DDD, you need a driven and committed team—a team committed to learning about its craft and the problem domains it works in. Passion lies within all of us, and if you feel value in the practices of DDD, it is up to you to inspire your team and become an evangelist. Passion is contagious; if you commit to spend time with your domain experts to understand a domain at a deeper level and can show how this results in a more expressive codebase then your team will follow.

Many developers are turned off when working in a brownfield environment because of fear of having to work on another developer's codebase. When working on enterprise systems, you have to integrate or work on brownfield environments at some point. Great developers excel at introducing new features into an existing codebase in a safe and maintainable manner.

THE SALIENT POINTS

- Don't sell DDD as a silver bullet. Focus on the alignment with the business and learning more about the domain you are building software for.
- Apply the principles of DDD only when they are needed. Don't use them as a tool for all problems.
- Decompose the problem space and focus on the core domain. All interesting conversations will happen here. This is where you apply the principles of DDD to maximize value and where you should apply the most effort.
- Before modeling a solution, capture the reality of the landscape, and understand other models and contexts in play. Who owns these? What relationships do you have with them? What and how is data exchanged?
- Build a model to satisfy feature scenarios. Start with the most risky or complex. Utilize your domain expert's time here, and don't bother him with simple data management.
- If you are working in a legacy environment, ensure that you protect yourself from external code, don't trust anyone, and enforce your borders. Carve out an area to add new functionality. Don't try to clean up everything.

- Constantly integrate, refine, and challenge your model. Don't stop at your first good idea. Explore and experiment, and validate good ideas by trying new models and solutions. Have at least three useful models.
- Don't assume anything, keep things simple, delay large design decisions, and wait for complexity or new behaviors to challenge your solution. Then refactor toward strategic patterns when you need to.
- Modeling is a team activity, and one that should happen whenever the team is stuck, encounters an area it is unsure of, or needs clarification. It should not be confined to a predefined step in a project time line.
- The model and the language evolve together. A model that cannot be communicated and talked about with ease will have limited usefulness and will be hard to evolve.

PART II

Strategic Patterns: Communicating Between Bounded Contexts

- **CHAPTER 11:** Introduction to Bounded Context Integration
- **CHAPTER 12:** Integrating via Messaging
- **CHAPTER 13:** Integrating via HTTP with RPC and REST

11

Introduction to Bounded Context Integration

WHAT'S IN THIS CHAPTER?

- How to integrate bounded contexts that form a distributed system
- Fundamental challenges inherent to building distributed systems
- Understanding how the principles of Service Oriented Architecture (SOA) can help to build loosely coupled bounded contexts and independent teams
- Addressing nonfunctional requirements while keeping an explicit event-driven domain model using reactive DDD

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 11 download and individually named according to the names throughout the chapter.

After identifying the bounded contexts in your system (as discussed in Chapter 6, "Maintaining the Integrity of Domain Models with Bounded Contexts," and Chapter 7, "Context Mapping"), the next step is to decide how you will integrate them to carry out full business use cases. One of the big challenges you face in this process is successfully designing a robust distributed system. For example, each step of placing an order, billing the customer, and arranging shipping may belong to a different bounded context running as a separate piece of software on a separate physical machine or cloud instance. In this chapter, you learn about fundamental concepts in distributed computing that allow you to retain explicit domain concepts while gracefully dealing with nonfunctional requirements, such as scalability and reliability, that are inherent to distributed systems.

Technical challenges are only one part of integrating bounded contexts and building distributed systems; social challenges are the other. Distributed systems are often too large for a single team to maintain responsibility of, requiring a number of teams to take ownership of one or more bounded contexts. This chapter introduces you to teamwork and communication patterns that successful teams use to build highly scalable systems using Domain-Driven Design (DDD). One common pattern is the adoption of Service Oriented Architecture (SOA).

SOA is an architectural style for building business-oriented, loosely coupled software services. This chapter shows you that, by conceptualizing bounded contexts as SOA services, you can use the principles of SOA to create loosely coupled, bounded contexts that help solve the technical and social challenges of bounded context integration. You also learn about the benefits event-driven reactive programming provides and how it synergizes with DDD by modeling communication between bounded contexts as events that occur in the domain.

Event-driven systems also bring challenges. Most notably they require developers to think differently about how they design their systems, and also give rise to eventual consistency. So this chapter also discusses the drawbacks and options for dealing with them. In addition this chapter also touches on operational concerns like monitoring service level agreements (SLAs) and errors.

After reading this chapter that lays the foundation of bounded context integration, the next chapters provide concrete coding examples of building systems that integrate bounded contexts by applying these concepts. After completing this and the next two chapters, you will be ready to start applying these concepts and your new technical skills to build event-driven distributed systems synergistically with DDD.

HOW TO INTEGRATE BOUNDED CONTEXTS

Software services need to have relationships with each other to provide advanced behaviors. It is your responsibility to choose these relationships and the methods of communication. This massive responsibility can have significant impacts on the speed of delivery, the efficiency, and the success of a project. You may need to integrate with an external payment provider, or you may need to communicate with a system written by another team in your company. In fact, you probably have a number of internal and external relationships like this on most projects.

When you choose relationships between services that reflect your domain, you get the familiar benefits of DDD, such as an explicit model that facilitates conversations with domain experts, allowing new concepts to be incorporated smoothly. Many organizations find that judicious choice of boundaries and communication protocols allows each team to work independently without hindering others.

The choice of communication method alone can be the difference between having an application that scales up to ten million users during periods of heavy viral growth, and a system that collapses in the same scenario taking the whole business down with it. Choosing the communication method is often easier once you've identified the relationship. A good place to start identifying relationships is by identifying your bounded contexts.

Bounded Contexts Are Autonomous

As systems grow, dependencies become more significant in a negative way. You should strive to avoid most forms of coupling unless you have a very good reason. A coupling on code means that one team can break another team's code or cause bottlenecks that slow down the delivery of new features. A runtime coupling between subsystems means that one system cannot function without the other.

If you design loosely coupled bounded contexts that limit dependencies, each bounded context can be developed in isolation. Its codebase can be evolved without fear of breaking behavior in another bounded context, and its developers do not have to wait for developers in other bounded contexts to carry out some work or approve a change.

Ultimately, when bounded contexts are loosely coupled there are likely to be fewer bottlenecks and a higher probability that business value will get created faster and more efficiently.

The Challenges of Integrating Bounded Contexts at the Code Level

You learned in Chapter 6 that bounded contexts represent discrete business capabilities, like sales or shipping. Just as you might walk through the corridors of the company you work for and see signs identifying each part of the business, it is good practice to partition your software systems in line with these business capabilities.

When you're looking at the shipping code, you're focused on the shipping part of the business. It's not that helpful if you have concepts from the Sales Department getting in the way of adding a new feature that integrates a new shipping provider. In fact, making changes to the shipping code might break sales features. If you've heard of the Single Responsibility Principle (SRP), this will make perfect sense to you. In DDD, you can use the SRP to isolate separate business capabilities into separate bounded contexts. It is sometimes acceptable for your bounded contexts to live as separate modules/projects inside the same solution, though.

Multiple Bounded Contexts Exist within a Solution

Once you have identified bounded contexts, it's useful to remind everyone building the system that there is a bigger picture. By putting all of the bounded contexts inside a single code repository or solution it can help developers to see that there is a world outside of their bounded context. Understanding the bigger picture is important because bounded contexts combine to carry out full business use cases.

It is not strictly necessary for each bounded context to live inside the same source code repository or solution (e.g., a Visual Studio solution), though. In some cases, it is not possible because bounded contexts are written in different programming languages that do not even run on the same operating system.

There is no single correct answer that determines where the code for your bounded contexts should live. You need to assess the trade-offs and decide which approach is best for you.

Namespaces or Projects to Keep Bounded Contexts Separate

If you do choose to locate all of your bounded contexts inside a single solution, there is an increased risk of creating dependencies between them. If two bounded contexts use code from another project inside the same solution, there is a dangerous risk of one bounded context breaking the other. Imagine that you have a `User` class in a shared project called `Eccommerce.Common` or similar:

```
public class User
{
    public String Name {get; set;}

    public String Id {get; set;}

    public void UpdateAddress(Address newAddress)
    {
        ...
    }
}
```

If the Shipping bounded context decides to change the implementation of `UpdateAddress()`, it might break the Sales bounded context, which relied on the old address being persisted in a certain location or format. Also, the teams that rely on the shared code might need to have meetings to decide how it can be changed and when they will be able to update their code to accommodate the change. This kind of dependency between teams can slow down a project and introduce undesirable political scenarios. Figure 11-1 visualizes a single solution that contains multiple independent bounded contexts.

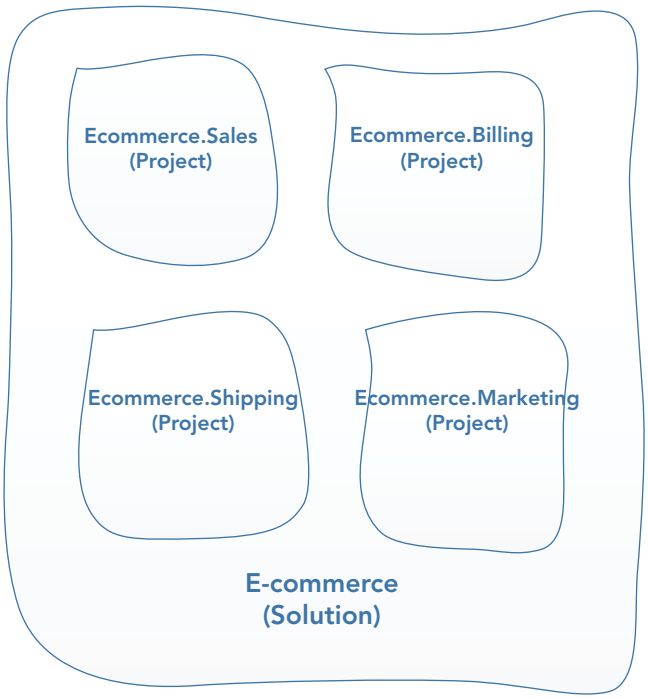


FIGURE 11-1: Multiple bounded contexts inside a single solution.

Integrating via the Database

Another common dependency that slows down teams is the database. For example, the Sales bounded context's team wants to update the `User` schema, but no one is sure if this will break code in the Shipping bounded context or the Billing bounded context. It's likely that several teams will be distracted from working on business priorities to support the Sales bounded context's schema change. A dependency between teams is undesirable because the rate of delivering new features is reduced while the teams synchronize.

If you've used database integration in the past you may also be familiar with another common problem, where each model that integrates through the database has similar but distinct domain concepts. This strategy becomes painful when multiple models use the same shared schema. For example, in the domain of furniture, suppliers may sell mass-produced furniture and custom-made furniture. These two parts of the business are likely to have many differences and are likely to be separate bounded contexts. But if they reside in the same codebase, there is the likelihood that the similarities of each model will result in the use of a shared schema for all types of furniture, as shown in Figure 11-2.

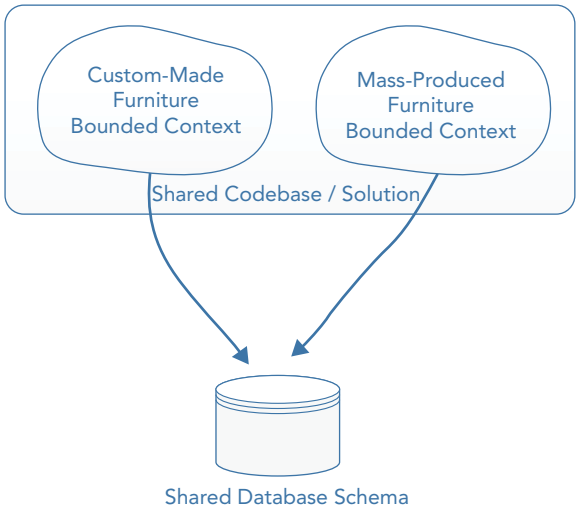


FIGURE 11-2: Multiple bounded contexts using a shared schema.

Initially, the two models of the custom-made and mass-produced bounded contexts may easily map onto the shared schema—but they are likely to diverge in the future. When differences between the models start to appear, the shared schema may have columns relevant to one model but not the other. For example, the custom-made furniture model may require a new `manufacturing_priority` column that has no relevance to the mass-produced context. Yet the shared schema will need to include the new field. In some scenarios columns may even be used for different purposes by each model. Creating reports from a shared schema that is used by multiple bounded contexts can be error-prone, ambiguous, or misleading to the business.

Fundamentally, sharing a schema between models with different semantics can be an expensive violation of the SRP. When the codebase is small the consequences are less severe. But as the system grows, the pain is likely to increase exponentially as each bounded context pulls in different directions.

Multiple Teams Working in a Single Codebase

If you split your domain into multiple bounded contexts that each has its own codebase you will preclude a whole category of organizational problems. When all of your developers are working on a single large codebase you will have more work in progress (WIP) on a single codebase than if you had a greater number of smaller codebases (each with a small amount of WIP). Excess WIP is a prolific, and usually unnoticed, source of inefficiency in the software industry.

WIP is a problem when you want to release new features. How can you release one completed feature if other features are still in progress? Many teams turn to feature branches; when a piece of work is complete, it is then merged into the master branch and released. But this means that you have long-lived branches. If you try to merge a branch after working on it for two weeks, you may be so far behind that the merge may not even be possible. Essentially you lose all the benefits of continuous integration. And you can spend as much time fighting merges and releasing code as you spend on writing it.

In a large domain you can easily have 10 or more developers all working on separate features. They will make great progress in adding the new features, but as-mentioned, trying to merge and release the code can be excruciatingly painful. Consequently deploys are likely to be more risky, needing more QA and manual regression testing. In the worst case, deploys can take a whole day of the company's time where no new value is created.

When many companies are now using continuous delivery and deploying value to their customers multiple times per day, it makes it painfully clear that a single codebase shared by multiple teams can result in heavy costs to the business. A single monolithic codebase is also the kind that is no fun to work on.

Models Blur

If you have a complex domain that effectively has multiple bounded contexts, but you have only one codebase, it is inevitable that boundaries of each model will not remain intact. Code from one bounded context will become coupled to code in another, leading to tight dependencies. As previously mentioned, once you introduce dependencies between bounded contexts you introduce friction that stops them evolving independently and each team working optimally.

Alternatively, if you have separate projects or modules for each bounded context, you remove the possibility of coupling. You might have code in two bounded contexts that look very similar, and you may feel you are violating the Don't Repeat Yourself (DRY) principle, but a lot of the time that is not a problem. Very often you will find that even though the code looks the same to begin with, it changes in each bounded context for different reasons as new concepts and insights emerge. By not coupling the bounded concepts, there is no friction when you try to incorporate the new concepts and insights.

Even if the code is the same and it never changes, usually the duplication causes no problem. Duplication is a problem because you may update code in one place, and forget to update it in another. However, this is rarely a problem when you have loosely coupled, bounded contexts that are intended to run in isolation. There are very few reasons that the same concept in two bounded contexts should be changed at the same time. So don't worry about duplicating similar code, and instead focus on isolating your bounded context and maintaining their boundaries.

Use Physical Boundaries to Enforce Clean Models

To retain the integrity of your bounded contexts and ensure they are autonomous, the most widely used approach is to use a shared-nothing architecture, where each bounded context has its own codebases, datastores, and team of developers. When done correctly, this approach results in a system composed of verticals, as demonstrated in Figure 11-3.

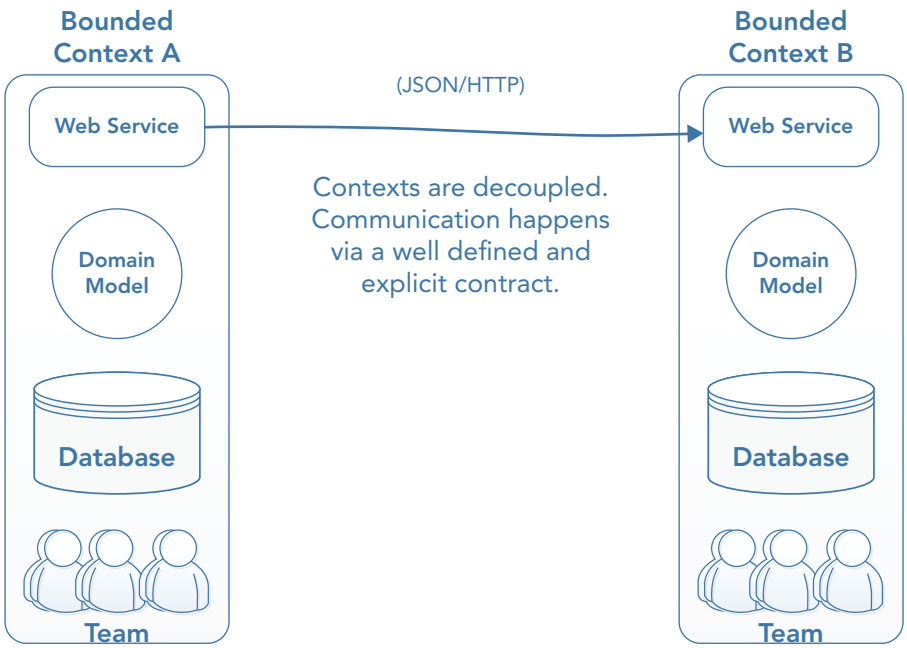


FIGURE 11-3: Autonomous bounded contexts with a shared-nothing architecture.

When each bounded context is physically isolated, no longer can developers in one bounded context call methods on another, or store data in a shared schema. Since there is a distinct physical separation, they have to go out of their way to introduce coupling. More than likely, they will be put off by the extra effort or brought up to speed by another team member before they can create an unnecessary dependency.

Once bounded contexts are clearly isolated and the potential for coupling is significantly reduced, other external factors are unlikely to influence the model. For example, if a concept is added or refined in one bounded context it will not affect concepts that look similar in other bounded contexts, which could easily get caught up in the changes had there been a single codebase. Fundamentally, the clear physical separation allows each bounded context to evolve only for internal reasons, resulting in an uncompromised domain model and more efficient delivery of business value in the short and long term.

Integrating with Legacy Systems

When you are faced with the constraints of integrating bounded contexts that are comprised of legacy code, there are a number of patterns you can use to limit the impact of the legacy on other parts of the system. These patterns help you manage the complexity and save you from having to reduce the explicitness of your new code in order to integrate the legacy components.

> **NOTE** *The patterns in this section are provided by Eric Evans in his paper: "Getting Started with DDD When Surrounded By Legacy Systems," accessible at:* `http://domainlanguage.com/ddd/strategy/GettingStartedWithDDDWhenSurroundedByLegacySystemsV1.pdf`.

Bubble Context

Teams that are unfamiliar with DDD but want to begin applying it to a legacy system are advised to consider using a bubble context. Because bubble contexts are isolated from existing codebases, they provide a clean slate for creating and evolving a domain model. Remember, DDD works best when you have full control over the domain model and are free to frequently iterate on it as you gain new domain insights. A bubble context facilitates frequent iteration even when legacy code is involved.

For bubble contexts to be effective, a translation layer is necessary between the bubble and the legacy model(s). The DDD concept of an anti-corruption layer (ACL) is ideal for this need, as shown in Figure 11-4.

Design and implementation of the ACL is a key activity when building a bubble context. It needs to keep details of the legacy system completely isolated from the bubble while at the same time accurately translating queries and commands from the bubble into queries and commands in the legacy model. It then has to map the response from the legacy into the format demanded by the bubble. Accordingly, the ACL itself can be a complex component that requires a lot of continued investment.

Autonomous Bubble Context

If you want to integrate with legacy code, but do not want to create a bubble context that is so dependent on the legacy code, you can instead use an autonomous bubble. Whereas a bubble context

gets all its data from the legacy system, an autonomous bubble is more independent—having its own datastore(s) and being able to run in isolation of the legacy code or other bounded contexts, as shown in Figure 11-5.

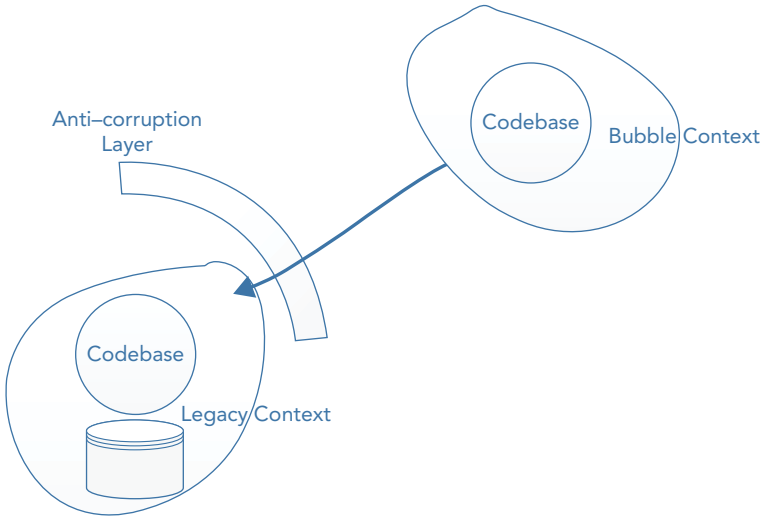


FIGURE 11-4: A bubble context.

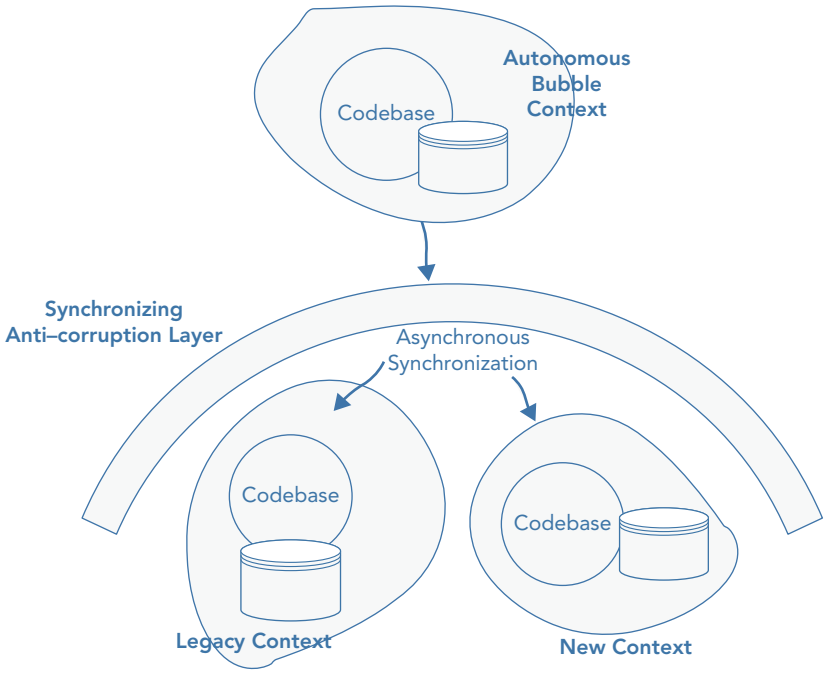


FIGURE 11-5: An autonomous bubble context.

Sometimes crucial to the autonomous bubble context's independence is asynchronous communication with other new and legacy contexts. Consequently, the ACL—a synchronizing ACL—often takes on the role of carrying out the asynchronous communication, as also illustrated in Figure 11-5.

Since the autonomous bubble context has its own datastore it does not require updating legacy codebases or schemas. Any new data can be stored in the autonomous bubble's datastore. This is an important characteristic to keep in mind when deciding between the bubble and the autonomous bubble. However, the costs and complexity of asynchronous synchronization can be significantly higher.

> **NOTE** *In the remainder of Part II you will learn more about the concepts of asynchronous communication, including message bus and HTTP-based examples.*

Exposing Legacy Systems as Services

When a legacy system needs to be consumed by multiple new contexts the cost of creating a dedicated ACL for each context can be excessive. Instead, you can expose the legacy context as a service that requires less translation by the new contexts. A common, and often simple approach, is to expose an HTTP API that returns JSON, as shown in Figure 11-6. This is formally known as the Open Host pattern.

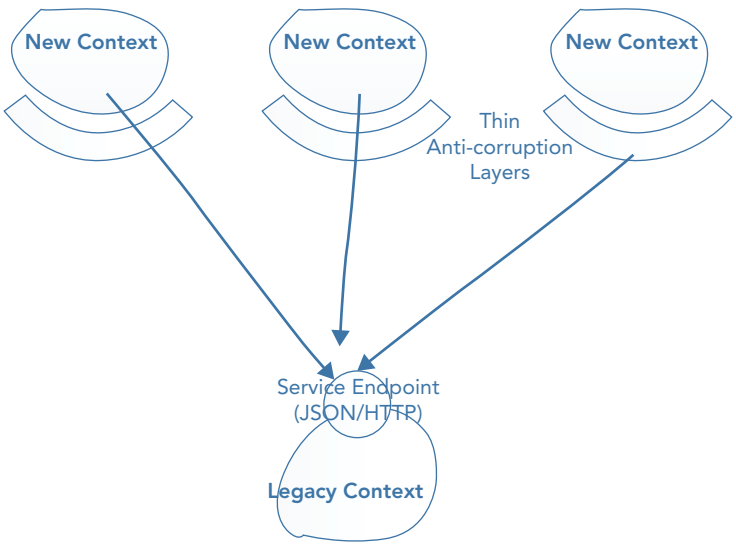


FIGURE 11-6: Exposing a legacy context as a JSON web service.

Each consuming context must still translate the response from the legacy context into its own internal model. However, the translation complexity in this scenario should be mitigated by the simplicity of the API provided by the open host.

While exposing legacy contexts as services can be more efficient when there are multiple consumers, both of the bubble approaches should still be considered. Two of the main drawbacks in exposing legacy contexts as services are that, first, modifications are required to the legacy context that may not be necessary with a bubble. Second, exposing a format that is easily consumable by multiple consumers may be challenging.

Admittedly, choosing a legacy integration strategy can be confusing. However, with three approaches to choose from, you can see that harnessing the benefits of DDD on legacy systems is definitely possible.

INTEGRATING DISTRIBUTED BOUNDED CONTEXTS

The phenomenal growth in the number of people using the Internet means that many applications today need to be able to support huge levels of web traffic. If the number of users cannot be supported, a business will not maximize its revenue potential. Primarily, the problem is a hardware one: A single affordable server is usually not powerful enough to support all the users that a popular website may have. Instead, the load needs to be spread across multiple machines. And it's not just websites, it's all of the back end of services that make up a system.

Spreading load across multiple machines is such a common problem that it has given rise to the cloud boom. Businesses are using cloud-hosted solutions to rapidly scale their system's price efficiently. If you design your systems so that they can be spread across multiple machines, you can take advantage of cloud hosting to efficiently scale your systems.

Another key reason that modern systems are distributed is fault tolerance. If one server fails or develops a problem, other servers must be able to take on the increased load to avoid end users suffering.

Having to distribute a system introduces the need to break it up into smaller deployable components. This poses a challenge to maintaining the explicitness that DDD strives for.

Integration Strategies for Distributed Bounded Contexts

Distributed systems, although helping to solve the problems of high scalability, come with their own set of problems. Luckily, you do have some choice about which set of problems you need to face, because there are a number of different integration strategies. Remote procedure call (RPC) and asynchronous messaging are prevalent options and encompass most of this chapter. However, sometimes sharing files or databases can be a good enough alternative.

Distributed systems bring nonfunctional requirements to the table: scalability, availability, reliability. *Scalability* is the ability to be able to support increasing loads, such as more concurrent users. *Availability* is concerned with how often the application is online, running, and supporting its users. Another consideration is *reliability*, which is concerned with how well a system copes with errors. You'll see shortly that these requirements are often traded off with the amount of coupling in a system and the level of complexity.

When integrating your bounded contexts, it's important to get an idea from the business what its nonfunctional requirements are so that you can choose an integration strategy that lets you meet them with the least amount of effort. Some options, such as messaging, take more effort to implement, but they provide a solid foundation for achieving high scalability and reliability. On the other hand, if you don't have such strong scalability requirements, you can integrate bounded contexts with a small initial effort using database integration. You can then get on with shipping other important features sooner.

Unfortunately, you can't just ask the business how scalable a system needs to be or how reliable it should be. Try to give them scenarios and explain the costs of each scenario. For example, you could tell them that you can guarantee 99.9999% reliability, but it will cost triple the amount of guaranteeing 99.99%.

Database Integration

An accessible approach to integrating bounded contexts is letting one application write to a specific database location that another application reads from. It's likely that you would want to employ this approach in first-iteration Minimum Viable Products (MVPs) or nonperformance critical parts of the system.

As an example, when you place an order, the Sales bounded context could add it to the `Sales` table in a SQL database. Later on, the Billing bounded context would come along and verify whether any new records had been added to the table. If it found any, it would then process the payment for each of them.

Implementing this solution has a few possibilities. The most likely example involves the Billing bounded context polling the table at a certain frequency, such as every 5 minutes, and keeping track of which orders it has processed by updating a column in the same row—`paymentProcessed` perhaps. You can see a visual representation of this in Figure 11-7.

Database integration also has some loose coupling benefits. Because both systems communicate through writing and reading to a database, the implementation of each system is free to change providing it maintains compatibility with the existing schema. However, the systems are coupled to the same database, so this style of solution can really start to hurt you as the system grows. Database locks are just one painful problem you might come up against. As increasingly more orders are added to the table by one part of the system and then updated by another, the two systems will be competing for database resources. The database will likely be a single point of failure (SPOF), meaning that if it grinds to a halt, both applications will suffer. Some databases, like SQL Server, are hard to cluster, so you have to keep buying bigger, more expensive hardware to scale.

Another drawback is that database integration doesn't guide you into a good solution for handling faults. What if the Sales bounded context crashes before saving an order? Is the order lost? What if the database goes down? Does that mean the company cannot take orders? These are big problems that you are left on your own to devise solutions for when using database integration.

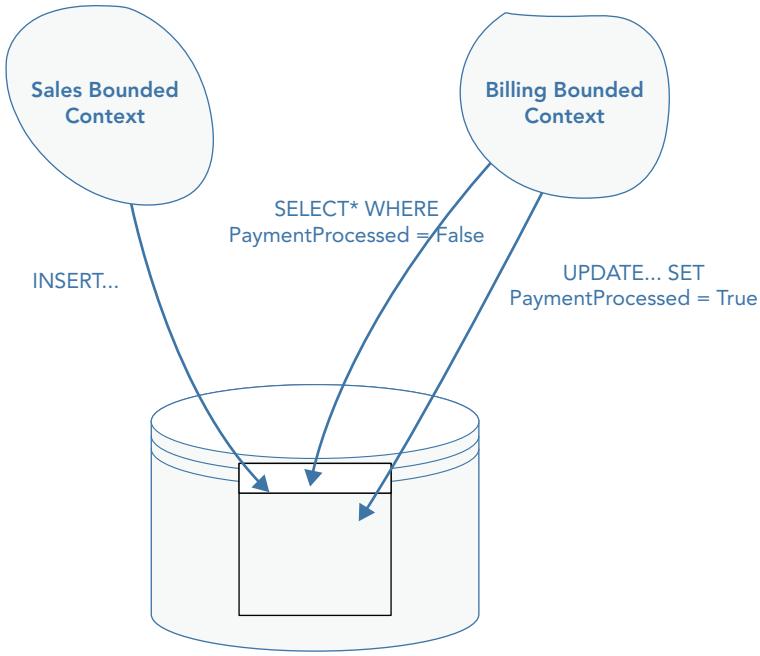


FIGURE 11-7: Database integration.

Flat File Integration

If you aren't using a database in your project, setting one up just to integrate two components can be unnecessary overhead. This is one example in which flat file integration may be good enough. One component puts files on a server somewhere, while another application picks them up later, in a similar way to database integration. Flat file integration is a more flexible approach than database integration, but you have to be more creative, which can in turn mean more effort and slower lead times on important business functionality. In Figure 11-8, you can see one possible implementation of a flat file integration alternative to the database integration solution in Figure 11-8.

Flat file integration retains the loose coupling features of database integration without suffering from the database locking problem. Unfortunately, you have to work harder to compensate for this. One area of compensation involves the file format. Because there is no schema or standard query language, you are responsible for creating your own file format and ensuring that all applications understand it and use it correctly. Because this is more manual work, it does increase the possibility for error, although a lot depends on your circumstances.

Because flat file integration is a do-it-yourself solution, there are no scalability or reliability guidelines. It's completely dependent on the choice of technologies you use and how you implement them. If you do need an approach that scales, going to all that effort can be a massive waste and a

massive risk. RPC, discussed next, is a common alternative that is a relatively well known quantity in terms of scalability and reliability characteristics.

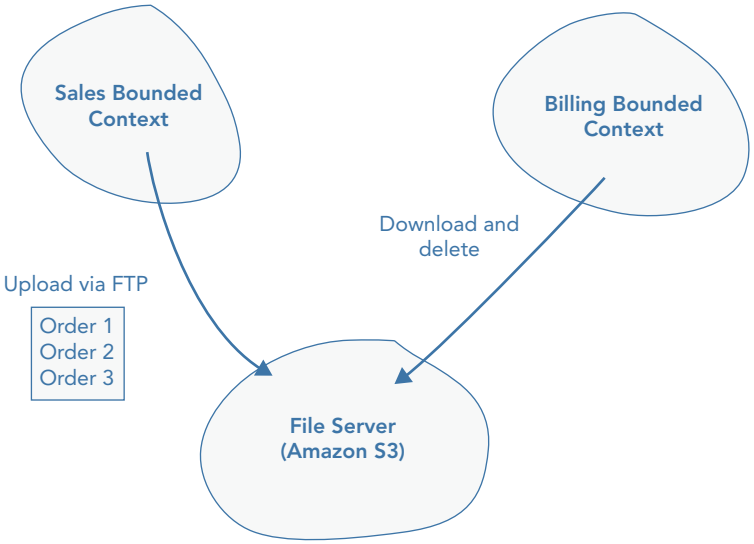


FIGURE 11-8: Flat file integration.

RPC

Imagine if you could keep your `monolithic` almost identical but get the scalability benefits of a distributed system. This is the motivating force behind the use of RPC. When using RPC, any method invocation could call another service across the network; it's not possible to tell unless you look at the implementation. Can you tell how many network calls would occur if you ran the following code snippet?

```
var order = salesBoundedContext.CreateOrder(orderRequest);
var paymentStatus = billingBoundedContext.ProcessPaymentFor(order);
if (paymentStatus.IsSuccessful)
{
    shippingBoundedContext.ArrangeShippingFor(order);
}
```

It's impossible to tell how many network calls occur in the previous code snippet because you don't know what happens inside any of the methods that are called. There could be in-memory logic, or there could be network calls to other applications that carry out the logic. Proponents of RPC see this trait as a huge benefit because it is a minimally invasive solution. In some situations, RPC can be the best choice.

When choosing RPC, you have a lot of freedom, because RPC itself is a concept that you can implement in a variety of ways. If you were to talk to enough companies using RPC, you would find examples using most kinds of web service—Simple Object Access Protocol (SOAP),

REpresentational State Transfer (REST), eXtensible Markup Language (XML)—using a variety of different technologies, such as Windows Communication Foundation (WCF). RPC tends to be easier than flat file integration because most programming communities have frameworks that deal with a lot of the infrastructure for you.

Many distributed systems novices are tempted to use RPC because most of their existing code can be reused. This is a compelling case for choosing RPC, but it is also a drawback. In a later section ("The Challenges of DDD with Distributed Systems"), you learn in some detail why RPC's appeal has some deep flaws for achieving scalability and reliability. That is why there is a need for asynchronous, reactive messaging solutions.

Messaging

Quite simply, networks are unreliable. Even the biggest companies like Netflix and Amazon suffer from network problems that result in system outages (`http://www.thewhir.com/web-hosting-news/netflix-outage-caused-by-ec2-downtime-reports`). Reactive solutions try to embrace failure by increasing reliability using asynchronous messaging patterns for communication. This means that when a message fails, there is a way for the system to detect this and try it again later (such as storing it in a queue) or take a different course of action.

Unfortunately, using messaging usually means that, unlike RPC, your code looks drastically different. When you look at RPC code, there's no hint of the network; when you look at many messaging solutions, it's clear that code is asynchronous, and there's probably a network involved. Not only that, but the entire design and architecture of messaging systems are significantly different, and teams are challenged with an intimidating learning curve. Fortunately, you will learn about messaging in this chapter and the next because, along with RPC, it is the other common option used for building distributed DDD systems. In particular, you will learn that the asynchronous nature of messaging also provides the platform for improved scalability.

REST

If you want the scalability and reliability benefits of messaging solutions, but you want to use Hypertext Transport Protocol (HTTP) instead of messaging frameworks, try REST. REST involves modeling your endpoints as hypermedia-rich resources and using many of the benefits of HTTP, such as its verbs and headers. You can then build event-driven systems on top of HTTP with REST to get many of the benefits of a messaging system, and sometimes fewer of the problems.

REST is also a useful tool for exposing your system as an application programming interface (API) for other applications to integrate with. After learning about distributed systems concepts in this chapter, you will learn more about integrating with REST in Chapter 13, "Integrating via HTTP with RPC and REST."

THE CHALLENGES OF DDD WITH DISTRIBUTED SYSTEMS

When your bounded contexts are separate services that communicate with each other over the network, you have a distributed system. In these systems, choosing the wrong integration strategy might cause slow or unreliable systems that lead to negative business impacts. Development teams

need to understand approaches to building distributed systems that can reduce the potential and severity of these problems and allow a business to scale as demand grows.

Accepting that failures happen, and preparing for them, is a critical aspect of building distributed systems, but one that is not inherent to RPC.

The Problem with RPC

When the time comes to scale your application from a single codebase to a number of smaller subsystems, you might be tempted to replace the implementation of a class with an HTTP call. The old logic then moves to a new subsystem, which is the target of this HTTP call. It's tempting because you can spread the load across two machines without having to change much code. In fact, the code looks the same, as the previous code snippet (repeated below) demonstrates:

```
var order = salesBoundedContext.CreateOrder(orderRequest);
var paymentStatus = billingBoundedContext.ProcessPaymentFor(order);
if (paymentStatus.IsSuccessful)
{
    shippingBoundedContext.ArrangeShippingFor(order);
}
```

Each method call can be processed entirely in memory or can make an HTTP call to another service that carries out the logic. Remember that this is the goal of RPC: to make network communication transparent. Have a look at Figure 11-9 to see an example of an e-commerce system using RPC to place an order.

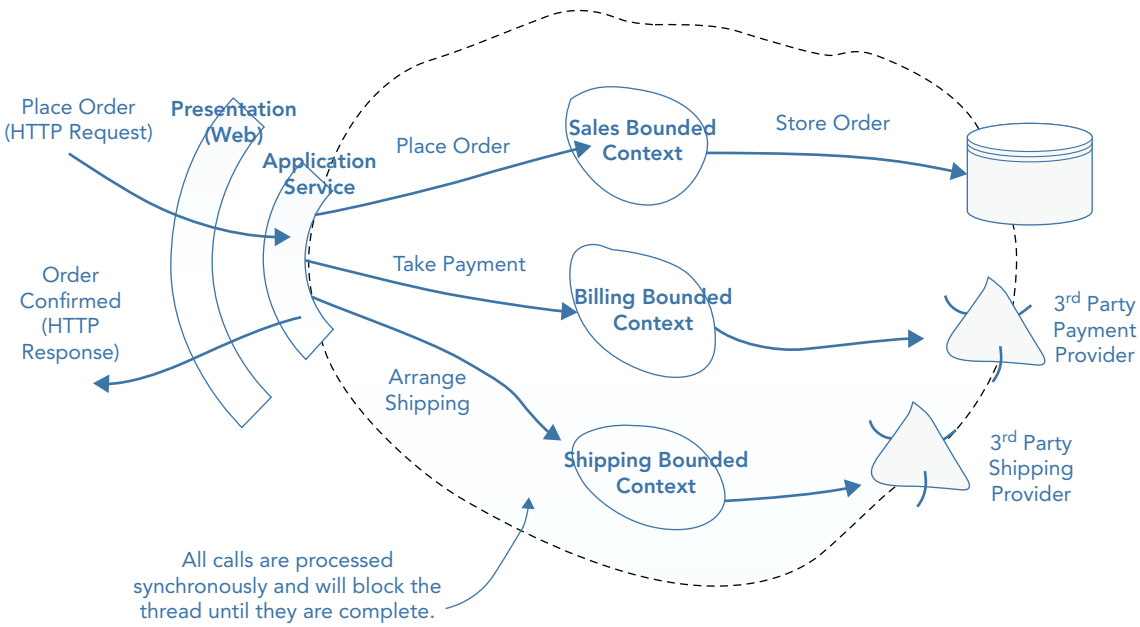


FIGURE 11-9: E-commerce system using synchronous RPC.

Although RPC feels like good use of object-oriented programming and encapsulation, it has some significant flaws that the distributed systems community has known for years. These flaws can easily negate the determined effort you spent frequently collaborating with domain experts and finely crafting your domain models. To save you suffering from the pain caused by RPC, you will now learn its inherent problems before being shown alternative approaches that attempt to address its drawbacks. You will then be able to decide for yourself if RPC or messaging is the best choice for the projects you work on.

> **NOTE** *Strictly speaking, this is an example of synchronous RPC: where the calling code blocks and waits for the result of the RPC. You can emulate RPC with asynchronous calls. This chapter refers only to synchronous RPC, which is usually the case. It's important to be aware of the distinction, though, because the synchronous communication is a fundamental part of the problem.*

RPC Is Harder to Make Resilient

Because RPC makes network communication transparent, it encourages you to forget the network is there. Unfortunately, network errors do happen, meaning that systems using RPC are more likely to be unreliable. Network errors have been such a major source of problems in distributed systems that they feature prominently in the Fallacies of Distributed Computing (`http://blog.newrelic.com/2011/01/06/the-fallacies-of-distributed-computing-reborn-the-cloud-era/`). Essentially, time and again, it has been proven that networks are neither reliable nor free from bandwidth and latency costs that RPC implementations often take for granted.

In an online order processing scenario in which the Billing bounded context makes an HTTP call to a third-party payment provider, if the network is down or the payment provider goes offline, the order cannot be completed. At this point, the potential customer will be unhappy that she can't purchase products, and the business will be even less happy considering it missed out on revenue. Consider the case of a busy Christmas period or a major sporting occasion; a large part of the business model for many companies is maximizing opportunity at these key events. If the system is down, there could be severe consequences for the business. You will see later in this chapter how to avoid these problems, even when major failures happen.

RPC Costs More to Scale

Using the same e-commerce scenario, the scalability limitations of systems that use RPC can be demonstrated. Consider the case in which a business stakeholder sends you an e-mail expressing concern at the number of complaints received from users. You are being told that hundreds of users are reporting a very slow website, while others are reporting that sometimes they completely fail to reach the home page at all because of timeout errors. The problem is a scaling issue because the current system cannot support the current number of concurrent users.

To improve user satisfaction and reduce the number of complaints, the website would need to be faster and able to support more concurrent users. Unfortunately, the Sales, Billing, and Shipping

bounded contexts all have to do some processing before the user gets a response, as illustrated in Figure 11-9. That means all of them need increased resources in order to give the business a faster website. If just the website gets faster servers, the user will still be waiting the same amount of time for each of the bounded contexts to do their work. This is undesirable because it is not easy to just make the website faster.

As before, alternative techniques exist that allow the business to scale just the website, or indeed any other bounded context, in isolation on an efficient, as-needed basis. You will read about them later in this chapter.

RPC Involves Tight Coupling

Systems that use RPC have tightly coupled software components that can lead to couplings between teams as well as technical problems like those you just read about. When a system makes an RPC call to another system, there are actually multiple forms of tight coupling that you really need to be aware of. First, there is a logical coupling, because the logic in the service making the call is coupled to the service receiving the call. Second, there is a temporal coupling, because the service making the call expects a response straight away.

Logical Coupling

Removing dependencies on shared code can still result in coupling. If a service makes a call to another service, the called service has to exist and has to behave in a way that the calling service understands. Therefore, the calling service is logically coupled to the receiving service. The kinds of pain caused by a logical coupling can be similar to a shared-code dependency—changes in one place break functionality in another. Logical couplings can also cause pain when the service being called goes down due to failure, because in those cases neither service is functioning. This hurts reliability.

Temporal Coupling

Performance is a feature. The more performant a website is, the more research indicates that users will be converted into paying customers. If part of your system needs a speed increase to improve some aspect of user experience, it can be difficult if that component relies on another component to do some part of the processing and respond immediately. This is known as temporal coupling, and it is inherent to RPC. Figure 11-9 highlights temporal coupling, showing the user waiting for a response while each bounded context takes its turn to perform some processing. This leads to a component not being able to scale independently.

> **NOTE** *To get an idea of how important performance is, Amazon announced that they saw significant increases in revenue that correlated to performance improvements of just one hundred milliseconds (*`http://glinden.blogspot.co.uk/2006/11/marissa-mayer-at-web-20.html`*). Google also reported similar findings.*

Distributed Transactions Hurt Scalability and Reliability

Transactions are a best practice for maintaining data consistency. Unfortunately, when building distributed systems, transactions carry a high cost because of the involvement of network communication. As a result, scalability and reliability can be negatively impacted for reasons such as excessive long-held database locks or partial failures. Therefore, you should carefully consider distributed transactions, as well as RPC, when building distributed systems.

A typical example of a distributed transaction is booking a holiday that consists of a hotel and a flight, where your system stores hotels but flights are handled by an external system. When you place an order, a lock is put on the hotel room in your database. The lock is then held for a while until the flight is booked. Because it takes on average a few seconds to make the HTTP call over the Internet to the flight company, the number of database locks and connections grows. Once you reach a tipping point with databases, they tend to fall over and reject new connections or worse. As your system starts to scale up, the severity of this issue only increases, meaning that opportunities for revenue loss increase.

Another concern when using distributed transactions is partial availability. What if you lock a hotel database record and find the flight provider is offline? In a distributed transaction, you have a failure, meaning the transaction aborts or rolls back. If there's no specific business requirement for hotels and flights to be booked at the same time, this is again a loss of revenue due to avoidable technical errors.

Bounded Contexts Don't Have to Be Consistent with Each Other

So what do you do instead of using database locks to prevent undesirable scenarios occurring, like booking a hotel room when there is no flight available? A common approach is to just roll forward into a new state that corrects the problem. In the holiday booking scenario, that would mean cancelling the hotel reservation when the flight booking fails. But this would not happen inside a single transaction. What this *does* mean is that your bounded contexts will have inconsistent views of the world. An order might exist in one of them, but not in another. In other parts of the system, a user might have updated his address, but the update hasn't reached other bounded contexts yet. This often isn't ideal, but you have to remember that the more you want to scale, the more you may have to make these kinds of trade-offs.

By avoiding distributed transactions, you can handle partial failures without incurring revenue loss. Once the hotel is booked, for example, it doesn't matter if the flight provider is offline or in some failure mode. You can book the flight when the provider comes back online or is working properly. You'll see examples of this later in this chapter and the next.

Allowing temporary inconsistencies in your system is not a radical approach. It's quite a common concept in distributed systems and goes by the name of eventual consistency.

Eventual Consistency

Although your system may be in inconsistent states, the aim is always to reach consistency at some point for each piece of data. (The system overall will probably never be in a fully consistent state). In

the previous example, perceived consistency was achieved by arranging the flight when the provider came back online. Essentially, this is eventual consistency, but it applies to a much wider range of scenarios.

> **NOTE** *If you prefer a more academic definition and description of eventual consistency, Wikipedia is a good place to start (*`http://en.wikipedia.org/wiki/Eventual_consistency`*).*

One important acronym that goes along with eventual consistency is BASE, which stands for Basically Available, Soft state, Eventual consistency. This is in contrast to ACID (Atomicity, Consistency, Isolation, Durability), which you are probably familiar with from relational databases. These acronyms highlight the fundamental differences in consistency semantics between the two approaches.

In the next chapter you see examples of how to build messaging systems that cater to eventual consistency in a way that tries not to hurt user experience. However, sacrificing user experience can be one of the big drawbacks to using eventual consistency. It's common for websites to allow you to place an order or add some other piece of information yet not have immediate confirmation. This upsets some users, and rightly so in a lot of cases. But there are also many success stories, from large companies like Amazon (`http://cacm.acm.org/magazines/2009/1/15666-eventually-consistent/fulltext`) to much smaller companies. If you do your homework and plan diligently, you give *yourself* a great chance of getting the scalability benefits of eventual consistency while still providing a great user experience.

Aside from the example in the next chapter, you are encouraged to spend a bit of time learning more about eventual consistency before you build a real system that uses it. Pat Helland's influential paper "Life Beyond Distributed Transactions: An Apostate's Opinion" is recommended reading (`http://cs.brown.edu/courses/cs227/archives/2012/papers/weaker/cidr07p15.pdf`). Martin Fowler also has a good piece on not being afraid of living without transactions (`http://martinfowler.com/bliki/Transactionless.html`).

Event-Driven Reactive DDD

For more than 20 years, distributed systems experts have known about the limitations of synchronous RPC and instead preferred asynchronous, event-driven messaging solutions in many situations (`http://armstrongonsoftware.blogspot.co.uk/2008/05/road-we-didnt-go-down.html`). Before delving into the fundamentals of why this approach can be more beneficial, you will see how to use it to remedy the resilience and scalability issues of the previous RPC example.

In this section, you learn how choosing a good integration strategy can have massive scalability and reliability benefits that make a positive business impact. What you are about to see is an alternative solution using the principles of reactive programming

(`http://www.reactivemanifesto.org/`)—a philosophy that replaces RPC with asynchronous messages. Figure 11-10 shows the design of a reactive messaging alternative to the RPC implementation shown in Figure 11-9.

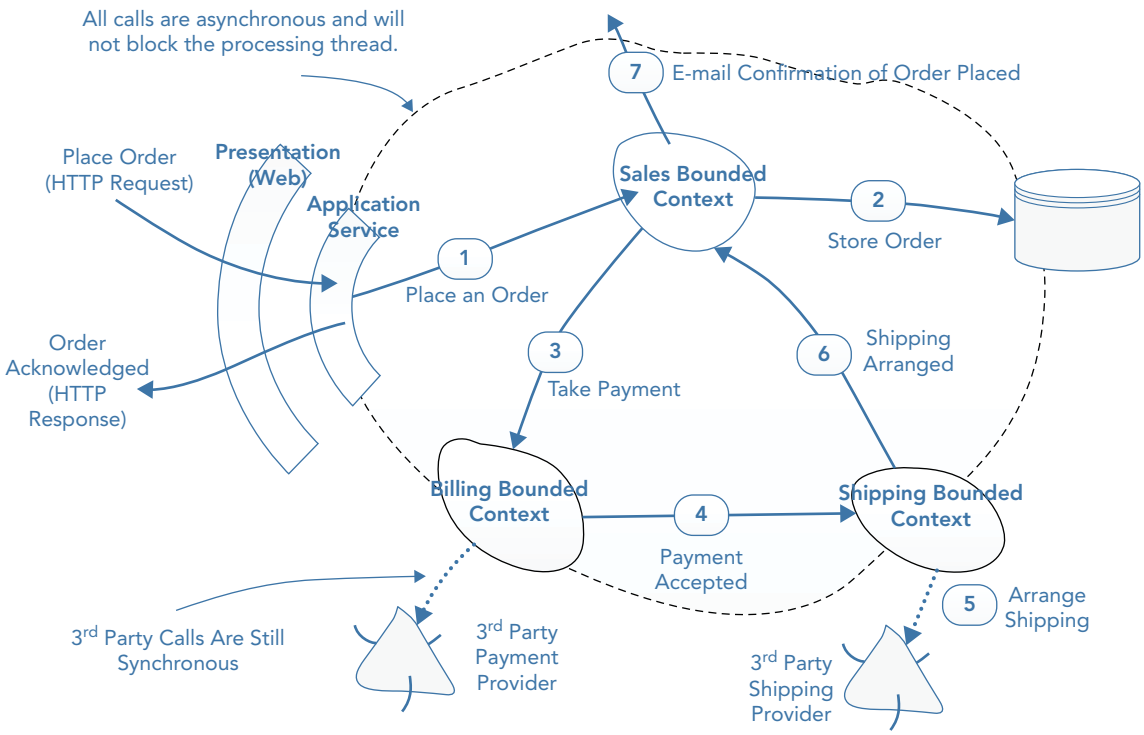


FIGURE 11-10: Replacing RPC with reactive.

> **NOTE** *Asynchronous messaging was around long before reactive programming became popular. In fact, reactive programming was just a name given to asynchronous messaging and other related concepts so that people could easily refer to them in conversation.*

Demonstrating the Resilience and Scalability of Reactive Solutions

In Figure 11-10, you can see that when a user places an order with the website, a message is asynchronously sent to the Billing bounded context indicating that an order has been placed. In turn, the Billing bounded context emits an asynchronous `PaymentAccepted` event after

communicating with the payment provider. The Shipping bounded context subscribes to the `PaymentAccepted` event, arranging shipping for an order when one is received. Meanwhile, the user interacting with the website gets an immediate response as soon as his intent to place an order is captured. He then gets an e-mail later once everything is confirmed. There's no wait for the asynchronous messages to flow through the bounded contexts. The performance problem is solved by processing the order asynchronously, but what about the resiliency problems?

To solve the resiliency problems, each message is put in a queue until the recipient(s) has successfully processed it. Now reconsider the problems that can occur during the processing of someone's intent to place an order. If the payment provider is experiencing problems, the message is stored in a queue and retried when the payment provider is available. If one of the bounded contexts has a hardware failure or contains a programming bug, the message again sits in a queue waiting for the bounded context to come back online. As you can now see, reactive solutions provide the platform for exceptional levels of resilience, particularly compared to RPC alternatives.

> **NOTE** *Most messaging frameworks alleviate the pain of storing messages and carrying out the retry logic, as you will learn in the next chapter.*

To further understand how the reactive solution increases scalability, think back to the case in which users are reporting that the website is unusably slow. You can easily achieve the optimal solution of scaling out by adding more instances of the web application to a load balancer without changes to the hardware used for the bounded contexts. Whenever bottlenecks occur in the system, you have more granular choices to make about where to add new hardware. Ultimately, this means the business makes more cost-effective use of hardware or cloud resources. In the old example, it might not even be possible to scale out each of the bounded contexts because of the way they have been programmed. This means they would have to be scaled up, which can be relatively more expensive.

> **NOTE** Scaling up, *also known as* vertical scaling, *means increasing the performance characteristics of the hardware, such as more RAM or faster servers. Scaling out, also known as horizontal scaling, means distributing the load across more machines.*
>
> *Scaling out is often more cost effective because the cost of adding more hardware remains the same for each new machine. However, scaling up is less cost effective because the cost of purchasing more powerful machines can grow steeply.*

Challenges and Trade-Offs of Asynchronous Messaging

Be careful about forming the impression that reactive programming solves all your problems and makes life easy, because a lot of effort on your part is still required. As with many decisions you need to make in software development, taking the reactive approach has positive and negative consequences that you need to trade off against the constraints you are currently working to. Here are some of the challenges you are likely to face when building reactive applications: increased difficulty in debugging your asynchronous solution, more indirection in your code when others try to understand how it works, and eventual consistency. You also have the added complexity of more infrastructure components that deliver and retry messages.

Don't worry too much about having to learn new techniques, though. Instead, be excited. You will see concrete examples in the next chapter of how infrastructure technologies assist you in debugging asynchronous systems. You will even see that, by having sensible naming and code structure conventions, working out what asynchronous code is doing doesn't have to be such a huge problem. A conceptual framework will also be outlined that puts you well on your way to coming to grips with eventual consistency.

Is RPC Still Relevant?

You should consider RPC as a tool that might be the best choice for certain situations. It poses scalability and reliability problems, but sometimes they might not be the most important constraints. Chapter 13 shows you how to build HTTP-based services for DDD systems that use RPC so that you can apply them in the situations described next.

Time-to-Market Advantages

Sometimes you just need to get a new feature or product out on the Internet for customers to evaluate. The business may want to beat a competitor, or it may just want to evaluate user reaction to the product. In these scenarios, scalability and reliability are not a problem. So if you feel that building an RPC-based solution meets these requirements, you should look for a good reason not to use it. One reason that it may give you a time-to-market advantage is that more developers are familiar with HTTP than they are with messaging frameworks and concepts. Therefore, you can hire people quicker, and they can build the system quicker.

Easier to Hire and Train Developers

It's hard to think of a developer who doesn't know HTTP, whereas those building distributed systems with messaging platforms are a little harder to come by. So when you're hiring people or building a team, you have a bigger pool to choose from if you're building an RPC-based system. If you go down the messaging route, you may need to train people to not only use messaging frameworks, but use the fundamental concepts you've learned about in this chapter so they can design and build messaging systems properly.

Platform Decoupled

A big drawback to using many messaging frameworks is they don't really provide tight integration across different development run times and operating systems. You'll see in the next chapter how

NServiceBus is great for integrating .NET applications, but as soon as you introduce another type of message bus, even one that is designed for .NET, life becomes a bit trickier (although certainly not impossible). Some messaging frameworks do claim to handle cross-platform scenarios, though. So they may be worth investigation if you have bounded contexts running on different platforms.

> **NOTE** *Chapter 13 also shows that you can use event-driven REST if you want the scalability benefits of a messaging system while using HTTP.*

External Integration

You saw in the e-commerce example that you have to communicate with external services like payment providers. This kind of communication across the Internet doesn't use messaging for many reasons. Instead, it uses HTTP. You may want other websites and applications to integrate with your system, perhaps showing your products and services on their website. To do this, you will nearly always provide REST or RPC APIs over HTTP. This ensures that everybody is able to integrate with your APIs because everyone knows and is capable of working with HTTP.

SOA AND REACTIVE DDD

In the previous section, you saw the need to go reactive in scenarios that require scalable, fault-tolerant distributed systems. In this section, you learn how to work down to a structured reactive solution from your bounded contexts using the principles of SOA. In the process, you discover how SOA guides you into having independent teams that can develop their solutions in parallel while still achieving smooth runtime integration.

Underlying SOA is the need to isolate different pieces of software that represent different capabilities of the business, such as billing customers or arranging shipping. In the true SOA sense, services need to be loosely coupled to other services and highly autonomous—able to carry out their specific functionality without help from other services. Loosely coupled services provide a number of technical and business benefits. For a start, the development teams responsible for them can work in parallel with minimal cross-team disruption. You see later in this section how to achieve this but first, you see how loosely coupled services are the ideal stepping stone to go from bounded contexts to integrated distributed systems based on reactive principles.

> **NOTE** *In this section,* loosely coupled *means minimal logical and temporal coupling compared to the original RPC example. However, the term does not have a fixed technical meaning in distributed systems nomenclature.*

View Your Bounded Contexts as SOA Services

If reactive programming is a set of low-level technical guidelines that lead to loosely coupled software components, and SOA is a high-level concept that facilitates loosely coupled business capabilities, then the combination appears perfect for creating business-oriented, scalable, resilient distributed systems. The missing link so far is how to combine these benefits with DDD. One answer is to view your bounded contexts as SOA services so that you can map high-level bounded contexts onto low-level, event-driven software components. Now you have the potential for the business alignment benefits of SOA *and* scalability/resiliency benefits of reactive programming. Full examples of implementing the concepts introduced in this section are provided in the next chapter.

DISAMBIGUATING SOA

SOA has become an ambiguous term in software development. Netflix has one definition that uses very small services (`http://techblog.netflix.com/2012/06/netflix-operations-part-i-going.html`), whereas many companies have fewer larger services. There is also some debate about communication protocols. Although a message bus is often used for SOA, it is not a requirement. Arnon Rotem-Gal-Oz has an excellent piece that goes into great detail explaining what is and isn't SOA (`http://www.manning.com/rotem/SOAp_SampleCh01.pdf`).

One important distinction to keep in mind is that SOA does not merely mean web services. At its heart, though, SOA is about loose coupling to provide business and technical benefits.

Decompose Bounded Contexts into Business Components

Inside a bounded context you may have a number of responsibilities. In a Shipping bounded context, for example, there may be logic to deal with priority orders and standard orders. By isolating each major responsibility as a component, you'll find yourself having clearer conversations with the business and all the other benefits of DDD that you've already seen arise from making the implicit explicit. Additionally, you'll have increased clarity in your codebase by having two separate modules, each with a single responsibility. These types of components are called *business components*. You can see examples in Figure 11-11 of the different business components that might exist inside a Shipping bounded context.

Don't try too hard to identify business components up front. Instead, tease them apart over time. Often you will notice patterns in communication with domain experts, or patterns in code that repeatedly check for the same condition. These are your triggers to dig deeper and restructure your model(s).

One important rule is that SOA services are just logical containers; once you've decomposed a bounded context in business components, the service itself has no remaining artifacts or behavior.

Just as services need to be decoupled and share no dependencies, the same rule applies to business components—no shared dependencies, even with other business components inside the same bounded context. However, business components themselves are also just logical containers that are composed of components.

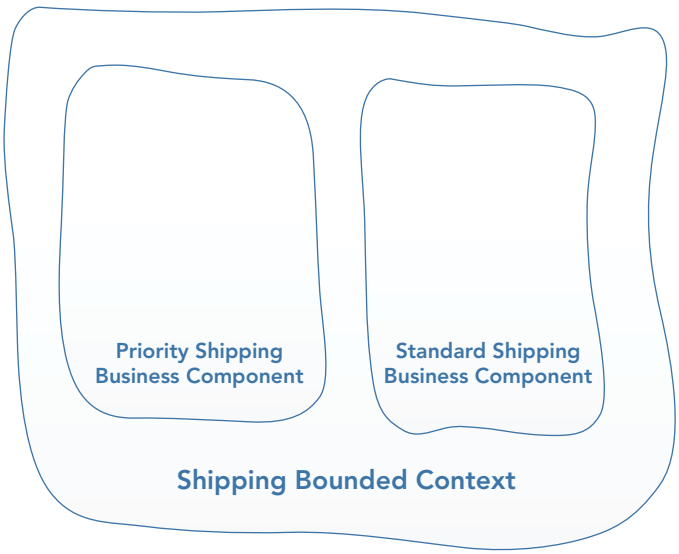


FIGURE 11-11: Decomposing the shipping bounded context into business components.

> **WARNING** *To gain the business benefits of teams that can work in isolation and the technical benefits of loosely coupled services that form scalable, resilient distributed systems, business components should ideally not make RPC calls between each other or have access to the same database. As soon as these coupling points are introduced, the problems outlined at the start of this chapter reappear.*
>
> *Your good judgment and experience will be your guide for deciding whether you should give in to the cravings of a coupling that may have short-term benefits.*

Decompose Business Components into Components

Business components may be responsible for handling multiple events, so you should break them down for further benefits. For example, in the Shipping bounded context, the Priority Shipping business component may handle `OrderPlaced` and `OrderCancelled` messages. By having these

messages processed by different software components, it's easier to better align hardware resources with the needs of the business. The business may want a rapid response to the `OrderPlaced` event so that shipping is arranged as soon as possible. However, it's not so important for them to respond as quickly to `OrderCancelled` messages because there are so few of them. Therefore, if the Priority Shipping business component is broken down into the Arrange Shipping component and the Cancel Shipping component, the business can put the Arrange Shipping component on blazing-fast, bare-metal servers while allowing the Cancel Shipping component to slowly grind away on a small virtualized server. Figure 11-12 shows an example of how the business components inside a Shipping bounded context could be broken down into components.

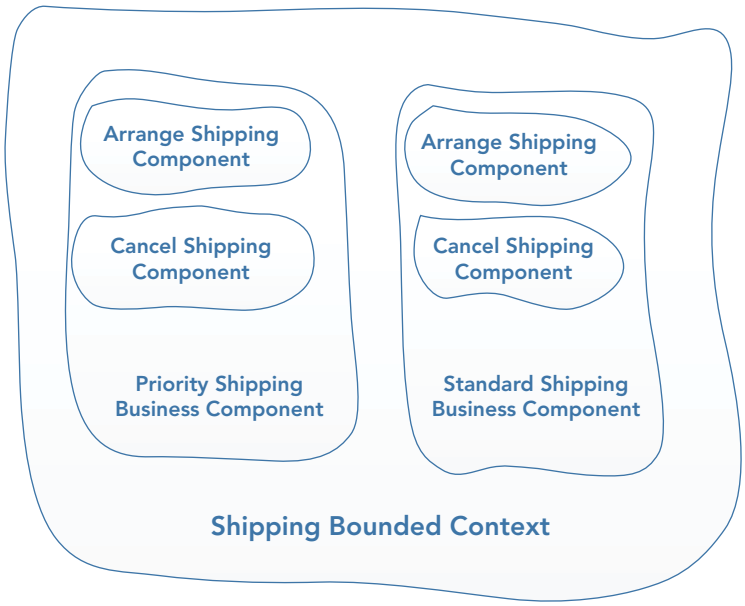


FIGURE 11-12 Shipping bounded context broken down into business components and components.

Ultimately, the benefit of components is that the business can spend money intelligently on the specific parts of the system that are likely to result in increased revenue. Hardware economics is just one of the benefits, though. Another is the possibility of locating components on different networks in line with business priority and performance needs. So, some bounded contexts that need high performance can have blazing-fast servers and dedicated high-bandwidth networks. In general, by having granular components you have more flexibility to align with the needs of the business.

As you may have gathered, components are the unit of deployment. Figure 11-13 visualizes this by showing how you can deploy different components in an e-commerce application on different machines or cloud instances with varying resource levels (CPU, RAM, SSDs, and so on).



FIGURE 11-13: Possible deployment view of components in the Shipping bounded context.

Whereas bounded contexts and business components are distinct enough that having RPC calls between them or a shared database can lead to resilience or scalability problems, the same is not true for components. Components often work together very closely; therefore, having shared dependencies like a database increases cohesion without necessarily causing problems. You are still free to use messaging between components if you feel it gives the best trade off in a given situation, though.

You can use Figure 11-14 to help you remember that shared dependencies may exist only inside a business component, not between them.

The term *component*, like many others in software development, is vague and ambiguous. In this chapter, you can see that there are business components and components (without a prefix). Unfortunately, the community hasn't really come up with a standard name for what this chapter refers to as just *components*. Udi Dahan refers to them as *autonomous components* (`http://www.drdobbs.com/web-development/business-and-autonomous-components-in-so/192200219`). Others refer to them as *components* (as in this chapter) based on the component-based development definition (`http://en.wikipedia.org/wiki/Component-based_software_engineering`). Yet others are now starting to refer to them as *micro services*, as shown in the next section.

Going Even Further with Micro Service Architecture

Netflix uses a fine-grained approach to SOA that other companies are now starting to apply under the name Micro Service Architecture (MSA). Some of the biggest reported benefits of MSA are its time-to-market and experimentation advantages. So if you find yourself working for a business

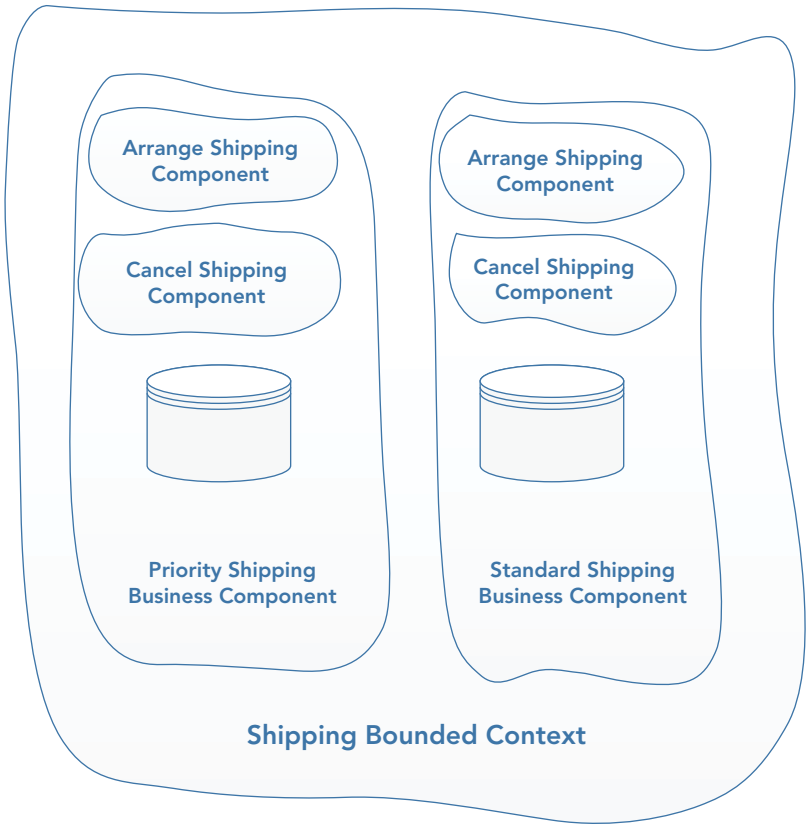


FIGURE 11-14: Sharing dependencies is only allowed between components inside the same business component.

that likes to make lots of changes to its system and measure the impact of every change to drive the evolution of new and existing features, you should consider MSA.

Following the guidance in this chapter so far will get you close to an MSA, where each component is akin to a micro service. However, MSA has some distinct differences. Each micro service must be completely autonomous so that the business feature can be added, removed, or modified without impacting any other micro service. Therefore, each micro service should have its down database and, almost certainly, communicate using events via publish/subscribe because commands and RPC introduce coupling. At least, this is the style of MSA described by Fred George, one of the first people to start talking about MSA, in his Oredev 2012 talk (`https://www.youtube.com/watch?v=q3q6ZsjZ_f0`). Another feature of micro services that many people agree on, although it is certainly contentious, is that they should be less than a thousand lines of code.

For more information on MSA, Martin Fowler's introductory blog series is a useful place to begin (`http://martinfowler.com/articles/microservices.html`).

THE SALIENT POINTS

- These days, the majority of applications you are likely to build are distributed systems because large web traffic arises from everyone having access to the Internet, and from many devices.
- Applying DDD to distributed systems still provides lots of benefits, but there are new challenges when integrating bounded contexts.
- Some of the challenges are technical, such as scalability and reliability, whereas others are social, such as integrating teams and developing at a high velocity.
- A number of techniques exist for building distributed systems that trade off simplicity, maintainability, and scalability. Database integration, for instance, can be quick to set up, but isn't recommended for use in high-scalability environments.
- RPC and messaging are the most common forms of distributed systems integration and the ones you are most likely to use. They are significantly different in nature, so it's essential you understand what benefits and complications they will add to your system.
- You can use the loosely coupled, business-oriented philosophy of SOA to help design your bounded context integration strategy by thinking of bounded contexts as SOA services.
- Combining SOA and reactive programming provides the platform to align your infrastructure with business priorities, deal with scalability and reliability challenges, and organize your teams by aligning them with bounded contexts to reduce communication overhead.

12

Integrating via Messaging

WHAT'S IN THIS CHAPTER?

- Integrating bounded contexts with asynchronous messaging using NServiceBus and Mass Transit
- Designing and modeling messaging systems around important events in the domain
- Understanding how messaging frameworks work
- Creating architectural diagrams
- Explanation of the conceptual differences between commands and events
- Theory and examples of industry-standard messaging patterns
- Dealing with eventual consistency
- Monitoring errors in messaging systems
- Monitoring service level agreements (SLAs) in messaging systems

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 12 download and individually named according to the names throughout the chapter.

Hopefully the concepts in the previous chapter got you excited about the prospect of building scalable distributed systems while still getting the full benefits of Domain-Driven Design (DDD). This chapter aims to provide you with the foundational skills you need so that you can immediately begin to apply those concepts to real systems. To acquire the necessary

initial skills, you're going to build an e-commerce messaging application using NServiceBus and Mass Transit. Along the way, you will see how to incorporate the knowledge you gain from domain experts into your model by using techniques that make domain concepts explicit in the code. You see how naming your messages after events that occur in the problem domain is one especially good technique for achieving this.

Throughout this chapter, the examples focus on showing common scenarios you are likely to come across. Some of these you will be familiar with from Chapter 11, " Introduction to Bounded Context Integration," such as adding reliability to synchronous Hypertext Transport Protocol (HTTP) calls that hit services you don't control. Building a system is only a small part of the puzzle, though, because systems need to be maintained. So in this chapter you'll also learn how to deal with changes to message formats that affect multiple teams, how to monitor the performance and errors of your messaging system, and how to scale out to multiple machines as business needs increase.

One of the big benefits of having loosely coupled systems is that you can have independent teams that can achieve high development throughout, as discussed in the previous chapter. In this chapter, you will see at the implementation level how you can organize your source files and projects to enable this. Before you get down to that level, it's often useful to design a system so that each team understands how its bounded contexts fit into the big picture. So in this chapter, you will learn how to create architecture diagrams that allow you to visualize important decisions and capture your domain-driven use-cases.

Messaging systems do have drawbacks, and this chapter demonstrates how to deal with some of them. One drawback compared to HTTP calls is that there is no standardization of formats. This can be a problem if you want to integrate two solutions using different messaging frameworks. However, this is a partially solved problem by using a messaging pattern called the messaging bridge. In this chapter, you will build a messaging bridge that connects an NServiceBus messaging system to a Mass Transit messaging system.

Before you begin writing code, there are some details about messaging systems that you need to learn about. These will greatly improve your ability to understand what you are building as you work through the examples. This chapter discusses those next.

MESSAGING FUNDAMENTALS

Messaging applications are fundamentally different from traditional, nondistributed object-oriented applications. Although they bring new benefits such as fault tolerance and scalability, they also bring new challenges, such as an asynchronous programming model that requires a significantly different mind-set. However, in addition to the theory presented in the previous chapter, the messaging fundamentals introduced in this section should give you all the foundational knowledge you need to begin building reactive applications using asynchronous messaging. The first concept you'll need to learn about is the message bus, which is the glue that holds the system together.

Message Bus

If one centralized component in the system was responsible for transmitting all the messages, the whole system would collapse if it stopped working. Equally as concerning, this single component

would make it more difficult for individual parts of the system to be scaled according to business need. You can solve these problems by using a *message bus*—a distributed system that has agents running on every component that sends or receives messages—avoiding the need for a centralized single point of failure. A message bus is all of these pieces collectively working together, as shown in Figure 12-1.

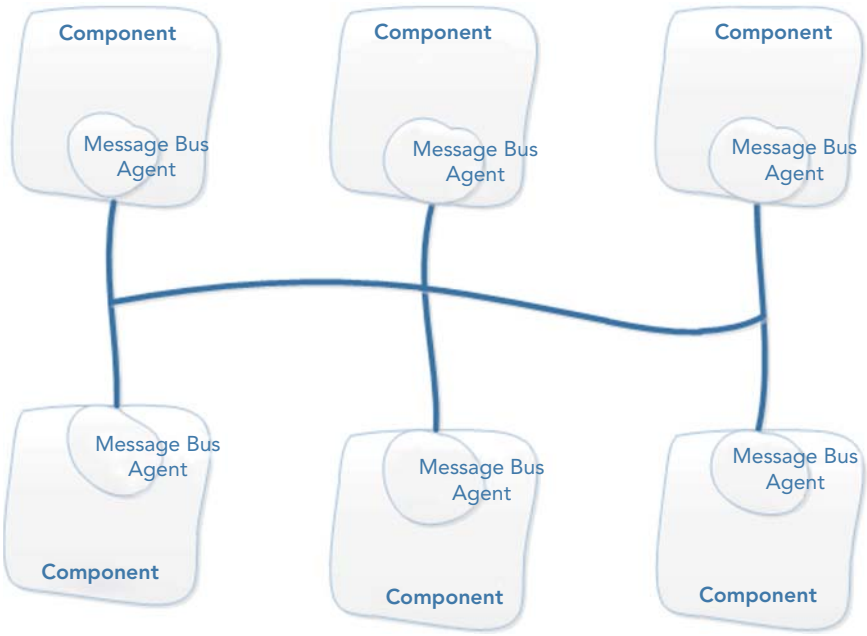


FIGURE 12-1: Message bus architecture.

> **NOTE** *A centralized component that receives and publishes all messages is known as a broker or message broker. Due to its fault-tolerance and scalability concerns, it is often overlooked in favor of the message bus. However, modern brokers are horizontally scalable and able to support many of the features of a message bus. One example is the highly regarded Apache Kafka (*`http://kafka.apache.org/`*), created and in-use at LinkedIn (*`http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying`*). Most of the concepts in this chapter could also be applied to a system using a modern broker like Kafka, though there could be more work with additional challenges and trade-offs.*

Using a message bus allows bounded contexts to be completely decoupled from each other. Each bounded context—more specifically, each component—is connected only to the bus. If a component goes offline, none of the other connected components is affected.

> **NOTE** *As mentioned in the previous chapter, components in this chapter refer to distinct applications that can be independently deployed and distributed. You may see them referred to as autonomous components, distributed components, messaging components, or even micro services elsewhere. Unfortunately the community has not found an accurate, unambiguous, and consistent name for them yet.*

Reliable Messaging

Sending messages from one of your bounded contexts to another could be costly if there were no guarantees the message would ever get there—customers aren't very happy when you charge their credit card but fail to ship their product because a message got lost in your system somewhere. This is the reason that reliable messaging is needed. Unfortunately, it's almost impossible to guarantee that a message will always be delivered only once. If a message is sent and no acknowledgement is received, it can be sent again. But what if the message was received but the acknowledgement wasn't? Of course, the same message will be sent again under the incorrect assumption that the first was not received.

Because of the challenges with reliable delivery, there are a variety of reliable messaging patterns, each with its own challenges and trade-offs, including: at-least-once, at-most-once, and only-once delivery as previously alluded to.

As you learn about messaging systems, you'll see that at-least-once is normally the preferred option, even though it does still carry the risk that a message may be handled twice. To avoid problems that this may cause, like charging valuable customers twice for a single order, this chapter shows how to combine at-least-once-delivery with idempotent messages.

> **NOTE** *Idempotent messages are messages that can be sent multiple times and only processed once. For example, if a message that causes a payment to be processed is sent twice, the recipient will only process it the first time. This is often achieved with some form of identifier, like a unique payment reference.*

At-least-once delivery involves retrying messages that failed or no acknowledgement from the receiver was received (appearing to fail). The actual plumbing for retrying failed or unacknowledged messages relies on a pattern known as store-and-forward.

Store-and-Forward

When a message is sent, it might not reach the recipient. The network could fail, hardware could fail, or programming errors could manifest. The store-and-forward pattern remedies many such problems by storing the message before it is sent. If the message reaches the recipient and is acknowledged, the local copy is deleted. However, if the message does not reach the recipient, it is tried again. You'll see in this chapter how messaging frameworks deal with most of the complexity

by allowing you to set up rules about how often messages should be retried and how long to wait between attempts.

Most messaging frameworks use queues to store messages. So when Service A sends Service B a message, the message ends up on Service B's queue. When Service B processes the message, it takes it off the queue. However, if Service B doesn't complete processing the message, it also gets put back on the queue and retried later.

Commands and Events

Sometimes you will want to send messages that specify something needs to happen, such as `PlaceOrder` from Chapter 11. These kinds of messages are known as *commands*. With commands there is a logical coupling between the sender and the receiver because the sender knows how the receiver should handle the message. Commands are only handled by one receiver, which is quite inflexible. Alternatively, events signal that something happened, such as `OrderCreated`. Events are more loosely coupled than commands because the sender doesn't know how the receiver will handle the message. In fact, the sender doesn't know who handles the messages. This is because events are based on the publish/subscribe pattern.

The looser coupling offered by publish/subscribe means events are often preferable to commands. One significant advantage of events is that you can add new subscribers without changing existing code. This allows the business to add a completely new bounded context without changing any existing code or slowing down other development teams. In the e-commerce example from the previous chapter, the new Marketing bounded context could be added to the flow just by subscribing to the `OrderCreated` event, as illustrated in Figure 12-2.

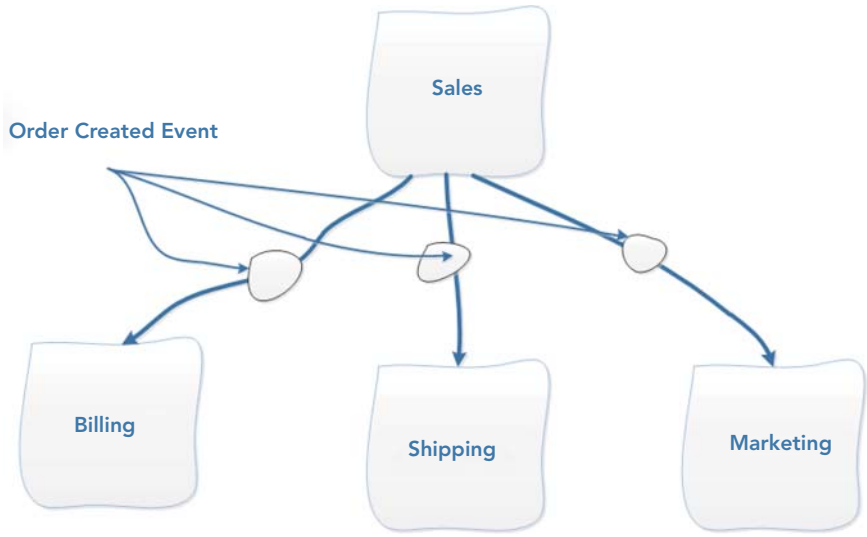


FIGURE 12-2: Adding new event subscribers doesn't affect existing code.

You might run into some debate online about naming commands and events. You're free to make your own choices, but the most common opinion is to name commands as instructions that you *want* to happen—`PlaceOrder`, `ChangeAddress`, `RefundAccount`—and to name events in the past tense describing what *has* happened—`OrderCreated`, `MovedAddress`, `AccountRefunded`.

Eventual Consistency

If you're building messaging systems in which each bounded context has its own database, you will end up in situations that aren't common in systems that use big transactions. An order without payment confirmation is a common example. In a system that uses transactions, this state might not be possible because creating an order is part of the same atomic operation that processes payments. Should the payment fail, no order will be created. Conversely, messaging solutions would allow this state because they remove big transactions and are eventually consistent.

One of the most important aspects of eventually consistent systems is managing the user experience. In consistent systems, users wait until the whole transaction completes. At which point, they know the order was created, the payment was processed, and the shipping was arranged. However, you know from Chapter 11 that big transactions do not always scale very well. In eventually consistent systems, users often get an immediate confirmation that the intent to place an order has been received, with a further e-mail confirmation later when payment and shipping have been processed and arranged, which can appear to be a degradation of user experience.

You can use eventual consistency as a positive by turning to the business and asking what should happen in eventually consistent states. Sometimes you can unlock hidden business policies or opportunities. For more detailed reading, Udi Dahan's "Race Conditions Don't Exist" blog post is highly recommended (`http://www.udidahan.com/2010/08/31/race-conditions-dont-exist/`).

> **NOTE** *Most of the messaging patterns in this chapter were formalized by Gregor Hohpe and Bobby Woolf in their highly influential book* Enterprise Integration Patterns. *All the patterns, and more, are freely accessible on that book's website (*`http://www.eaipatterns.com/`*).*

BUILDING AN E-COMMERCE APPLICATION WITH NSERVICEBUS

Now is your opportunity to get practical experience building a reactive messaging system. To begin with, you'll build an e-commerce system where users can make online purchases of their favorite products. You will use commands, events, and other well-known messaging patterns to appreciate how messaging systems provide the platform for enhanced scalability and reliability. When you're ready, your first task is to download NServiceBus so that your machine is equipped with the necessary dependencies to work through the examples.

> **NOTE** *To follow along with the examples in this chapter, you need to install the freely available Visual Studio Express 2013 for Web (*`http://www.visualstudio.com/products/visual-studio-express-vs`*). You can also use one of the full Visual Studio editions if you prefer.*

It's not too difficult to get started with NServiceBus; simply download and run the installer. To follow along with this chapter, you need to choose version 4.3.3 (`https://github.com/Particular/NServiceBus/releases/download/4.3.3/Particular.NServiceBus-4.3.3.exe`). This adds the required dependencies to your operating system, including Microsoft Message Queuing (MSMQ) and the Distributed Transactions Coordinator (DTC). As you will see shortly, you still need to reference the NServiceBus assemblies in your projects.

> **NOTE** *As you work through the examples in this chapter, remember that you are more than welcome to ask any questions or share or any thoughts you may have on the Wrox discussion forum.*

Designing the System

Before you crack open an IDE, it can be helpful to visualize what you are trying to build. This can be particularly useful when some of the concepts are new. Seeing how they fit together allows you to better understand what you're actually building as you go along. Designing the e-commerce application in this chapter requires three design steps. One step is to create a containers diagram showing application groupings, technology choices, and communication protocols. Another step is to create a component diagram showing the flow of logic between bounded contexts. But the first and most important step is to start with the domain.

Domain-Driven Design

As you've learned in the first part of this book, arguably the most important aspect to building software is to understand why you are building it so that you can provide value to people you are building the system for. So when designing a system, it is highly recommended that you spend a significant amount of time with domain experts building up a ubiquitous language (UL) and making implicit domain concepts explicit. Several DDD practitioners recommend that you start by identifying important events that occur in the domain. This is known as *Event Storming*, and it can be highly synergistic with messaging applications (`http://ziobrando.blogspot.co.uk/2013/11/introducing-event-storming.html`).

Domain Events

When you're building bounded contexts that integrate by sharing commands and events, you have an excellent opportunity to map important events that occur in the domain with the events occurring in the software system. This pattern is used by a number of well-known DDD practitioners to express

domain concepts explicitly in the messaging systems they build. To do this yourself, you first need to identify important events that occur in the real-world domain. These are known as domain events.

In Part I of this book, "The Principles and Practices of Domain-Driven Design," you saw a number of ways to capture domain knowledge, including events. This normally occurs when you are with domain experts during knowledge-crunching sessions, informal get-togethers, or even team lunches and such. In the previous chapter, a number of events occurred in the real-world domain and would still occur even if there was no software system. Some of them are Order Placed, Payment Accepted, and Shipping Arranged. When you are building a messaging system, it can pay dividends if you make an extra effort to look for domain events like these.

Once captured, your domain events form part of the UL, and you can start to piece them together to learn how they combine to form full business use cases. An excellent way to store and share this knowledge is by visualizing the sequence of events on a component diagram.

Component Diagrams

By sketching out high-level logic before you write code, you are in a better position to build the component efficiently and properly because you understand what you are doing. This is where a component diagram provides a lot of benefit. A useful time to start creating component diagrams is during knowledge-crunching sessions with domain experts. You can produce basic sketches together using just boxes and lines to communicate domain events and processes. When you then sit down to start coding, you already have an idea of what you need to build and terminology from the UL that needs to be modeled in your system.

Component diagrams have no formal structure; they communicate the flow of logic or interaction between certain components. You shouldn't go too high level and show technology choices, and you shouldn't go too detailed and show class or method names. You can see an example of this in Figure 12-3, which shows the flow of messages between bounded contexts in the e-commerce application you're going to build.

You can have as many component diagrams as you feel are necessary. In this example, there is one component diagram that communicates the flow of the order placement scenario. You could start by following this convention: having one component diagram per high-level use case and assessing how that works for you.

Containers Diagrams

Once you have spent time with domain experts and you are starting to understand the domain, at some point you'll need to start building the system. It's at this time you need to map the requirements of the business onto a working, distributed software system that provides them value. One technique that can help you balance the functional domain requirements with the nonfunctional technical requirements is to create containers diagrams.

How do the different parts of an application talk to each other? How can you be confident the system will provide the necessary fault tolerance and scalability? How can you make sure everyone on the team understands what is being built so they don't do something completely wrong? These are important questions for most software projects, and a containers diagram helps you answer them. This type of diagram shows how different parts of the system are grouped, how different parts of the system communicate, and what the major technology choices are.

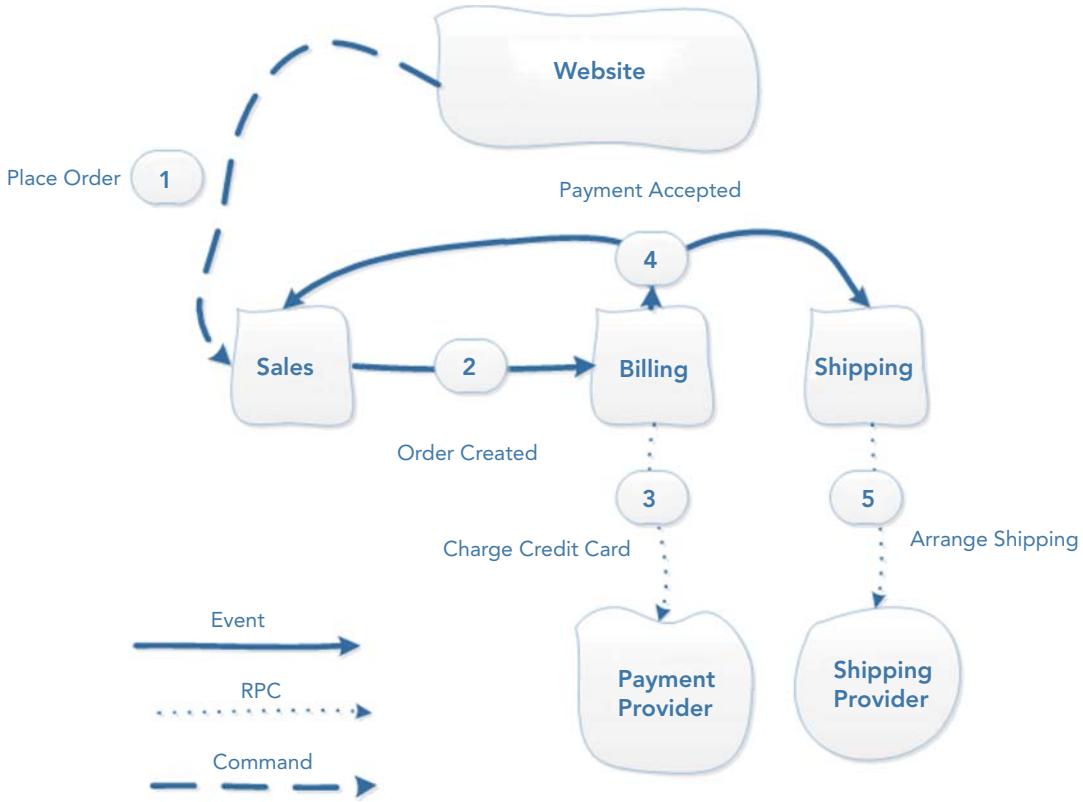


FIGURE 12-3: A component diagram showing domain events.

> **NOTE** *Major technology choices are ones that have a big impact on the development, delivery, or maintenance of a project. They are usually those that are harder to change. Generally, major technology choices are operating systems, programming languages/run times, web servers, middleware, and major application frameworks such as web frameworks or concurrency frameworks.*

Take a look at Figure 12-4, which is a containers diagram of the e-commerce application that is the focus of this chapter.

Some of the key points to take note of in the system design follow:

- Internal communication between bounded contexts uses messaging.
- Unreliable communication with the external payment provider is wrapped with a messaging gateway to add reliability.
- Each bounded context is able to use different technologies, including choice of database technology.

- Bounded contexts do not share databases or other dependencies.
- The website retrieves information from bounded contexts using the HTTP application programming interfaces (APIs) they provide via AJAX requests from the browser.

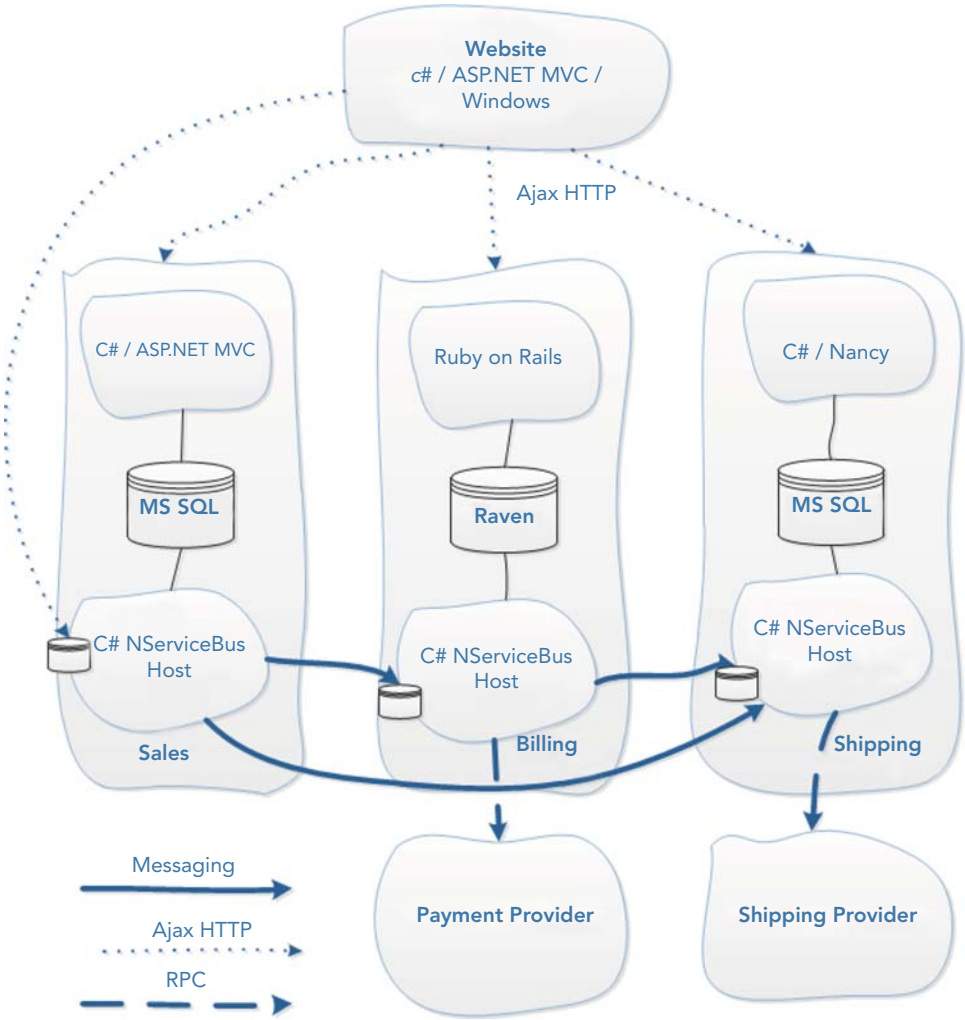


FIGURE 12-4: A containers diagram for a typical e-commerce application.

From the containers diagram in Figure 12-4, you can see that fault tolerance and scalability have been addressed by using messaging. You can also see that each bounded context provides an API that the browser can use to get data to show on web pages using HTTP. The diagram also validates some technology choices. For instance, because each bounded context is using C#, you can use a single messaging technology (NServiceBus) that makes integration easier. However, if your

requirements stated that new bounded contexts may be built using different languages and platforms, the design might make you think deeper about choosing NServiceBus or any messaging technology and instead prefer REST (which you will build a system with in the next chapter) for its friendlier cross-platform integration.

Some of the design decisions may make sense following the discussion from the previous chapter. Other decisions you may find counterintuitive or just disagree with. And that's completely fine. By the end of this chapter, all the design decisions in this application will have been covered. You can then decide for yourself if you agree.

> **WARNING** *Try to keep diagrams concise and effective by maintaining all details at the same level of abstraction. For instance, the containers and component diagram in this section could have been merged into a single diagram, but there would have been lots of information used for different purposes, making it harder to understand. For more guidance, refer to Simon Brown's "Coding the Architecture" blog:* `http://static.codingthearchitecture.com/c4.pdf`.

> **NOTE** *Not everything on the containers diagram will be built in this chapter. In particular, the examples will not use real databases. They were included in the design to show a more detailed design process and accentuate the fact that technologies do not need to be consistent across bounded contexts.*

Evolutionary Architecture

Through frequent collaboration with domain experts, you continue to learn more about the domain. You've seen that you need to refactor your domain models and enhance the UL as the learning process continues. Accordingly, the design of your system needs to be updated to reflect your learnings. If new events are added to the domain or your context boundaries need to be adjusted, you should try to model your architecture around your new findings and update your diagrams accordingly. When you do this, your diagrams continue to be relevant, help people make informed business decisions, and better understand what they are building. Your architecture will also be more closely aligned with the problem domain.

> **WARNING** *Be sure to prioritize the evolution of your architecture and the revision of your diagrams with the needs of the business. Sometimes getting a new feature out in front of customers is more economical than perfectly aligning context boundaries. However, you should think about informing the business about the short- and long-term costs of accruing this technical debt.*

Sending Commands from a Web Application

Placing an order begins with customers checking out on the website. This is the first step in the component diagram shown in Figure 12-3. To get started with the messaging system, you need to create a web application that can initiate this process by sending a `PlaceOrder` command representing the customer's wishes. For this example, a simple MCV 4 website with just a single page where orders can be placed will be used. Figure 12-5 shows what this will look like.

As you can clearly see, the appearance of the UI is not a significant part of this chapter. However, pushing data into the messaging system is.



FIGURE 12-5: The basic web page for placing an order.

> **NOTE** *When working through the examples in this part of the chapter, if you need any assistance or are unsure what to do, feel free to consult this book's discussion forum.*

Creating a Web Application to Send Messages with NServiceBus

Before creating the web application, you need to create a blank solution inside Visual Studio called `DDDesign`. This solution will eventually host the entire system. Figure 12-6 illustrates the process of creating an empty Visual Studio solution.

In an NServiceBus application, messages are the contract between bounded contexts. The messages must be accessible to the bounded context that publishes them and the bounded contexts that consume them. So a good convention is for each bounded context to have a project that contains just the messages it publishes. Other bounded contexts can reference this project to access the messages.

> **WARNING** *Be careful that bounded contexts only share projects containing messages. You need to ensure that this is the* only *dependency that exists between bounded contexts in a messaging system because sharing code can introduce coupling and remove the benefits of being loosely coupled.*

Defining Messages

The component diagram in Figure 12-3 shows that the first step is a `PlaceOrder` command being sent from the website to the Sales bounded context. This makes the `PlaceOrder` command a

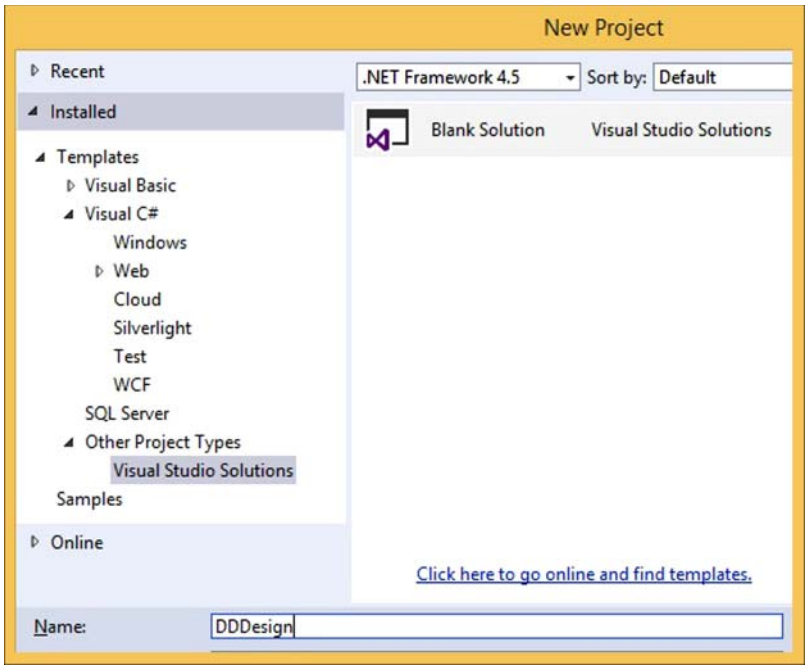


FIGURE 12-6: Creating an empty Visual Studio solution.

message that belongs to the Sales bounded context. Therefore, the first project you need to create is the Sales bounded context's messages project. You can do that by adding a new C# class library called `Sales.Messages` to the `DDDesign` solution you just created.

Inside your newly created `Sales.Messages` project, you can now add the `PlaceOrder` command that represents the real-world scenario of a user indicating he would like to buy certain products. For this example, add a folder called `Commands` to the root of the project and a class called `PlaceOrder` with the following content:

```
namespace Sales.Messages.Commands
{
    public class PlaceOrder
    {
        public string UserId { get; set; }

        public string[] ProductIds { get; set; }

        public string ShippingTypeId { get; set;}

        public DateTime TimeStamp { get; set; }
    }
}
```

> **NOTE** *Feel free to delete* `Class1.cs` *that is added by default to every C# class library in Visual Studio. You can do this every time you create a class library in this chapter.*

Now you're in a position to create the website that can send the `PlaceOrder` command. Inside the same solution, you need to add a new ASP.NET MVC 4 project, choosing the empty template and the Razor View Engine. Name this project `DDDesign.Web`. These steps are illustrated in Figure 12-7 and Figure 12-8.

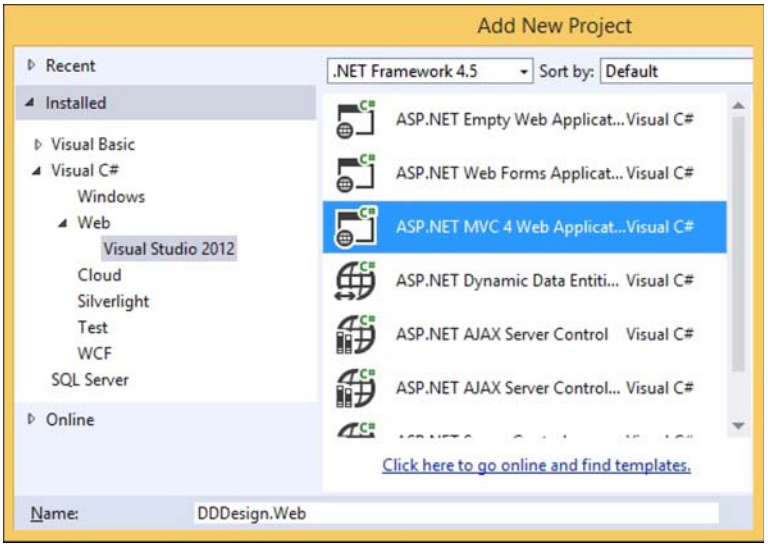


FIGURE 12-7: Adding the DDDesign.Web MVC 4 web application.

Now you need to right-click on the `DDDesign.Web` project icon in the Solution Explorer and choose Add → Reference. Then select the Solution → Projects option. Finally, you can add a reference to the `Sales.Messages` project, as illustrated in Figure 12-9.

For the website to send commands, it needs NServiceBus running inside it. You can make that happen by adding a reference to NServiceBus using the NuGet package manager console inside Visual Studio. Just run the `Install-Package` command, as demonstrated in Figure 12-10, and the following snippet:

```
Install-Package NServiceBus -ProjectName DDDesign.
Web -Version 4.3.3
```

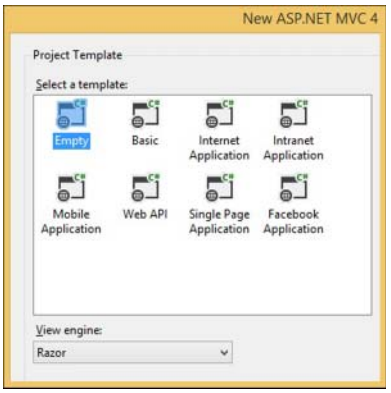


FIGURE 12-8: Selecting empty ASP.NET MVC template with the Razor view engine.

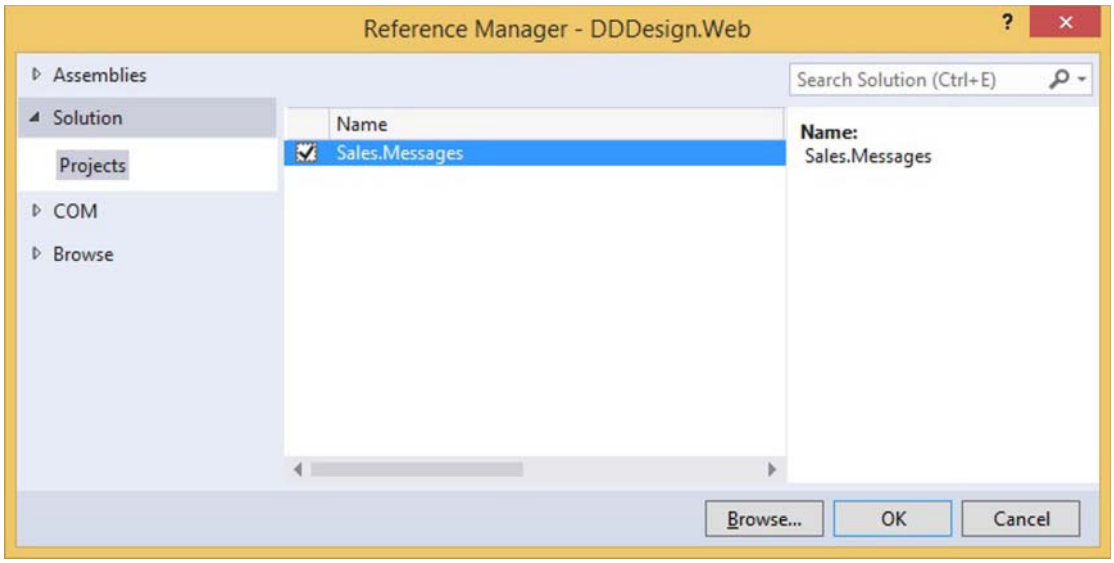


FIGURE 12-9: Adding a reference to the Sales.Messages project from the DDDesign.Web project.

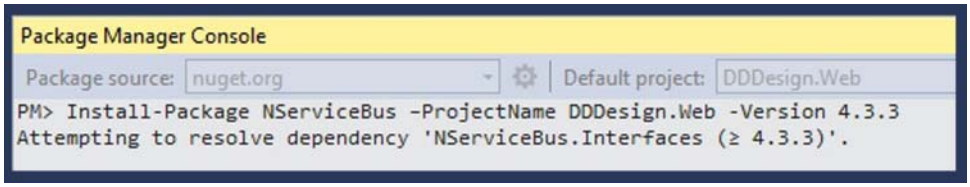


FIGURE 12-10: Installing NServiceBus with the NuGet package manager console.

Configuring a Send-Only NServiceBus Client

Now that you have referenced the NServiceBus dependencies, you need to configure NServiceBus to actually run inside the web application. NServiceBus makes this quite easy with the self-hosting option, which you can configure in `Global.asax.cs`, as shown in Listing 12-1.

LISTING 12-1: Configuring an NServiceBus Send-Only Client Inside ASP.NET MVC 4

```
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Routing;
using NServiceBus;
using NServiceBus.Installation.Environments;

namespace DDDesign.Web
{
```

(*continued*)

LISTING 12-1: *(continued)*

```
    public class MvcApplication : System.Web.HttpApplication
    {
        private static IBus bus;

        public static IBus Bus { get { return bus; } }

        protected void Application_Start()
        {
            Configure.Serialization.Xml();
            bus = Configure.With()
                           .DefaultBuilder()
                           .DefiningCommandsAs(t => t.Namespace != null
                             && t.Namespace.Contains("Commands"))
                           .UseTransport<Msmq>()
                           .UnicastBus()
                           .SendOnly();

            // the following lines were provided by default
            AreaRegistration.RegisterAllAreas();

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

As Listing 12-1 shows, there isn't too much configuration needed. However, among those few lines are some key details. First, the last line of configuration, `SendOnly()`, tells NServiceBus that this application only sends messages and does not receive them. If you look back at the containers diagram, you can see that this is exactly as per the design. In NServiceBus nomenclature, a send-only application is known as a *client*.

The `DefiningCommandsAs()` method is used to specify a convention that lets NServiceBus know which classes in the solution are commands. In this example, you can see it is any class whose namespace contains the word `Commands`. If you look back to where you defined the `PlaceOrder` command, you'll see that it is inside a folder called `Commands` and therefore will be picked up by this convention. You don't have to use this convention; you can use any convention you like, and you can even use XML configuration instead if you prefer.

The other key piece of configuration is the line `UseTransport<Msmq>()`. As mentioned at the start of the chapter, a message bus usually stores messages in a queue for fault tolerance so that messages can be retried if there is an error. This line of configuration instructs NServiceBus to use Microsoft's MSMQ. You can also use RabbitMQ or ActiveMQ.

> **NOTE** *For more information about configuring NServiceBus, including choices of transport technology and conventions, check out the official documentation:* `http://particular.net/documentation/nservicebus`.

Sending Commands

Now that the bus is configured and available, you can use it to send commands. In this example, that is accomplished by adding an `OrdersController` that a web page posts the user's order back to. The `OrdersController` then sends the `PlaceOrder` command using the bus you configured earlier. To add the `OrdersController`, right-click on the `Controllers` folder inside the `DDDesign.Web` project and choose Add → Controller. Type in **`OrdersController`** as the controller name and click Add. Figure 12-11 illustrates this.

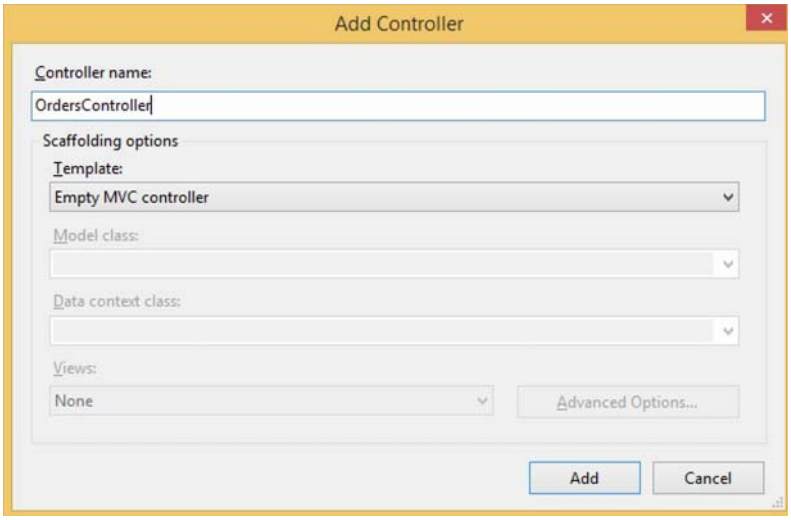


FIGURE 12-11: Adding the Orders controller.

Once you've created the `OrdersController`, you can replace the entire contents of the file with the code in Listing 12-2.

LISTING 12-2: OrdersController That Sends PlaceOrder Commands with NServiceBus

```
using Sales.Messages.Commands;
using System;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace DDDesign.Web.Controllers
{
    public class OrdersController : Controller
    {
        [HttpGet]
        public ActionResult Index()
        {
            return View();
```

(*continued*)

LISTING 12-2: *(continued)*

```
        }

        [HttpPost]
        public ActionResult Place(string userId, string productIds,
                                  string shippingTypeId)
        {
            var realProductIds = productIds.Split(',');
            var placeOrderCommand = new PlaceOrder
            {
                UserId = userId,
                ProductIds = realProductIds,
                ShippingTypeId = shippingTypeId,
                TimeStamp = DateTime.Now
            };
            MvcApplication.Bus.Send(
                "Sales.Orders.OrderCreated", placeOrderCommand
            );

            return Content(
             "Your order has been placed. " +
             "You will receive email confirmation shortly."
            );
        }

    }
}
```

This example is intentionally quite simplistic to accentuate the messaging aspects. Sending messages with NServiceBus is relatively straightforward anyway. You just create an instance of your message and, together with the name of the recipient, pass it to `Bus.Send()`, which sends the message asynchronously. When sending commands, make sure you specify the recipient because commands are only handled in a single place. Later you will see that specifying the recipients is not necessary when publishing events. This is a useful distinction to remember.

You can also see in this case an example of eventual consistency. The user gets an immediate response before the order has successfully been processed by each part of the system. This was specifically highlighted in Chapter 11 because it provides fault tolerance should any of the steps in the use case fail. If you remember, it also allows the website to be scaled independently of other parts of the system because it does not wait for them to do processing as would be the case with remote procedure call (RPC) solutions. It would be useful to briefly revisit the last chapter if these benefits aren't clear.

> **WARNING** *Listing 12-2 takes the product IDs for an order as a comma-separated string. This is not a recommended practice, and all web frameworks have a better way to achieve this, usually known as* model-binding. *This example takes a basic approach to accentuate the relevant details of using NServiceBus and messaging. The same also applies for returning string content. You should still return a web page or API response as you normally would on your real projects.*

> **WARNING** *Throughout this chapter static classes and static variables are used because they are the simplest solution and allow the examples to remain focused on the relevant concepts. In a real application you should think very carefully about static methods and classes because they introduce tight-coupling that can hinder code maintainability and testability. If you're not familiar with dependency inversion, it is highly worthwhile learning. One good source is* `http://martinfowler.com/articles/dipInTheWild.html`.
>
> *Some examples of static usages in this chapter that are not recommended for production applications are:* `MvcApplication.Bus.Send()`, `Database.GetCardDetailsFor()`, *and* `PaymentProvider.ChargeCreditCard()`.

Only one thing now remains before `PlaceOrder` commands are sent: a web page with a form to enter the order details needs to be added. To create that web page, you first need to add a folder called `Orders` inside the `Views` folder in the `DDDesign.Web` project. Inside the newly created `Orders` folder, add a file called `Index.cshtml`. Then delete the entire contents of `Index.cshtml` and replace it with the contents of Listing 12-3. If you have done this correctly, your solution structure should resemble Figure 12-12.

Listing 12-3 is just a simple HTML form that allows the order to be placed. Figure 12-5 shows how this page appears when rendered.

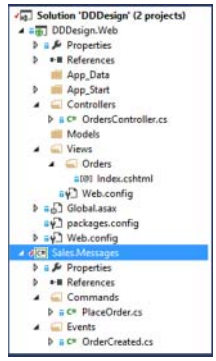


FIGURE 12-12: Solution structure after adding Orders controller and view.

LISTING 12-3: Index.cshtml

```
<!DOCTYPE html>
<html>
    <head>
        <title>Place an order</title>
    </head>
    <body>
        <h1>Place an order</h1>
        <form method="post" action="/orders/place">
            <p>
             UserId: <input type="text" name="userId" />
            </p>
            <p>
             ProductIds: <input type="text" name="productIds" />
            </p>
            <p>
             ShippingTypeId: <input type="text" name="shippingTypeId" />
            </p>
            <input type="submit" value="Place order" />
        </form>
    </body>
</html>
```

Handling Commands and Publishing Events

Now that the web application has a page that allows customers to create orders and a controller that will use the supplied details to send a `PlaceOrder` command on the bus, you need to actually handle `PlaceOrder` commands. Looking back to the diagram in Figure 12-3, you can see that `PlaceOrder` commands are handled by the Sales bounded context, which processes the command and publishes an `OrderCreated` event. So in this section, you'll create a component in the Sales bounded context that contains this logic.

Creating an NServiceBus Server to Handle Commands

NServiceBus servers are like Windows Services—they are applications that run in the background without a UI. In this example, an NServiceBus server is added to handle `PlaceOrder` commands inside the Sales bounded context. Following the naming convention of `{BoundedContext}.{BusinessComponent}.{Component}`, add a new C# class library called `Sales.Orders.OrderCreated`. The name `OrderCreated` indicates the message type published by the component. This is the convention for naming components used in this chapter.

To turn your class library into an NServiceBus server, you need to install the `NServiceBus.Host` NuGet package. You can do that by running the following command in the NuGet package manager console inside Visual Studio (it should be entered as a single-line command):

```
Install-Package NServiceBus.Host -ProjectName
Sales.Orders.OrderCreated -Version 4.3.3
```

NServiceBus servers come configured with many common defaults. But one thing they don't know about are your custom conventions. To set the conventions in the `Sales.Orders.OrderCreated` project, replace the contents of the `EndpointConfig` class that NServiceBus automatically creates with the code shown in Listing 12-4. All it's doing is applying the same defaults for locating commands that you set up for the web application, plus a similar convention for locating events. Do note the two additional interfaces that this class has been updated to inherit: `IWantCustomInitialization` and `AsA_Publisher`.

LISTING 12-4: Adding Custom Conventions to an NServiceBus EndpointConfig

```
using NServiceBus;

namespace Sales.Orders.OrderCreated
{
    public class EndpointConfig : IConfigureThisEndpoint, AsA_Server,
                              IWantCustomInitialization, AsA_Publisher
    {
        public void Init()
        {
            Configure.With()
                     .DefiningCommandsAs(t => t.Namespace != null
                        && t.Namespace.Contains("Commands"))
                     .DefiningEventsAs(t => t.Namespace != null
                        && t.Namespace.Contains("Events"));
```

```
        }
    }
}
```

Inside your `Sales.Orders.OrderCreated` component, you want to handle `PlaceOrder` commands. To be able to do this, you need to add a reference to the `Sales.Messages` project.

A useful convention for handling messages is to create a class called `{MessageName}Handler`. So in this example, you should create a class called `PlaceOrderHandler` in the root of the `Sales.Orders.OrderCreated` project. Once you've done that, you can replace the contents with the code shown in Listing 12-5.

LISTING 12-5: PlaceOrderHandler Handling PlaceOrder Commands

```
using System;
using NServiceBus;
using Sales.Messages.Commands;

namespace Sales.Orders.OrderCreated
{
    public class PlaceOrderHandler : IHandleMessages<PlaceOrder>
    {
       // dependency injected by NServiceBus
        public IBus Bus { get; set; }

        public void Handle(PlaceOrder message)
        {
            Console.WriteLine(
                @"Order for Products:{0} with shipping: {1}" +
                 " made by user: {2}",
                 String.Join(",", message.ProductIds),
                 message.ShippingTypeId, message.UserId
            );
        }
    }
}
```

Listing 12-5 shows the `PlaceOrderHandler` simply printing the details of the received `PlaceOrder` command to the console. This is just temporary so that you can run the application and see everything working so far. However, there are some important details to note regarding NServiceBus message handlers. First, NServiceBus knows that any class inheriting `IHandleMessages<T>` handles messages of type `T`. Second, it will inject an instance of `IBus` if you declare a property of type `IBus` named `Bus`. That's quite useful, as you are about to see shortly, because it allows you to send other messages, including events, inside your message handlers.

Configuring the Solution for Testing and Debugging

Before moving on to publishing events, this is a good opportunity to test that everything is working so far and that you have a basic understanding of how messaging works. NServiceBus makes it easy

to debug a distributed system on your local machine. All you need to do is configure each project to start up appropriately, as detailed in the following steps:

1. Right-click on the solution file in the Solution Explorer and select Properties (Figure 12-13).
2. Select the Startup Project option beneath Common Properties.
3. Activate the Multiple Startup Projects radio button.
4. Set the Action as `Start` for the `DDDesign.Web` and `Sales.Orders.OrderCreated` projects.
5. Move `DDDesign.Web` to the bottom of the list.
6. Check that your screen looks like Figure 12-14.
7. Click OK.

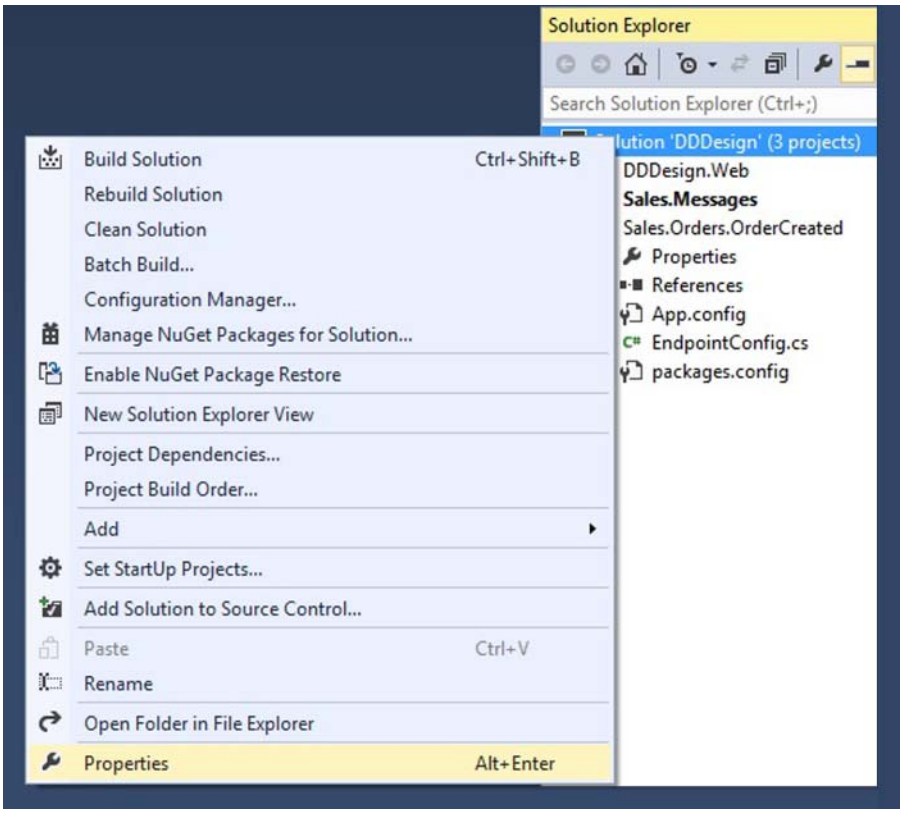


FIGURE 12-13: Selecting solution properties.

At this point, you can press F5 to start debugging the application. The web application loads in the browser, and the Sales bounded context loads in a console. If you look carefully at the console, you can see that NServiceBus takes care of lots of the hard work, such as creating all the necessary queues, as shown in Figure 12-15.

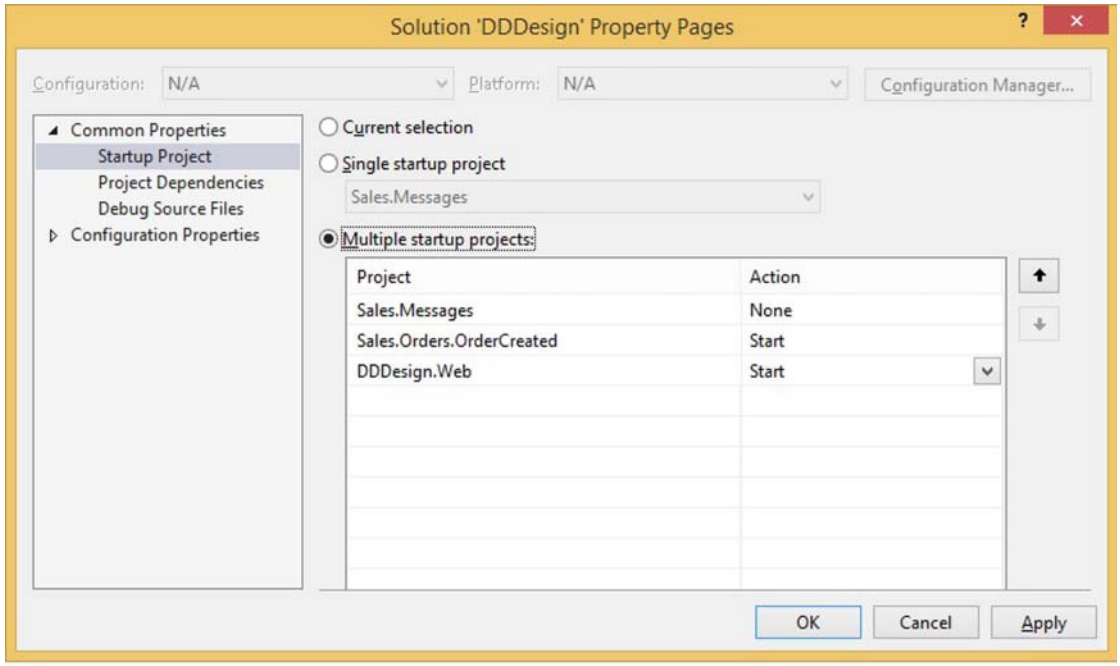


FIGURE 12-14: Configuring each project to start up.

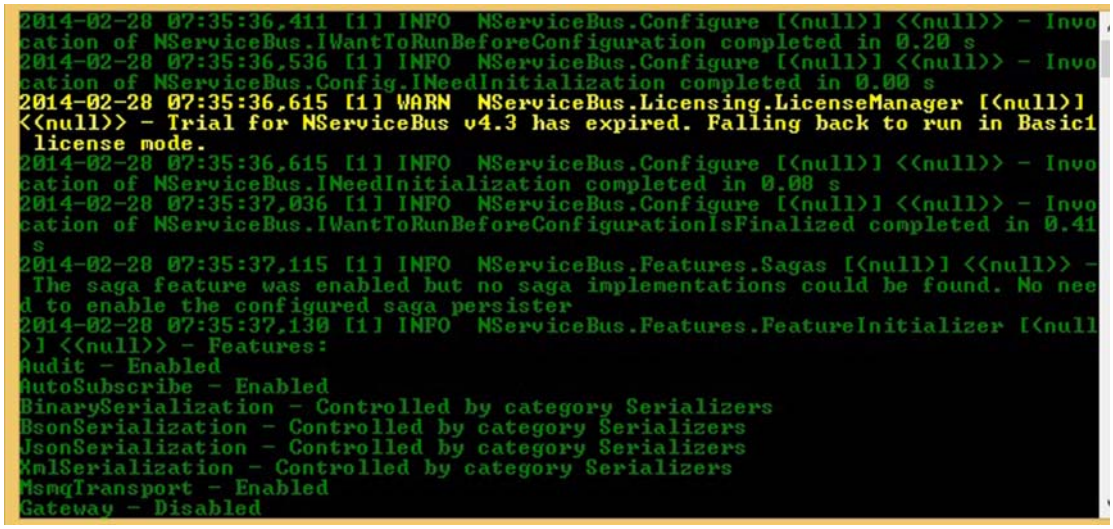


FIGURE 12-15: An NServiceBus server setting up an application.

To test the system, you need to navigate to the `/orders` action in the browser that automatically opened and fill out the form. (You just need to append "orders" to the localhost URL the browser automatically opens with.) Then look back at the console output of the host, and you will see it printing the details of the message it received.

Publishing Events

Now that the web application is sending commands that the Sales bounded context is successfully receiving, the second step on your component diagram shows that the Sales bounded context needs to create the order and publish an `OrderCreated` event when it has done so. To begin, you need to define the `OrderCreated` event in the `Sales.Messages` project. Similar to the way you created the `PlaceOrder` command, simply add a folder called `Events` in the root of the project, and then add a class called `OrderCreated` inside it, as shown in Figure 12-16.

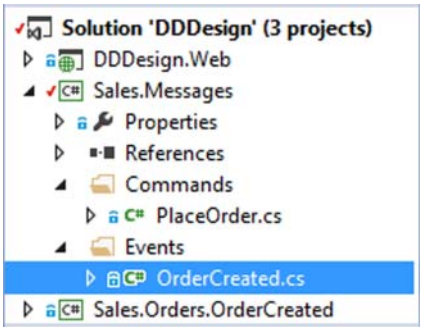


FIGURE 12-16: Adding the OrderCreated event to the Sales.Messages project.

Inside your `OrderCreated` event, you need to add the necessary pieces of information that are part of the event. Just add them as properties to the class, as shown in Listing 12-6.

LISTING 12-6: The OrderCreated Event

```
using System;
using System.Collections.Generic;

namespace Sales.Messages.Events
{
    public class OrderCreated
    {
        public string OrderId { get; set; }

        public string UserId { get; set; }

        public string[] ProductIds { get; set; }

        public string ShippingTypeId { get; set; }

        public DateTime TimeStamp { get; set; }

        public double Amount { get; set; }
    }
}
```

Now that you've defined the `OrderCreated` event, you are free to publish it when you need to. To publish any event, invoke `Bus.Publish()`, passing in an instance of the event. Listing 12-7 shows the updated `PlaceOrderHandler` saving the order to the database, and then notifying interested parties it has done so by publishing an `OrderCreated` event.

LISTING 12-7: The Updated PlaceOrderHandler Publishing and OrderCreated Event

```
using System;
using NServiceBus;
using Sales.Messages.Commands;
using System.Collections.Generic;

namespace Sales.Orders.OrderCreated
{
    public class PlaceOrderHandler : IHandleMessages<PlaceOrder>
    {
        public IBus Bus { get; set; }

        public void Handle(PlaceOrder message)
        {
            var orderId = Database.SaveOrder(
              message.ProductIds, message.UserId, message.ShippingTypeId
            );

            Console.WriteLine(
                @"Created order #{3} : Products:{0} " +
                "with shipping: {1} made by user: {2}",
                String.Join(",", message.ProductIds),
                message.ShippingTypeId, message.UserId, orderId
            );

            var orderCreatedEvent =
                new Sales.Messages.Events.OrderCreated
            {
                OrderId = orderId,
                UserId = message.UserId,
                ProductIds = message.ProductIds,
                ShippingTypeId = message.ShippingTypeId,
                TimeStamp = DateTime.Now,
                Amount = CalculateCostOf(message.ProductIds)
            };

            Bus.Publish(orderCreatedEvent);
        }

        private double CalculateCostOf(IEnumerable<string> productIds)
        {
            // database lookup, etc.
            return 1000.00;
        }
    }

    // This can be any database technology; it can differ between
    // business components
    public static class Database
    {
        private static int Id = 0;

        public static string SaveOrder(IEnumerable<string> productIds,
```

(*continued*)

LISTING 12-7: *(continued)*

```
                                    string userId, string shippingTypeId)
        {
            var nextOrderId = Id++;
            return nextOrderId.ToString();
        }
    }
}
```

For publishing events to be worthwhile, you need to be able to handle them and apply your domain policies.

Subscribing to Events

Handling events is really just the same as handling commands. To demonstrate, you'll now see how to implement the next step on your component diagram (Figure 12-3)—step 3—which involves the Billing bounded context subscribing to and handling the `OrderCreated` event that the Sales bounded context publishes. Your component diagram also shows that step 4 involves the Billing bounded context publishing a `PaymentAccepted` event. So, following the convention of naming components after the messages they send, you now need to add a new C# class library project to the solution called `Billing.Payments.PaymentAccepted`. Inside your new project, you need to carry out a few familiar steps:

1. Add a reference to the NServiceBus packages using the package manager (run as a single-line command):

```
Install-Package NServiceBus.Host -ProjectName
Billing.Payments.PaymentAccepted -Version 4.3.3
```

2. Add your event-locating conventions to `EndpointConfig`. This component will be publishing messages, so in the same file, configure it as a publisher by inheriting `AsA_Publisher`:

```
using NServiceBus;
namespace Billing.Payments.PaymentAccepted
{
    public class EndpointConfig : IConfigureThisEndpoint,
            AsA_Server, IWantCustomInitialization, AsA_Publisher
    {
        public void Init()
        {
            Configure.With()
                    .DefiningEventsAs(t => t.Namespace != null
                        && t.Namespace.Contains("Events"));
        }
    }
}
```

3. Add a reference to the `Sales.Message` project.

4. Add an `OrderCreatedHandler` to the root of the project:

```
using NServiceBus;
using Sales.Messages.Events;
using System;

namespace Billing.Payments.PaymentAccepted
{
    public class OrderCreatedHandler : IHandleMessages<OrderCreated>
    {
         public IBus Bus { get; set; }

         public void Handle(OrderCreated message)
         {
             Console.WriteLine(
              "Received order created event: OrderId: {0}",
              message.OrderId
             );
         }
    }
}
```

There's one additional step required when handling events with NServiceBus. You need to update the `App.config` in the subscriber's project to identify the source of the event messages it subscribes to. In this case, the `Billing.Payments.PaymentAccepted` component wants to subscribe to `OrderCreated` events published by the `Sales.Orders.OrderCreated` component. So the `App.config` in `Billing.Payments.PaymentAccepted` should be updated, as shown in Listing 12-8.

LISTING 12-8: Adding NServiceBus Event Subscription to App.config

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<configuration>
  <configSections>
    <section name="MessageForwardingInCaseOfFaultConfig"
      type="NServiceBus.Config.MessageForwardingInCaseOfFaultConfig,
NServiceBus.Core" />
    <section name="UnicastBusConfig"
             type="NServiceBus.Config.UnicastBusConfig,
NServiceBus.Core" />
    <section name="AuditConfig"
             type="NServiceBus.Config.AuditConfig, NServiceBus.Core" />
  </configSections>
  <MessageForwardingInCaseOfFaultConfig ErrorQueue="error" />
  <UnicastBusConfig>
    <MessageEndpointMappings>
      <add Messages="Sales.Messages"
           Type="Sales.Messages.OrderCreated"
           Endpoint="Sales.Orders.OrderCreated" />
    </MessageEndpointMappings>
  </UnicastBusConfig>
  <AuditConfig QueueName="audit" />
</configuration>
```

Making External HTTP Calls Reliable with Messaging Gateways

You saw in Chapter 11 that one of the major benefits of a reactive solution is improved fault tolerance. One particular scenario you saw was downtime of external services, such as the payment provider. In the old RPC scenario, the order would have failed. But in this solution, as you will see, you can insulate yourself from failures that happen to external services by using a messaging gateway. Messaging gateways wrap unreliable communication with a message so that it can be retried until the stricken service is available again.

Messaging Gateways Improve Fault Tolerance

Before implementing the messaging gateway, here's a quick example to help you fully understand why it's needed. Imagine that the `OrderCreatedHandler` was implemented as per Listing 12-9.

LISTING 12-9: Vulnerable Code Lacking a Messaging Gateway

```
public void Handle(OrderCreated message)
{
    Console.WriteLine(
        "Received order created event: OrderId: {0}",
        message.OrderId
    );
    var cardDetails = Database.GetCardDetailsFor(message.UserId);
    var confirmation = PaymentProvider.ChargeCreditCard(
        cardDetails, message.Amount
    );
    if (confirmation.IsSuccess)
    {
        Database.SavePaymentDetails(confirmation);
        Bus.Publish(
            new PaymentAccepted
            {
                OrderId = message.OrderId
            }
        );
    }
    else
    {
        // failed payment domain policy
    }
}
```

Can you see possible conditions related to Listing 12-9 that would annoy paying customers? If you need a clue, think about communication with external services. What would happen if the database was down when `Database.SavePaymentDetails()` was executed? The message would fail and be put back in the queue. But you've already charged the customer's credit card at this point. When the message is retried, the customer is charged again and again. This is precisely why you need a messaging gateway.

Essentially, messaging gateways split one big transaction in half, so if one of the calls fails, you don't repeat actions that have already been carried out (like charging a customer's credit card). In the

Billing bounded context, communicating with the payment provider is one-half and updating the database is the other. To implement a messaging gateway, you just need to add an extra message in-between them, which is the next step of this example.

Implementing a Messaging Gateway

As mentioned, messaging gateways are just a pattern; it's a case of splitting one message handler into two or more, where there is unreliable network communication. Therefore, there is no special facility provided by messaging frameworks—you just need to create and send additional messages. The real detail, though, is ensuring that you isolate the risky communication as much as possible.

Start by Defining the Messages

The first responsibility of the `OrderCreatedHandler` is communicating with the external supplier. You could fire an event indicating this has happened. However, because the messaging gateway is more of an implementation detail for fault tolerance and not a domain event, you don't want to expose it to other components. Therefore, it makes sense to go with a command to ensure it is only handled in a single place. In this example, the command is called `RecordPaymentAttempt`. You know from the component diagram that the `Billing.Payments.PaymentAccepted` component also needs to publish a `PaymentAccepted` event. So you can add both of those messages to a new C# class library called `Billing.Messages`. Remember to put the command in a `Commands` folder and the event in an `Events` folder, as Figure 12-17 illustrates. The format of those two messages is shown in Listing 12-10.

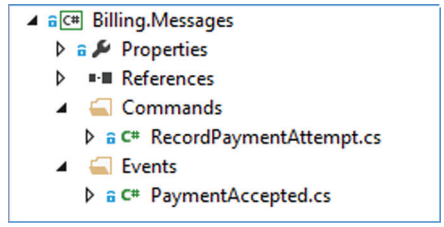


FIGURE 12-17: Adding the two messages to the Billing.Messages project.

LISTING 12-10: Billing Bounded Context's Messages

```
using System;

namespace Billing.Messages.Events
{
    public class PaymentAccepted
    {
        public string OrderId { get; set; }
    }
}

using System;

namespace Billing.Messages.Commands
{
    public class RecordPaymentAttempt
    {
        public string OrderId { get; set; }
```

(*continued*)

LISTING 12-10: *(continued)*

```
        public PaymentStatus Status { get; set; }
    }

    public enum PaymentStatus
    {
        Accepted,
        Rejected
    }
}
```

Then Implement Each Handler

You're now ready to add the robust `OrderPlacedHandler` that doesn't contain the vulnerabilities of the version in Listing 12-9. In this version, the call to the payment provider is wrapped with a `RecordPaymentAttempt` command. Saving the payment details to the database then comes in the next handler. Don't forget to add a reference to `Billing.Messages` in the `Billing.Payments.PaymentAccepted` component. You can then add the `OrderCreatedHandler` shown in Listing 12-11.

LISTING 12-11: Robust OrderCreatedHandler with a Messaging Gateway

```
using Billing.Messages.Commands;
using NServiceBus;
using Sales.Messages.Events;
using System;

    public class OrderCreatedHandler : IHandleMessages<OrderCreated>
    {
         // dependency injected by NServiceBus
         public IBus Bus { get; set; }

         public void Handle(OrderCreated message)
         {
             Console.WriteLine(
                 "Received order created event: OrderId: {0}",
                 message.OrderId
             );

             var cardDetails = Database.GetCardDetailsFor(
                 message.UserId
             );

             var conf = PaymentProvider.ChargeCreditCard(
                 cardDetails, message.Amount
             );

             var command = new RecordPaymentAttempt
             {
                 OrderId = message.OrderId,
                 Status = conf.Status
             };
```

```
                Bus.SendLocal(command);
            }
        }

        // dummy class for demo purposes only
        // beware of static dependencies
        public static class PaymentProvider
        {
            public static PaymentConfirmation ChargeCreditCard(
                              CardDetails details, double amount)
            {
                // to keep the example focused on what's relevant
                return new PaymentConfirmation
                           {
                               Status = PaymentStatus.Accepted
                           };
            }
        }

        public class PaymentConfirmation
        {
            public PaymentStatus Status { get; set; }
        }

        public static class Database
        {
            public static CardDetails GetCardDetailsFor(string userId)
            {
                // to keep the example focused on what's relevant
                return new CardDetails();
            }
        }

        public class CardDetails
        {
            // ...
        }
    }
```

One small detail to note is the `Bus.SendLocal()` call. Hopefully this makes sense, because the sending component also needs to handle the command. `SendLocal()` just sends the message to the same NServiceBus host that sent the message, where it then looks for a handler of that message.

To complete this step, add a handler for the `RecordPaymentAttempt` command that simulates recording the payment to the database. Listing 12-12 shows how this handler should look. You can put it in the same file as the `OrderCreatedHandler` if that helps you to mentally piece together the flow of messages.

LISTING 12-12: RecordPaymentHandler That Continues After the Messaging Gateway

```
using System;
using Billing.Messages.Commands;
```

(*continued*)

LISTING 12-12: *(continued)*

```
using NServiceBus;

namespace Billing.Payments.PaymentAccepted
{
    public class RecordPaymentAttemptHandler :
                 IHandleMessages<RecordPaymentAttempt>
    {
        // dependency injected by NServiceBus
        public IBus Bus { get; set; }

        public void Handle(RecordPaymentAttempt message)
        {
            Database.SavePaymentAttempt(
                message.OrderId, message.Status
            );

            if (message.Status == PaymentStatus.Accepted)
            {
                var evnt = new Billing.Messages.events.PaymentAccepted
                {
                    OrderId = message.OrderId
                };
                Bus.Publish(evnt);
                Console.WriteLine(
                    "Received payment accepted notification for Order:" +
                    " {0}. Published PaymentAccepted event",
                    message.OrderId
                );
            }
            else
            {
                // publish a payment rejected event
            }
        }

        public  static class Database
        {
            public static void SavePaymentAttempt(string orderId,
                                                  PaymentStatus status)
            {
                // .. save it to your favorite database
            }
        }
    }
}
```

Controlling Message Retries

Now is a great opportunity to really understand the value of using messaging. You're going to actually see the message bus retry messages because an external service is unavailable. In the process,

this example fully makes use of the messaging gateway you just implemented. What you need to do to simulate failure is change your stub payment provider implementation to throw an exception the first two times it is called and then happily accept the third request.

Simulating Failure

If you update the `PaymentProvider` so that it resembles Listing 12-13, you will be almost ready to see the fault tolerance of your reactive system in action.

LISTING 12-13: PaymentProvider Stubbed to Simulate External Failure

```
public static class PaymentProvider
{
    private static int Attempts = 0;

    public static PaymentConfirmation ChargeCreditCard(
                    CardDetails details, double amount)
    {
        if (Attempts < 2)
        {
            Attempts++;
            throw new Exception(
                "Service unavailable. Down for maintenance."
            );
        }

        return new PaymentConfirmation
                   {
                       Status = PaymentStatus.Accepted
                   };
    }
}
```

Before you tested your first message handler earlier in this chapter, you had to set each NServiceBus component to start up when the solution was started. You also need to do that now for the `Billing.Payment.PaymentAccepted` component so that you can test your messaging gateway. So ensure you're start-up settings resemble Figure 12-18.

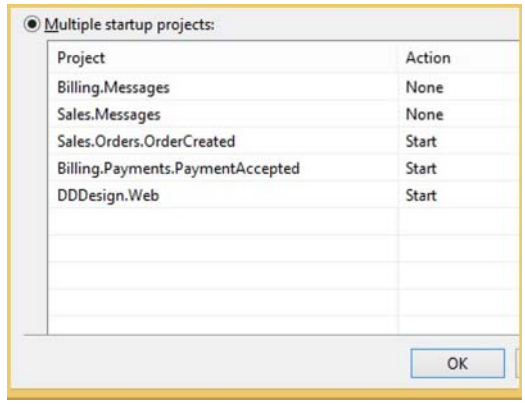


FIGURE 12-18: Setting all your NServiceBus servers to start up.

Now you can press F5 to run the application. After you've done that, navigate to the `/orders` page in your browser and place an order. Pay special attention to the console window for the `Billing.Payments.PaymentAccepted` component. At first you will just see that `OrderCreatedHandler` received the `OrderCreated` event. Below that you'll see that the `PaymentAccepted` event was published, as shown in Figure 12-19. But if you scroll up in the console

window, you actually see that was the third attempt. Figure 12-20 shows you the messages NServiceBus printed to the console when it failed to process the first message. Look how it even shows the exception type and message you threw in your stub payment provider: `System.Exception: Service unavailable. Down for maintenance.`

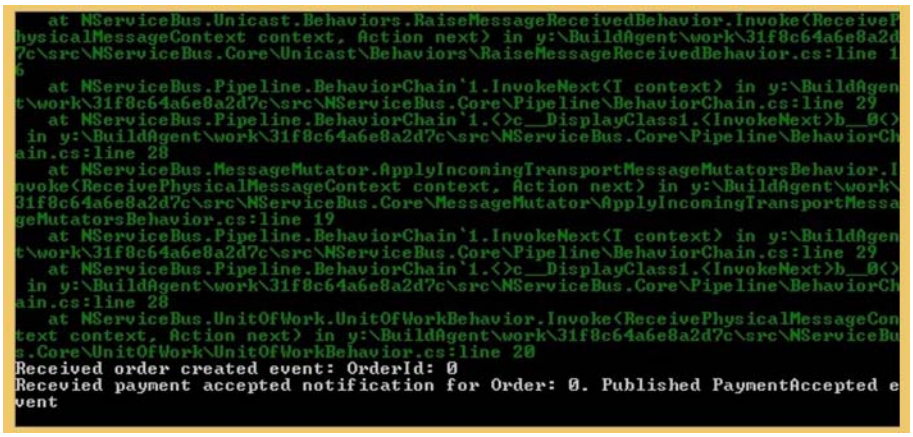


FIGURE 12-19: Looks like the payment was processed successfully.

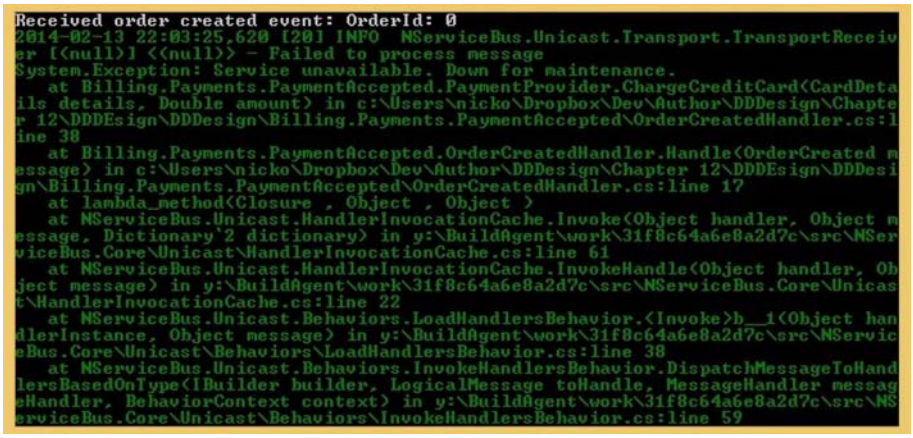


FIGURE 12-20: Actually, the payment failed twice first.

Second-Level Retries

If the payment provider is down for a long time, say ten minutes, NServiceBus may have reached the limit for the number of times it retries the message. Fortunately, NServiceBus has a feature called *2nd level retries*, allowing you to specify custom retry periods that cover longer durations. In this example, it would be useful to retry the message every ten minutes for an hour. The configuration in Listing 12-14 shows how to do that.

LISTING 12-14: NServiceBus Second-Level Retries Config

```
<configSections>
  <section name="SecondLevelRetriesConfig"
           type="NServiceBus.Config.SecondLevelRetriesConfig,
NServiceBus.Core"/>
</configSections>

<SecondLevelRetriesConfig Enabled="true"
                          TimeIncrease ="00:10:00"
                          NumberOfRetries="6" />
```

Sometimes messages will never succeed because the system cannot handle them. Perhaps a message wasn't validated properly, or data got wiped from the database that needed to be looked up to process the message. These are known as *poison messages* and inevitably end up in the `error` queue. You'll learn more about the `error` queue later in this chapter, including how to monitor it and handle messages that end up in it.

Eventual Consistency in Practice

At this point in the place order process, the system could be considered to be in an inconsistent state. Although the order was created by the Sales bounded context and payment has been processed by the Billing bounded context, the Shipping bounded context does not know this and hasn't arranged shipping yet. This would never happen in some systems, where the whole series is a single atomic transaction that succeeds or fails completely. So this can be considered an example of eventual consistency. It's an opportune moment for you to learn how to deal with eventual consistency when you are integrating bounded contexts using messaging.

Dealing with Inconsistency

Eventual consistency can lead to undesirable scenarios. For example, if a payment has been rejected, you can't just roll back the transaction and not create the order (as many non-eventually consistent systems would); the order was already created as part of a previous transaction in a different component and currently lives in that component's database. What you can do, though, is roll forward into a new state. You'd probably tell the customer the order could not be completed because payment failed. Ideally you would tell her immediately when she tries to place an order. However, you have to remember that you're trying to build a scalable fault-tolerant solution and you need to make sacrifices. Upsetting the few customers who cannot successfully place orders so that everybody else gets a superior user experience is often an acceptable trade-off.

When you are in an inconsistent state, you need to roll forward into a new state that represents the wishes of the business or the real-world domain processes you are modeling.

Rolling Forward into New States

To deal with the eventual consistency of a failed order, you can work with the business to create a new component diagram of the payment failed use case and use it to see what state you need to move into. Imagine that you did speak to the business and were told that when a payment is rejected, it is the Sales team's responsibility to inform the customer the order was cancelled. Figure 12-21 shows a partial component diagram for this use case.

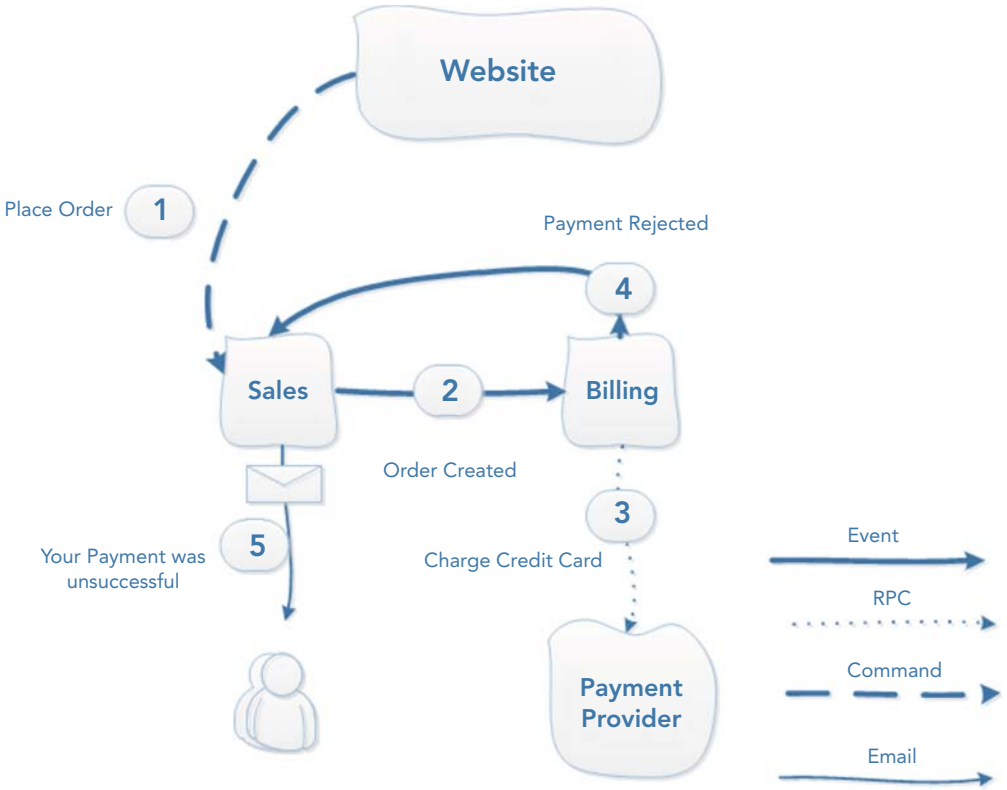


FIGURE 12-21: The payment rejected use case.

As you can see in Figure 12.21, all you need to do to deal with this eventually consistent scenario is publish a `PaymentRejected` event in the Billing bounded context and handle it in the Sales bounded context by sending an e-mail to the customer. To implement this use-case you just need to send and handle a few messages. To avoid repetition that is not shown here, you can implement it yourself if you want to using previous examples as a guide.

> **NOTE** *If you're still unsure about ditching transactions in favor of eventual consistency, Jimmy Bogard published a blog post in 2013 that tries to allay some common fears (*`http://lostechies.com/jimmybogard/2013/05/09/ditching-two-phased-commits/`*). It might be worth spending a quick five minutes reading and digesting it.*

Bounded Contexts Store All the Data They Need Locally

The next steps on your component diagram (remainder of 4, then 5, and 6) involve the Shipping bounded context arranging shipping. But there's a problem. Look at the `PaymentAccepted` event again:

```
public class PaymentAccepted
        {
        public string OrderId { get; set; }
        }
```

How can you arrange shipping when all you have is the ID of an order? Unfortunately, you can't, so you need to carefully consider your options. To keep things simple, imagine that each user has just one address and it lives inside the database belonging to the `Sales.Customers` business component. One option is to RPC across to the `Sales.Customers` business component using HTTP to get the data you need. But then you've introduced a temporal coupling. If the `Sales.Customers` database or RPC endpoint goes down, the Shipping bounded context cannot function. If other bounded contexts went straight to the `Sales.Customers` database, the Sales development team would not be able to refactor or change the database according to their own best needs without worry of breaking other bounded contexts.

Another option is to publish the customer's address on the `PaymentAccepted` event. This approach is also dangerous because, essentially, you are aggregating two messages by adding the properties from the first message to the second. In a use case in which four messages are sent, the fourth message needs to contain all the properties of the first three. This could cause high coupling and hard-to-debug issues trying to work out where data came from. Being pragmatic, you could try to get away with it in a few places, but it's highly unrecommended unless you have a compelling business case.

Storage Is Cheap—Keep a Local Copy

An option that many teams take in this situation to reduce unwanted coupling is to store all the data each bounded context needs locally. This is often a good trade-off because storage is cheap these days. As an example, you can buy a 1TB hard drive for around $50 at the time of writing. What this means to you is that both the Sales bounded context and the Shipping bounded context should store the customer's address. Shortly you'll see some of the concerns people have with this, but for now just try to focus on implementing the Shipping bounded context using this approach. Those concerns will be addressed afterward.

Figure 12-22 illustrates the user registration process. Notice how an event is published—`NewBusinessUserRegistered`—and two bounded contexts subscribe to this event. The Shipping bounded context's Regular Customers business component takes the customer's ID and address. The Marketing bounded context's Online Marketing business component takes the customer's ID address, and the market the business operates in. Both bounded contexts store the data locally, even though it's the same initial data being stored in multiple places.

> **NOTE** *Implementing the Shipping bounded context involves the same general process as implementing the Billing bounded context, so some of the precise details are omitted. Look back to the previous sections if you need reminders. You're more than welcome to post your question on the Wrox discussion forum if you need further assistance.*

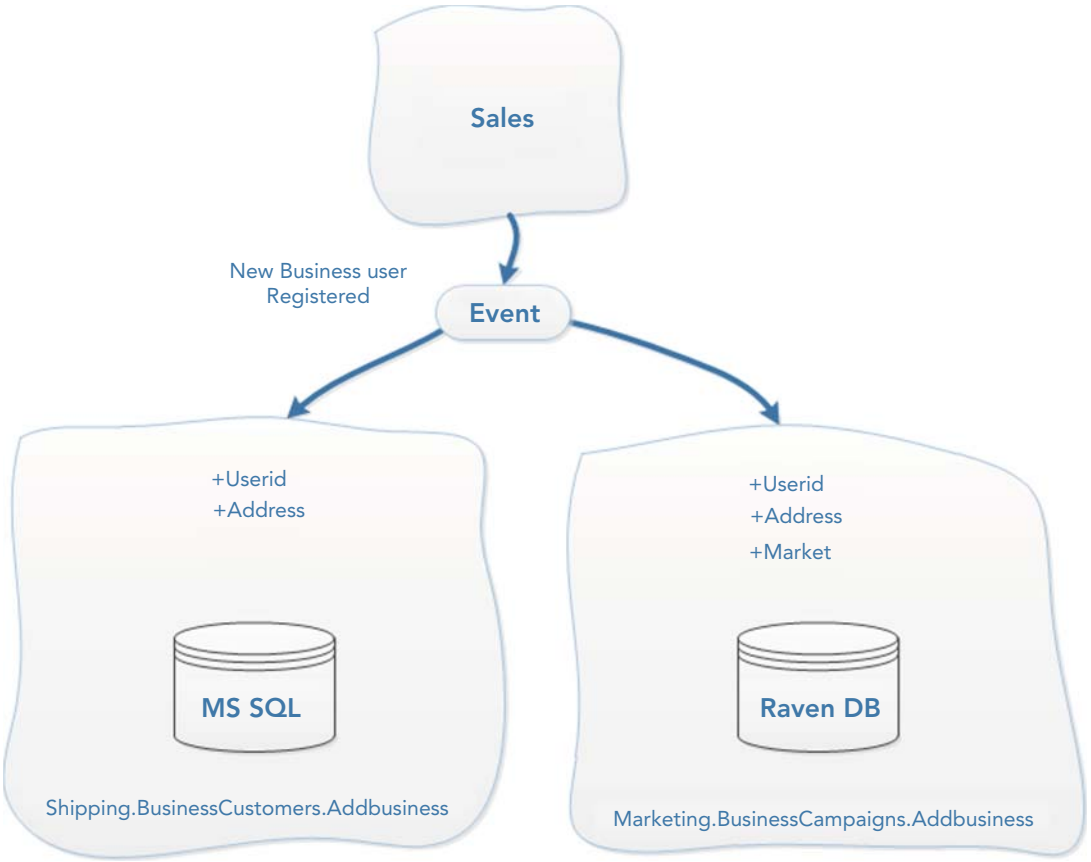


FIGURE 12-22: Multiple bounded contexts store the same piece of data locally.

To implement the remaining part of the e-commerce system you've been building in this chapter, you'll use local storage. Before that, you need to create the `Shipping.BusinessCustomers.ArrangeShipping` component that subscribes to the Sales bounded context's `OrderCreated` event and the Billing bounded context's `PaymentAccepted` event. You can get to that stage by carrying out the following steps:

1. Create a new C# class library called `Shipping.BusinessCustomers.ShippingArranged`.
2. Add a reference to the `Sales.Messages` and `Billing.Messages` projects.
3. Add the NServiceBus dependencies using the NuGet Package Manager.
4. Copy across the conventions for locating commands and events into `EndpointConfig`. Also make `EndpointConfig` inherit `AsA_Publisher` and `IWantCustomInitialization`, as Listing 12-15 illustrates.
5. Add an `OrderCreatedHandler` and `PaymentAcceptedHandler` to the project, as shown in Listing 12-16.

6. Add subscription configuration for the `OrderCreated` and `PaymentAccepted` events to the project's `App.config`. Your `App.config` should then resemble Listing 12-17.
7. Configure the project to start up.
8. Add a C# class library called `Shipping.Messages` with an `Events` folder containing the `ShippingArranged` event shown in Listing 12-18.
9. Add a reference to the `Shipping.Messages` project in the `Shipping.BusinesCustomers.ArrangeShipping` component.
10. Configure the `Shipping.BusinessCustomers.ArrangeShipping` project to start up in Visual Studio.

LISTING 12-15: EndpointConfig for the ShippingArranged Subcomponent

```
using NServiceBus;

namespace Shipping.BusinessCustomers.ShippingArranged
{
    public class EndpointConfig : IConfigureThisEndpoint,
                AsA_Server, AsA_Publisher, IWantCustomInitialization
    {
        public void Init()
        {
            Configure.With()
                     .DefiningCommandsAs(t => t.Namespace != null
                          && t.Namespace.Contains("Commands"))
                     .DefiningEventsAs(t => t.Namespace != null
                          && t.Namespace.Contains("Events"));
        }
    }
}
```

LISTING 12-16: OrderCreatedHandler and PaymentAcceptedHandler

```
using Billing.Messages.Events;
using NServiceBus;
using Sales.Messages.Events;
using System;
using System.Linq;
using System.Collections.Generic;

namespace Shipping.BusinessCustomers.ShippingArranged
{
    public class OrderCreatedHandler : IHandleMessages<OrderCreated>
    {
        // dependency injected by NServiceBus
        public IBus Bus { get; set; }

        public void Handle(OrderCreated message)
        {
            Console.WriteLine(
```

(*continued*)

LISTING 12-16: *(continued)*

```
            "Shipping BC storing: Order: {0} User: {1} " +
            "Shipping Type: {2}",
            message.OrderId, message.UserId, message.ShippingTypeId
        );
        var order = new ShippingOrder
        {
            UserId = message.UserId,
            OrderId = message.OrderId,
            ShippingTypeId = message.ShippingTypeId
        };
        ShippingDatabase.AddOrderDetails(order);
    }
}

public class PaymentAcceptedHandler : IHandleMessages<PaymentAccepted>
{
    // dependency injected by NServiceBus
    public IBus Bus { get; set; }

    public void Handle(PaymentAccepted message)
    {
        var address = ShippingDatabase.GetCustomerAddress(
            message.OrderId
        );

        var confirmation = ShippingProvider.ArrangeShippingFor(
            address, message.OrderId
        );

        if (confirmation.Status == ShippingStatus.Success)
        {
            var evnt = new Shipping.Messages.events.ShippingArranged
            {
                OrderId = message.OrderId
            };
            Bus.Publish(evnt);

            Console.WriteLine(
                "Shipping BC arranged shipping for Order:" +
                " {0} to: {1}",
                message.OrderId, address
            );
        }
        else
        {
            // ..
        }
    }
}

public static class ShippingDatabase
```

```
    {
          private static List<ShippingOrder> Orders =
    new List<ShippingOrder>();

        public static void AddOrderDetails(ShippingOrder order)
        {
            Orders.Add(order);
        }
        public static string GetCustomerAddress(string orderId)
        {
            /*
             * Implement this properly by storing the user's address
             * when the Sales bounded context publishes an event
             * with their details (add that message yourself, too)
             */
            var order = Orders.Single(o => o.OrderId == orderId);
                return string.Format(
                 "{0}, 55 DDDEsign Street",
                 Order.UserId
            );
       }
    }

    public class ShippingOrder
    {
        public string UserId { get; set; }

        public string OrderId { get; set; }

        public string ShippingTypeId { get; set; }

     }

    public static class ShippingProvider
    {
        public static ShippingConfirmation ArrangeShippingFor(
                           string address, string referenceCode)
        {
            return new ShippingConfirmation
                       {
                           Status = ShippingStatus.Success
                       };
        }
    }
    public class ShippingConfirmation
    {
        public ShippingStatus Status { get; set; }
    }
    public enum ShippingStatus
    {
        Success,
        Failure
    }
}
```

LISTING 12-17: App.config for the ShippingArranged Subcomponent

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<configuration>
  <configSections>
    <section name="MessageForwardingInCaseOfFaultConfig"
type="NServiceBus.Config.MessageForwardingInCaseOfFaultConfig,
NServiceBus.Core" />
    <section name="UnicastBusConfig"
             type="NServiceBus.Config.UnicastBusConfig,
NServiceBus.Core" />
    <section name="AuditConfig"
             type="NServiceBus.Config.AuditConfig,
NServiceBus.Core" />
  </configSections>
  <MessageForwardingInCaseOfFaultConfig ErrorQueue="error" />
  <UnicastBusConfig>
    <MessageEndpointMappings>
      <add Messages="Sales.Messages"
           Type="Sales.Messages.OrderCreated_V2"
           Endpoint="Sales.Orders.OrderCreated" />
      <add Messages="Billing.Messages"
           Type="Billing.Messages.PaymentAccepted"
           Endpoint="Billing.Payments.PaymentAccepted" />
    </MessageEndpointMappings>
  </UnicastBusConfig>
  <AuditConfig QueueName="audit" />
</configuration>
```

LISTING 12-18: The ShippingArranged Event

```
using System;

namespace Shipping.Messages
{
    public class ShippingArranged
    {
        public string OrderId { get; set; }

        /*
         * Other fields, such as date/date range,
         * could be added here depending on your
         * shipping provider(s) API
         */
    }
}
```

If you've done everything correctly, your system should now be working up to the point of arranging shipping. Try running the application by pressing F5, navigating to the `/orders` page, and filling out the form. You then see three console windows appear—one for each component. Inside the `Shipping.BusinessCustomer.ArrangeShipping` console you should see that the `ArrangeShipping`

component stored the `UserId` and `OrderId` from the Sales bounded context's `OrderCreated` event, and it then used the `UserId` to look up the address when it handled the `PaymentAccepted` event. You should see console output that looks like Figure 12-23 (with the values you entered into the form).

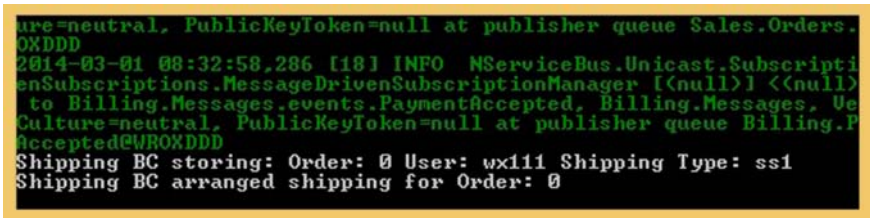


FIGURE 12-23: Shipping bounded context storing data locally and using it to arrange shipping.

Common Data Duplication Concerns

Now that you've seen how to store data locally to each business component, it's time to go over some of the concerns that people express when confronted with this considerable change in philosophy. A lot of people raise the issue of data being duplicated because they are concerned with how inefficient it may be. Others question how you can handle data updates, such as a customer updating his address, when there are duplicate copies scattered around. Messaging systems in particular raise the concern of messages arriving out of order. Imagine that a user signs up and then places an order immediately. What would happen if the Shipping bounded context was asked to arrange shipping before it had stored the user's address?

Remember, storing data locally is not ideal. But it often reduces coupling, providing a better platform for scalability, fault tolerance, and rate of development. Having central points of control for logic or data in a distributed system can be dangerous for these reasons. But the concerns raised are serious.

The Data Is Conceptually Different

Two business components may store the price of a product. For example, the Billing bounded context will store the latest price so that when users place an order, they are charged the current price. But the Sales department will probably want to store the price of a product when the order was placed. Think about purchases you make online yourself. When you ask for a refund and you have a receipt, you get the money you paid at the time, not the latest price. This is an example of why, even though bounded contexts appear to duplicate the same data, conceptually the data is different—it's used for different purposes, and it changes for different reasons.

Avoid Inconsistencies by Using Validity Periods

Businesses often change the price of products for special promotions, lack of popularity, or newer alternatives appearing on the market. When this happens, there is the opportunity for one bounded context to update the price before another. A problem arising from this is that users may see one price on the website but be charged a different price when they add it to their basket. This common concern can easily be addressed by adding validity periods to a message. The following snippet

shows a `PriceUpdated` event with `AvailableFrom` and `AvailableTo` fields that indicate the period of time that the price in this message is valid for:

```
public class PriceUpdated
{
    public string ProductId { get; set; }

    public double Price { get; set; }

    public DateTime AvailableFrom { get; set; }

    public DateTime AvailableTo { get; set; }
 }
```

When sending messages with validity periods, it's best to send them as soon as you can. The more notice you give to each bounded context, the more time it has to prepare for the new message taking effect. Often you can align your strategy with business rules. In this example, you could talk to the business and ask how much notice it gets for price changes. You could then agree with the business that you can guarantee all prices can be up to date if you get at least 24 hours' notice, for example.

The Economics of Duplicating Data

You've just seen that conceptually the data is different, so the word *duplication* is semantically incorrect in some cases. But whether it is financially cost effective is based on a number of factors you need to consider. It was mentioned previously that a 1TB hard drive costs around $50. The bigger your bulk purchase, the more cost efficient this becomes. Cloud storage is also incredibly cheap.

It's what you don't pay for that contains hidden costs, though. How much would you lose by having software components and teams that were tightly coupled to the same data? Think about your scalability needs; try to imagine how much quicker you could build your system by having isolated teams. It's difficult to predict these costs, but considering that storage is so cheap, teams are normally happy to accept the costs of duplicating data so they can focus on delivering business value faster.

Messages Arriving Out of Order

What would happen if a user signed up and placed an order, but the Shipping bounded context received the `PaymentAccepted` notification before it received the user's address? In this kind of situation, you can simply put the message back on the queue and retry it again later, by which time the message containing the address will have been processed. In general, this is how to deal with out of order messages.

Pulling It All Together in the UI

Pushing data into a messaging system, as you've seen so far in this chapter, is only part of the story. You also need to be able to get data out of the messaging system. Again, this can require a different mind-set compared to some traditional practices, especially when you are storing data locally in each business component. You saw a clue about how to achieve this in the containers diagram illustrated in Figure 12-4. The answer is that bounded contexts often need to expose their data over HTTP for web applications to fetch.

In the next chapter, you will see lots of good practices for, and concrete examples of, building HTTP APIs to expose your domain functionality and data. In this section, you'll learn about a few of the concepts and trade-offs that are especially important in messaging systems. Many of the other concepts from the next chapter will still be relevant to some extent, though.

> **NOTE** *Chapter 23, "Architecting Application Users Interfaces," goes into more detail about building UIs that sit on top of HTTP APIs and messaging systems. It contains a number of practical examples.*

Business Components Need Their Own APIs

If you work to the principle that each business component has its own private database, the only sensible way to share data with a web application is for each business component to have its own set of APIs exposing that data. Figure 12-24 visualizes this suggested layout.

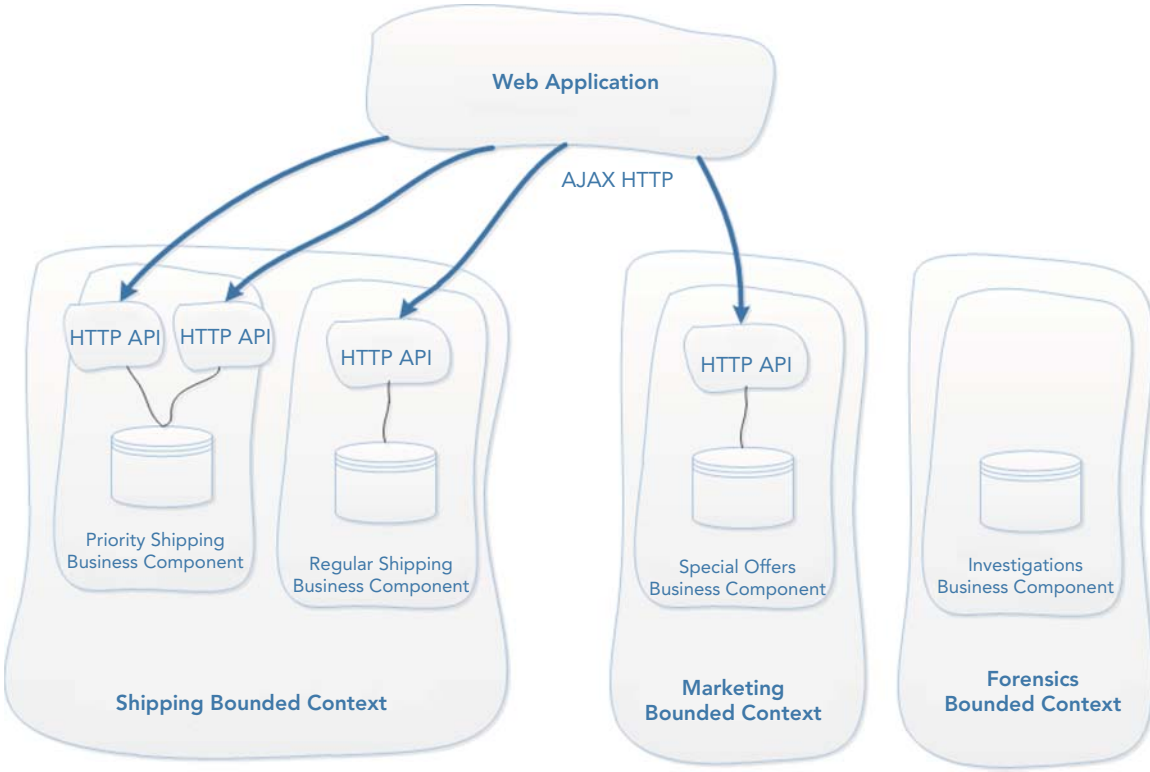


FIGURE 12-24: Business components have their own APIs.

As you can see in Figure 12-24, some business components have chosen to have two web applications exposing their data. In contrast, other business components do not provide APIs because they do not expose data; they simply consume it and carry out other tasks. Inside your business components, you are free to do as you please. The number of projects, the number of APIs, and even the technologies used are down to your best judgment and the needs of the project.

Be Wary of Server-Side Orchestration

It can be tempting to query all your APIs on the server (inside a controller) and send a combined model to the web page. Unfortunately, this can undo a lot of the hard work of building a loosely coupled, fault-tolerant system, because you have introduced another coupling point. First, if one of the APIs is down, there may be an error and the page may not render. Think carefully, though. Is that really necessary? Imagine that you built a page showing a catalogue of products, at the bottom of which page you showed special offers. Figure 12-25 exemplifies this by indicating how different parts of the page get their data from different APIs.

Do you think the business would want customers to see an error screen preventing them from spending money if the Marketing bounded context was experiencing issues and could not provide the special offers? Some businesses would definitely prefer you to still show the main list of products, even if you couldn't show the special offers. You could do this with server-side orchestration, but it requires more effort and can easily be forgotten. Also, the controller in your web application that performs the orchestration is an extra component that could fail. Some teams prefer to load the data directly from within the page using AJAX for these reasons.

UI Composition with AJAX Data

Looking back at Figure 12-25, you can see that each part of the page pulls in the relevant bits of data it needs from the different APIs. So why not just make AJAX web requests directly in the page and remove the need for server-side orchestration? Your APIs can return lightweight JSON, while your web pages use your favorite JavaScript libraries for managing and presenting the data. This style of UI composition can work especially well when you are trying to build fast, singe-page applications (SPAs) or generic APIs that are used by many other websites.

UI Composition with AJAX HTML

Instead of returning data, such as JSON or XML, from your APIs, you can just return HTML that is directly rendered on the page. Proponents of this style of UI composition tend to prefer the ability of each bounded context to decide how certain parts of the page are rendered. Looking back at Figure 12-25, for example, if the Marketing bounded context returned HTML, it could control how the special offers section of the page appeared. It could even restyle the special offers section of the page whenever it wanted just by returning updated HTML.

Although UI composition with HTML gives extra control to each bounded context, it can be hard to manage the consistency of the page that would normally be easy to achieve when the HTML lived in a single file. This approach is also difficult to apply when multiple web applications share the API. But it's certainly an option that some teams use, and you should at least consider the advantages of applying it on your projects.

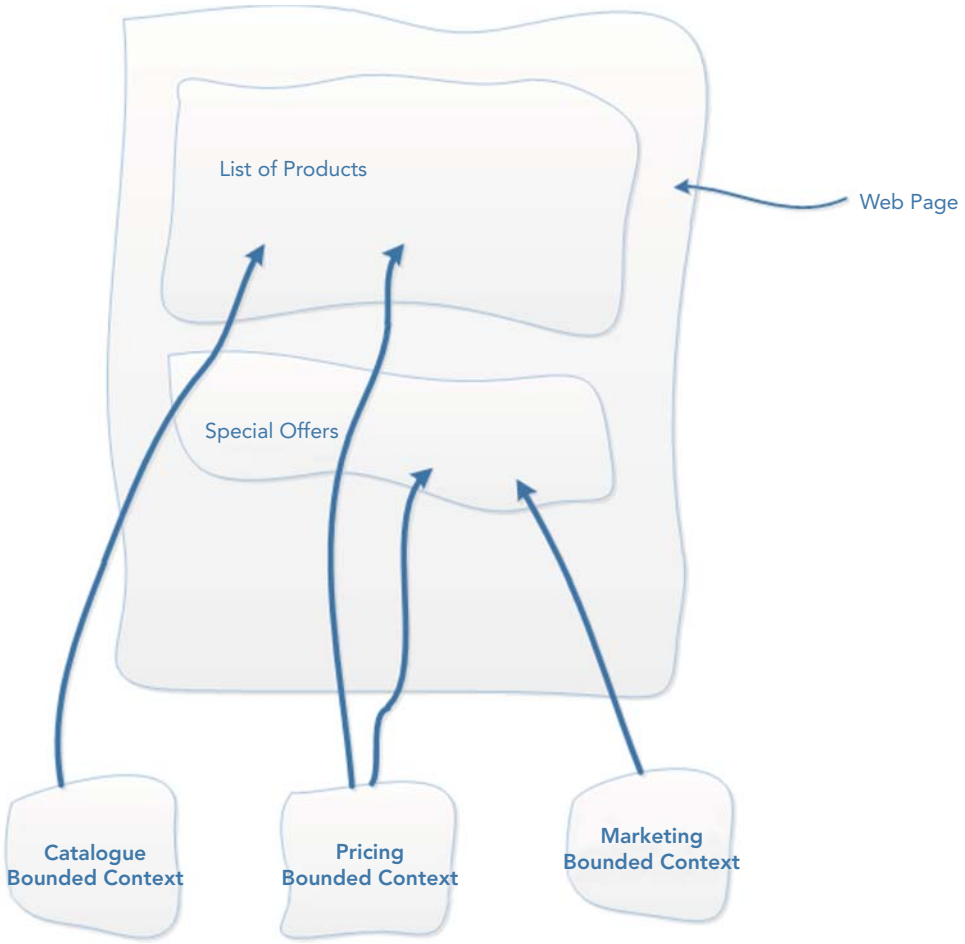


FIGURE 12-25: Pages get their data from multiple APIs.

Sharing Your APIs with the Outside World

APIs aren't always for internal use. Sometimes they are used by multiple applications that you don't own. For some companies, an API is even the main product. You will learn in the next chapter how to create HTTP APIs, using patterns like REST, to expose them to external consumers.

MAINTAINING A MESSAGING APPLICATION

Once you've built a messaging system, there are still big challenges you need to deal with. You'll need to monitor the system to ensure it is behaving well and supporting the needs of the business. You'll want to see how well the application is performing, and you'll definitely want to find any errors that are occurring so you can fix them and prevent lost revenue. Importantly, you will need

to put in place a strategy for dealing with development changes that affect multiple teams, such as change in the structure of a domain event.

Message Versioning

At some point, you will almost certainly be required to change the format of a message. But the message format is the contract between two teams. Any changes to the format may require input and collaboration from both teams. Although these scenarios are inevitable, one of the goals of loosely coupled Service Oriented Architecture (SOA) systems is that teams don't slow each other down. Changing the contract between multiple bounded contexts is a situation in which this could happen.

If you were to just change the format of a message and deploy it, the systems that had not upgraded to the new format wouldn't work anymore. What you need is a strategy that allows you to update message formats in a way that doesn't break existing consumers. This would then allow all the teams involved to update to the new format when it was important for them to do so. Until then, they could use the old format. This is why you need to aim for backward compatibility when you change message formats.

Backward-Compatible Message Versioning

Maintaining backward compatibility with your messages can be an important step in ensuring that integration between bounded contexts remains intact. It can also be important in enabling each bounded context to be developed independently, without teams having to have lots of meetings and schedule a simultaneous rollout when the message format changes. Consider the `OrderCreated` event from earlier in the chapter:

```
public class OrderCreated
{
    public string OrderId { get; set; }

    public string UserId { get; set; }

    public string[] ProductIds { get; set; }

    public string ShippingTypeId { get; set; }

    public DateTime TimeStamp { get; set; }

    public double Amount { get; set; }
}
```

Imagine that a new authorization came in from the business permitting customers to have more than a single address. All the business's online competitors allow this, so the business really needs to establish feature parity. To enable the new functionality, the `OrderCreated` event needs to contain an `AddressId`. For many messaging frameworks, this is a breaking change, meaning that adding this new field to the contract will break existing subscribers who do not update to the latest version.

Look at the situation a little more closely, though. Both the Shipping and Billing bounded contexts subscribe to the event. It's important that the Shipping bounded context knows the new address so the order can be sent there. But does the Billing bounded context need to know that? In some domains, it probably doesn't care, because it only sends bills to the customer's e-mail address. Even

if it did send paper mail, it would send it to the billing address, not the shipping address. So is there any need to divert the Billing team away from adding important business value, just to accommodate a new message format that they don't care about? In many cases there's not, and backward-compatible message versioning saves teams that don't care about the new format from wasting valuable time on technical issues that don't add business value.

Messaging frameworks can vary in how they let you define and update message formats. You'll now see how to achieve backward-compatible message versioning with NServiceBus, but the concepts will be loosely applicable to any messaging framework you use.

Handling Versioning with NServiceBus's Polymorphic Handlers

In this section, you'll make the `OrderCreated` event backward compatible so that the Shipping bounded context uses the new version, whereas the Billing bounded context uses the existing version. This involves just a few short steps:

1. Create a new version of the message that inherits the original version.
2. Update handlers that require the new version to handle the new version.
3. Add subscription details for the new message in each subscriber's `App.config`.
4. Update the sender to send the new version of the message.

No changes are necessary to a service that doesn't care about the new version (they will still get messages in the old format).

In this example, carrying out these steps involves creating an `OrderCreated_V2` event that inherits from the `OrderCreated` event, updating `Shipping.BusinessCustomers.ArrangeShipping` to handle the new event, and finally sending an `OrderCreated_V2` event from the Sales bounded context.

First, add the `OrderCratedEvent_V2` event in the same folder as the original `OrderCreated` event. It should look like Listing 12-19. Second, your updated `OrderCreatedHandler` should look something like Listing 12-20, taking advantage of the new message format, and using the `AddressID` supplied in the message. You'll also need to update `App.config` in `Shipping.BusinessCustomers.ShippingArranged` to identify where the new `OrderCreated_V2` event will be published from, as per Listing 12-21. Finally, you can send an `OrderCreated_V2` event, as shown in Listing 12-22.

LISTING 12-19: OrderCreated_V2 Event

```
using System;

namespace Sales.Messages.Events
{
    public class OrderCreated_V2 : OrderCreated
    {
        public string AddressId { get; set; }
    }
}
```

LISTING 12-20: OrderCreated Handler Now Handling OrderCreated_V2 Event

```
// update: handling V2 messages
public class OrderCreatedHandler : IHandleMessages<OrderCreated_V2>
{
    // dependency injected by NServiceBus
    public IBus Bus { get; set; }

    // update: accepts a V2 message
    public void Handle(OrderCreated_V2 message)
    {
        Console.WriteLine(
            "Shipping BC storing: Order: {0} User: {1} Address: {3}" +
            "Shipping Type: {2}",
            message.OrderId, message.UserId, message.ShippingTypeId,
            message.AddressId
        );
        var order = new ShippingOrder
        {
            UserId = message.UserId,
            OrderId = message.OrderId,
            AddressId = message.AddressId,
            ShippingTypeId = message.ShippingTypeId
        };
        ShippingDatabase.AddOrderDetails(order);
    }
}

public class PaymentAcceptedHandler : IHandleMessages<PaymentAccepted>
{
    // dependency injected by NServiceBus
    public IBus Bus { get; set; }

    public void Handle(PaymentAccepted message)
    {
        var address = ShippingDatabase.GetCustomerAddress(
            message.OrderId
        );
        var confirmation = ShippingProvider.ArrangeShippingFor(
            address, message.OrderId
        );
        if (confirmation.Status == ShippingStatus.Success)
        {
            var evnt = new Shipping.Messages.Events.ShippingArranged
            {
                OrderId = message.OrderId
            };
            Bus.Publish(evnt);
            Console.WriteLine(
                "Shipping BC arranged shipping for Order: {0} to: {1}",
                message.OrderId, address
            );
        }
```

```
        else
        {
            // .. notify failed shipping instead
        }
    }
}

public static class ShippingDatabase
{
    private static List<ShippingOrder> Orders = new List<ShippingOrder>();

    public static void AddOrderDetails(ShippingOrder order)
    {
        Orders.Add(order);
    }

    public static string GetCustomerAddress(string orderId)
    {
        var order = Orders.Single(o => o.OrderId == orderId);

        // in reality you would look up customer's address here
        // that was saved during an AddressCreated event or similar
        return string.Format(
            "{0}, Address ID: {1}",
            order.UserId, order.AddressId
        );
    }
}

public class ShippingOrder
{
    public string UserId { get; set; }

    public string OrderId { get; set; }

    public string ShippingTypeId { get; set; }

    public string AddressId { get; set; }
}
```

LISTING 12-21: Updated App.config Section Now Subscribing to OrderCreated_V2

```
<UnicastBusConfig>
    <MessageEndpointMappings>
      <add Messages="Sales.Messages"
           Type="Sales.Messages.OrderCreated_V2"
           Endpoint="Sales.Orders.OrderCreated" />
      <add Messages="Billing.Messages"
           Type="Billing.Messages.PaymentAccepted"
           Endpoint="Billing.Payments.PaymentAccepted" />
    </MessageEndpointMappings>
  </UnicastBusConfig>
```

LISTING 12-22: Updated PlaceOrderHandler Now Sending OrderCreated_V2

```
public class PlaceOrderHandler : IHandleMessages<PlaceOrder>
{
    public IBus Bus { get; set; }

    public void Handle(PlaceOrder message)
    {
        var orderId = Database.SaveOrder(
            message.ProductIds, message.UserId,
            message.ShippingTypeId
        );

        Console.WriteLine(
            "Created order #{3} : Products:{0} with shipping: {1}" +
            " made by user: {2}",
            String.Join(",", message.ProductIds),
            message.ShippingTypeId, message.UserId, orderId
        );

        // update: sending a V2 message now
        var orderCreatedEvent =
            new Sales.Messages.Events.OrderCreated_V2
        {
            OrderId = orderId,
            UserId = message.UserId,
            ProductIds = message.ProductIds,
            ShippingTypeId = message.ShippingTypeId,
            TimeStamp = DateTime.Now,
            Amount = CalculateCostOf(message.ProductIds),
            // Feel free to implement this fully yourself
            AddressId = "AddressID123"
        };

        Bus.Publish(orderCreatedEvent);
    }

    private double CalculateCostOf(IEnumerable<string> productIds)
    {
        // database lookup, etc.
        return 1000.00;
    }
}
```

If you run the application now, the Billing bounded context should work as it used to, handling the original message format (because an `OrderCreated_V2` is still an `OrderCreated`). The Shipping bounded context incorporates the new address logic using the same `OrderCreated` event but cast as an `OrderCreated_V2`. One thing to notice is that the address used for the order will not be sent with the `OrderCreated` event. It will be sent from another event, such as `UserAddedAddress`. The `Shipping.BusinessCustomers` business component needs to subscribe to this event and store the address locally.

> **NOTE** *For further examples of polymorphic message versioning, take a look at the NServiceBus documentation (*`http://support.nservicebus.com/customer/portal/articles/894151-versioning-sample`*).*

Monitoring and Scaling

When a system is up and running, it's important to keep any eye on errors to ensure that users are getting an acceptable level of service. It's also important to measure SLAs and business metrics so that the business can use them as the basis for making key decisions, such as how to evolve the product or the business model. Measuring SLAs also informs the development team when the system is too slow and it's time to consider scaling. You'll now see how to handle errors, measure technical and business metrics, and scale bounded contexts using NServiceBus. These concepts should apply to other messaging frameworks, too, although they will be implemented differently, of course.

Monitoring Errors

You saw earlier in this chapter that messages that are not successfully delivered or processed are retried. In that example, though, you just added some logic to ensure the payment provider only failed twice. This is known as a transient error, because the situation resolves itself after some time. However, sometimes errors are not transient, and no matter how many times a message is retried, it always fails. These are poison messages (discussed in more detail at `http://msdn.microsoft.com/en-us/library/ms166137.aspx`). At some point, your messaging framework will stop retrying poison messages. Most messaging frameworks then put them in a special queue called the `error` queue, where manual intervention is required to remove the messages.

You can see the `error` queue in action by updating one of your message handlers to always throw an exception. Try replacing the logic in your `Sales.Orders.OrderCreated.PlaceOrderHandler`'s `Handle()` method so that it simulates a permanent error condition as follows:

```
public void Handle(PlaceOrder message)
{
    throw new Exception("I have received a poison message");
}
```

Running your application and placing orders now causes the message to repeatedly fail and ultimately end up in the `error` queue of the component that sends the poison message. (You need to wait for a red error message to appear in the console.) To confirm this, you need to activate the Server Explorer in Visual Studio, expand `Servers`, expand the name of your machine, and then expand `Message Queues`. Finally, you can expand the `Private Queues` node and examine the contents of the `error` queue, as shown in Figure 12-26.

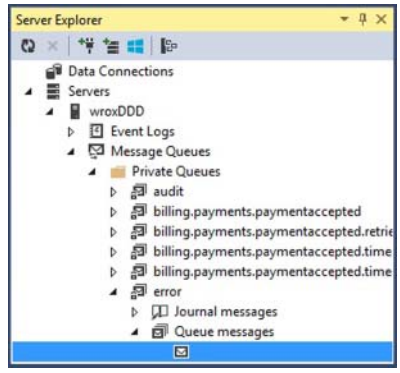


FIGURE 12-26: Locating the error queue in Visual Studio's Server Explorer.

To inspect all the messages in the `error` queue, expand the `Queue messages` node. If you want to inspect the contents of a single message, right-click on it and choose Properties,

highlight the `BodyStream` row, and click the button with three dots that appears in the right column. Figure 12-27 shows an example of viewing the `BodyStream` value of a message in the `error` queue.

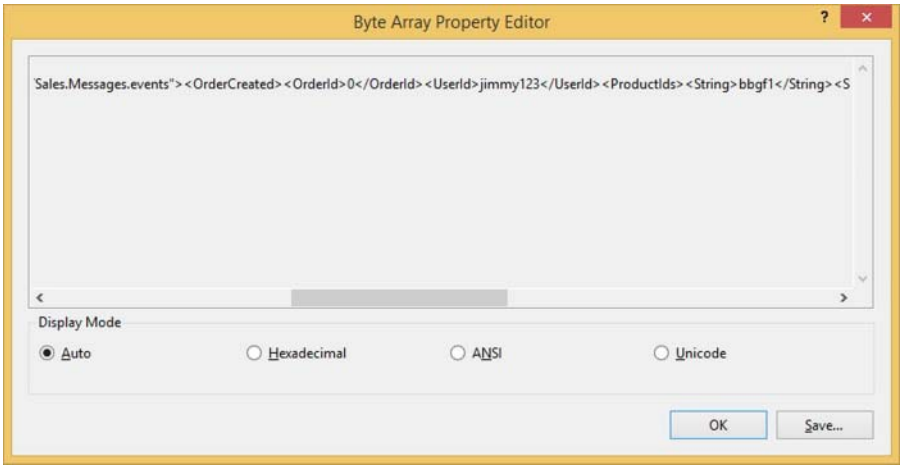


FIGURE 12-27: Viewing the contents of a message in the error queue.

Monitoring the `error` queue is an important activity that lets you spot production bugs quickly. But how you actually monitor the queue is up to you. Fortunately, you have a number of choices. For example, you can use the Open Source Wolfpack monitoring tool (`http://wolfpack.codeplex.com/`) or even create your own monitoring system.

Once a message is put in an `error` queue, manual intervention is required to determine what happens to it next. After inspecting the message, you might realize that the problem has gone away and the message should be fine (it was really a transient error), or there is an error in the code, meaning the message will never successfully be handled. Either way, when the problem has been fixed, NServiceBus ships with a tool that sends all messages back to their queue so they can be attempted again (`ReturnToSourceQueue.exe`).

You can learn more about monitoring and dealing with errors from the NServiceBus documentation (`http://support.nservicebus.com/customer/portal/articles/860511-getting-started—fault-tolerance`).

Monitoring SLAs

How many orders were placed in the past two hours? How long does it take an order to be processed? How long is the external payment provider taking to charge credit cards? These are all questions that can be answered by measuring the rate and frequency of messages flowing through your system. They're important for helping the business make decisions, and they're important for helping the tech team remove bottlenecks and improve capacity planning.

It should be possible to capture metrics and feed them into your own analytics systems with your preferred choice of technologies. With MSMQ and NServiceBus, you get a lot of monitoring goodies

out of the box. Both technologies provide additional performance counters, which you can see in the NServiceBus documentation (`http://support.nservicebus.com/customer/portal/articles/859446-monitoring-nservicebus-endpoints`). You can then import these metrics into your preferred monitoring system using Windows Management Instrumentation (`http://msdn.microsoft.com/en-us/library/aa310909(v=vs.71).aspx`).

Scaling Out

When your metrics are telling you that messages are spending a long time in the queue, and the businesses are telling you that customers are complaining about slow turnaround times, you need to scale the system to process messages faster. Because it's inefficient to keep buying bigger servers, you often need to add more servers and spread the load among them. In a messaging system, this means multiple instances of the same component feeding off the same queue. Unfortunately, this represents a slight problem, because NServiceBus creates queues on the machines that are processing the messages and won't allow the queues to be accessed from other machines. However, NServiceBus has a load balancer-like solution for these scenarios called the distributor. You should be able to get going by quickly reading the official documentation for it (`http://support.nservicebus.com/customer/portal/articles/859556-load-balancing-with-the-distributor`). Should you choose another messaging framework, it will probably have similar solutions for scaling.

Before throwing more hardware at the problem, NServiceBus does have a setting that may allow it to run faster on a single machine. The `MaximumConcurrencyLevel` setting on the `TransportConfig` node, as shown in the following snippet, controls the number of threads NServiceBus will use. The following snippet shows four threads being granted to NServiceBus:

```
<TransportConfig MaxRetries="5" MaximumConcurrencyLevel="4" />
```

INTEGRATING A BOUNDED CONTEXT WITH MASS TRANSIT

You've chosen messaging because it gives you lots of freedom and loose coupling in your architecture. But then you find out you're tied to NServiceBus, which is actually another form of coupling (platform coupling). This can be a problem if you want to build new systems in different technologies or incorporate existing systems that don't run on .NET. Platform coupling in messaging systems is a well-known problem with a well-known solution: a messaging pattern called the messaging bridge.

In this section, you'll see how you can use a messaging bridge to integrate another service that employs a different messaging framework: Mass Transit (you could also use a messaging bridge to integrate with non-.NET systems). You will learn a little about how Mass Transit differs from NServiceBus, and you'll learn enough of the basics to get started with it. But first, try to picture this scenario: The company that you built the e-commerce system for earlier in this chapter has acquired a successful start-up that deals with promotions. The start-up's platform is already set up to process messages and send out free gifts to random customers when they place an order. To integrate it into your e-commerce NServiceBus system, the acquired platform needs to be able to subscribe to the `OrderCreated` events placed by the Sales bounded context, as shown in Figure 12-28.

Unfortunately, it's not so simple for the new Promotions bounded context to subscribe to the `OrderCreated` event because it isn't using NServiceBus. This is where the messaging bridge comes into play.

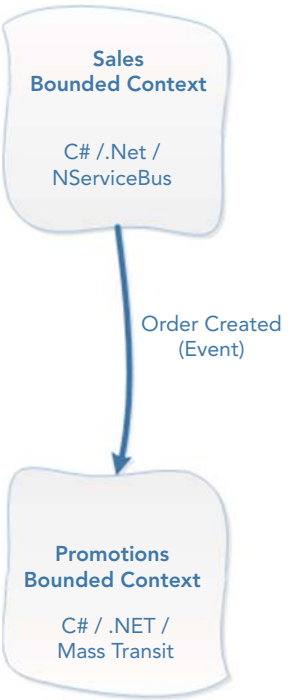


FIGURE 12-28: Promotions bounded context integrating into e-commerce messaging system.

Messaging Bridge

When you have two separate messaging systems that need to share messages, a messaging bridge (`http://www.eaipatterns.com/MessagingBridge.html`) is one way of addressing the issue. It's usually not possible to form a single, fully connected message bus using two independent frameworks, but by using a messaging bridge, you can connect individual endpoints from each messaging system to effectively create a link, as illustrated in Figure 12-29.

Implementing a messaging bridge usually just involves creating an application that can receive messages from one messaging system and hand them over to the other. In your e-commerce system, it's a case of creating a new subscriber to the `OrderCreated_V2` event that converts the event and then pushes it into the new bounded context that uses Mass Transit.

Before you can build that, you need to build the Promotions bounded context that uses Mass Transit. And before that, you need a brief introduction to Mass Transit.

Mass Transit

Variety is often a good thing, which is why .NET developers are lucky to have Mass Transit (`http://masstransit-project.com/`)—an alternative messaging framework to NServiceBus that can be used stand-alone or alongside other messaging frameworks. The latter case is demonstrated in this example. Although it is implemented differently and has its own API, Mass Transit still shares many commonalities with NServiceBus.

Installing and Configuring Mass Transit

Inside the existing solution you created for the NServiceBus parts of the application, you need to add a new C# class library called `Promotions.LuckyWinner.LuckyWinnerSelected`. Then you can add Mass Transit to it by running the following commands in the package manager console (run each as a single-line command):

```
Install-Package MassTransit
-ProjectName Promotions.LuckyWinner.LuckyWinnerSelected
- Version 2.9.5
```

and

```
Install-Package MassTransit.msmq
-ProjectName Promotions.LuckyWinner.LuckyWinnerSelected
- Version 2.9.5
```

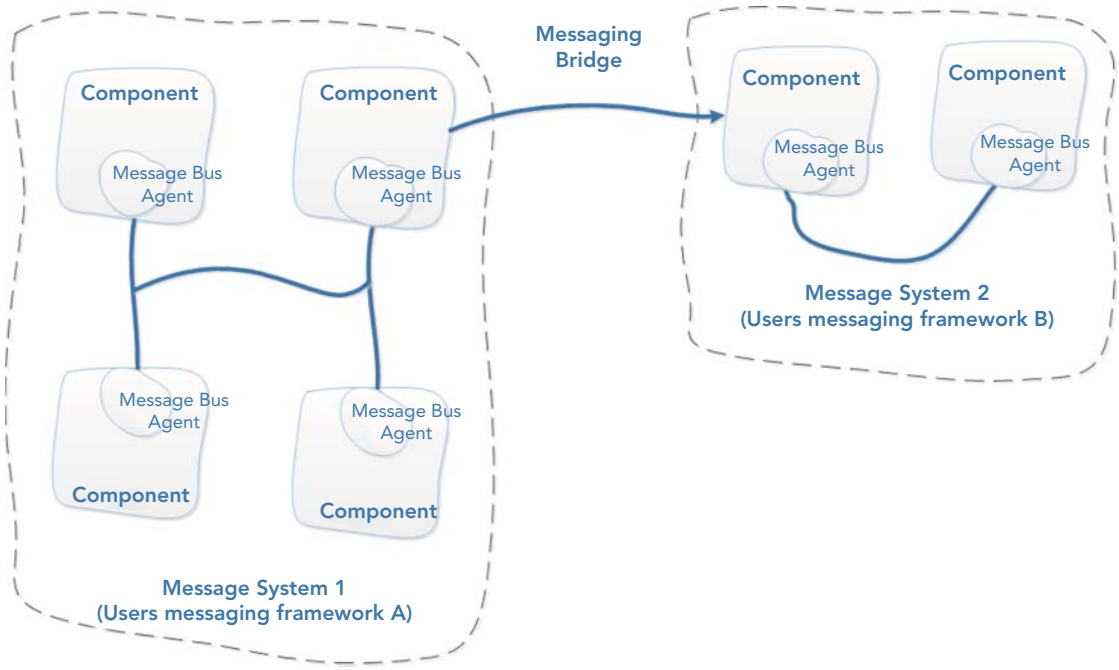


FIGURE 12.29 A messaging bridge is like a link.

At this point, you have all the Mass Transit dependencies referenced in your project, so the next step is to configure an instance of the Mass Transit bus. You'll be happy to know that configuring Mass Transit can be done purely in code. It's quite minimal, too, as Listing 12-23 demonstrates. You can add the contents of Listing 12-23 to your new project.

LISTING 12-23: Bare-Bones Mass Transit Code-Only Configuration

```
using MassTransit;
using System;
using System.Linq;

namespace Promotions.LuckyWinner.LuckyWinnerSelected
{
    class Program
    {
        static void Main(string[] args)
        {
            Bus.Initialize(config =>
            {
                config.UseMsmq();
            });
        }
    }
}
```

At the top, you can see that the Mass Transit classes are imported with the first `using` statement. Inside the main method is where the bus is configured using `Bus.Initialize()`, which gives you a configuration object so you can set up the bus according to your needs. In this example, the bus is being configured to use MSMQ, which you should be familiar with from earlier in this chapter.

Although Listing 12-25 shows the bare-bones configuration needed to get the bus running with MSMQ, it doesn't tell Mass Transit about the kinds of messages that need to be handled or how to handle them.

Declaring Messages for Use by Mass Transit

In keeping with the minimalist theme, Mass Transit messages can be any C# class; they do not have to be named in any way or inherit from any base classes. Because this example is to create a message bridge with NServiceBus, you could just reference and use the same `OrderCreated` class that NServiceBus will publish. However, to make this example stand-alone, it's best if you create a copy of the class in this project. In this way, the two sides of the bridge do not share a dependency. If you were creating a cross-platform messaging bridge, code dependencies would not be possible anyway.

Below the `Program` class you created from Listing 12-23, add an `OrderCreated` class to represent the order-created domain event. This class is now available for Mass Transit to use; you just need to tell Mass Transit how and when to use it. Listing 12-24 shows what your `OrderCreated` event should look like.

LISTING 12-24: OrderCreated Event for Use with Mass Transit

```
namespace Promotions.LuckyWinner.LuckyWinnerSelected
{
    class Program
    {
        // same as previous listing
        ...
    }

    public class OrderCreated
    {
        public string OrderId { get; set; }

        public string UserId { get; set; }

        public List<string> ProductIds { get; set; }

        public string ShippingTypeId { get; set; }

        public DateTime TimeStamp { get; set; }

        public double Amount { get; set; }
    }
}
```

Creating a Message Handler

Again, when configuring message handlers, Mass Transit maintains a lightweight, code-only approach. To create a message handler, you just need to define a method somewhere that handles the message you are interested in. You can then tell Mass Transit to call that method when it receives a message of the required type. This will make sense after an example.

Create a class called `OrderCreatedHandler` with a single `Handle()` method that takes an instance of the `OrderCreated` event you just created inside this project. To keep things simple, you can put this class just below the `Program` class in the same file, as you just did with the `OrderCreated` class. Listing 12-25 contains the implementation of the `OrderCreatedHandler`.

LISTING 12-25: OrderCreateHandler for the Mass Transit Project

```
public class OrderCreatedHandler
{
    public void Handle(OrderCreated message)
    {
        Console.WriteLine(
            "Mass Transit handling order placed event: Order:" +
            " {0} for User: {1}",
            message.OrderId, message.UserId
        );
    }
}
```

Now you're ready to tell Mass Transit how to put your event and handler together.

Subscribing to Events

There's not much work to adding event subscriptions with Mass Transit; just update the code-only configuration to inform Mass Transit which queue to watch for messages and what do with them. Listing 12-26, which is an updated version of the bus initialization you saw in Listing 12-23, sets up a subscription for the `OrderPlaced` event.

LISTING 12-26: Adding Event Subscription Configuration for Mass Transit

```
static void Main(string[] args)
{
    Bus.Initialize(config =>
    {
        config.UseMsmq();
        // look on this queue for order placed event messages
        config.ReceiveFrom("msmq://localhost/promotions.ordercreated");
        // subscribe to order placed events
        config.Subscribe(sub =>
        {
            // handle order placed events like this
            sub.Handler<OrderCreated>(msg =>
                new OrderCreatedHandler().Handle(msg)
```

(*continued*)

LISTING 12-26: *(continued)*

```
                );
            });

        });
        // keep the console running
        while(true)
        {
            Thread.Sleep(1000);
        }
    }
```

The first new line instructs Mass Transit to look on the `promotions.ordercreated` queue on the machine the application is running on (localhost). Mass Transit creates this queue for you, and it can be inspected in Visual Studio's Server Explorer in the same way you inspected the `error` queue previously. When Mass Transit finds a message on that queue, it looks for a subscription to the message's type. You can see how such a subscription is being set up to handle `OrderCreated` events on the other new lines of configuration you just added in Listing 12-26. All that's happening is that inside the `config.Subscribe()` invocation Mass Transit is told to create a new instance of the `OrderPlacedHandler` and pass the message it has taken off the queue to its `Handle` method.

Linking the Systems with a Messaging Bridge

Now is the interesting part: implementing the messaging bridge. To do that, in this example you create a new NServiceBus handler that subscribes to the NServiceBus event and pushes it onto the queue Mass Transit has been configured to receive from. Imagine if you were not using Mass Transit, .NET, or Windows. It doesn't really matter, because you're putting the message in a queue that another messaging framework, running on any run-time or operating system, can take messages off.

> **NOTE** *A messaging bridge is just a pattern for connecting two disparate messaging systems. Because it's just a pattern, it has no prescribed implementation. You can implement a messaging bridge in a variety of ways. For example, you might use flat file or database integration, as outlined in the previous chapter.*

To start building your messaging bridge, you need to create a new C# class library inside the same solution called `Promotions.LuckyWinner.LuckyWinnerSelected.Bridge` with an `OrderCreatedHandler` that contains the content in Listing 12-27. You need to import NServiceBus using the package manager, configure message conventions, reference `Sales.Messages`, and set up the subscription in `App.config`. (Refer to earlier parts of the chapter if you need a reminder about how to carry out these steps, or check this chapter's code samples).

LISTING 12-27: OrderCreatedHandler in the Messaging Bridge

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using NServiceBus;
using System.Messaging;
using Sales.Messages.Events;
using System.Xml.Linq;
using System.Xml;

namespace Promotions.LuckyWinner.LuckyWinnerSelected.Bridge
{
    public class OrderCreatedHandler : IHandleMessages<OrderCreated_V2>
    {

        public void Handle(OrderCreated_V2 message)
        {
            Console.WriteLine(
                "Bridge received order: {0}. " +
                "About to push it onto Mass Transit's queue",
                message.OrderId
            );
            var massMsg = ConvertToMassTransitXmlMessageFormat(message);
            var msmqMsg = new Message
            {
                Body = XDocument.Parse(massMsg).Root,
                Extension = Encoding.UTF8.GetBytes(
                     "{\"Content-Type\":\"application/vnd.masstransit+xml\"}"
                )
            };
            var queue = new MessageQueue(
                ".\\Private$\\promotions.ordercreated", QueueAccessMode.Send
            );
            queue.Send(msmqMsg);
        }

        // use a more robust strategy in production
        // this approach is used to highlight the XML format
        // mass transit needs
        private string ConvertToMassTransitXmlMessageFormat(
                                    OrderCreated_V2 message)
        {
            return "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" +
             "<envelope>" +
                "<headers />" +
                <destinationAddress>"
"msmq://localhost/"promotions.ordercreated?tx=false&amp;recoverable=true" +
                "</destinationAddress>" +
                 "<message>" +
                     "<orderId>" + message.OrderId + "</orderId>" +
                     "<userId>" + message.UserId + "</userId>" +
                     GenerateProductIdsXml(message.ProductIds) +
                     "<shippingTypeId>" + message.ShippingTypeId +
                     "</shippingTypeId>" +
                     "<amount>" + message.Amount + "</amount>" +
                     "<timestamp>" +
```

(*continued*)

LISTING 12-27: *(continued)*

```
                        XmlConvert.ToString(
                         message.TimeStamp,
                         XmlDateTimeSerializationMode.Utc
                        ) +
                      "</timestamp>" +
                  "</message>" +
                  "<messageType>" +
"urn:message:Promotions.LuckyWinner.LuckyWinnerSelected:OrderCreated" +
                  "</messageType>" +
               "</envelope>";
        }

        private string GenerateProductIdsXml(IEnumerable<string> productIds)
        {
            return String.Join(
                "", productIds.Select(p => "<productIds>" + p + "</productIds>")
            );
        }
    }
}
```

As mentioned, this handler subscribes to the NServiceBus event and puts it in the queue Mass Transit has been configured to watch. There are a few key details to understand. First, notice how there is no reference to Mass Transit? All the bridge does is use the MSMQ classes provided by the `System.Messaging.dll` (you need to add a reference to that) to put messages into a queue. The second big aspect to notice is the format of the messages. Mass Transit first looks for a header indicating what type of content is contained in the messages. You can see this being set to `application/vnd.masstransit+xml`, which tells Mass Transit the body contains XML. Mass Transit then treats the body of the message as XML, expecting it to be in the format that is returned from `ConvertToMassTransitXmlMessageFormat()`. Once all that is done, you just create a queue and send a message to it, as shown in the final two lines of `Handle()`.

From this example, you can see a few of the concerns related to a messaging bridge. To push a message into a messaging framework, you need to understand what format it expects. You also need to go down a few levels of abstraction and deal with the queues yourself. Understanding the internals of your messaging framework might be extra work, but it isn't always a bad thing, especially when you often need to monitor the queues and check that everything is working as expected.

If you find that building messaging bridges is becoming inefficient on a project but you still want to build scalable, fault-tolerant systems using some reactive principles, REST may be better. REST and other HTTP-based solutions are covered in the next chapter.

Publishing Events

Publishing events with Mass Transit is almost identical to publishing events with NServiceBus, as shown in the following snippet:

```
Bus.Instance.Publish(new LuckyWinnerSelected(UserId = "user123"));
```

As an exercise, see if you can build a new component with Mass Transit that handles events published by the `Promotions.LuckyWinner.LuckyWinnerSelected` component.

Testing It Out

Your messaging bridge should now be working, so your new Promotions bounded context should be able to handle `OrderCreated` events. All you need is proof of this. To test the application, carry out the following few steps:

1. Set the application as a start-up project and a console application following the advice in this blog post: `http://possiblythemostboringblogever.blogspot.co.uk/2012/08/creating-console-app-in-visual-studio.html`.
2. Run the system (press F5).
3. Place an order as before (browse to `/orders`).
4. Check that the NServiceBus consoles are printing the messages as they previously did.
5. Check that the console for the Mass Transit component received the `OrderCreated` event. If it all works, the console should resemble Figure 12-30. (Yours will be different based on the user ID you entered into the form.)

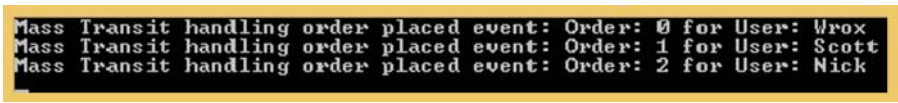


FIGURE 12-30: Mass Transit receiving messages via the bridge.

Where to Learn More about Mass Transit

Mass Transit contains lots of useful functionality that was not shown in this chapter. For example, it can be used with other queuing technologies such as Rabbit MQ. Mass Transit also has a strong community. So if you're keen to learn more about it, you can view all of the source code on GitHub (`https://github.com/phatboyg/MassTransit`), you can read the official documentation (`http://masstransit-project.com/`), or you can read and post questions on the Mass Transit mailing list (`https://groups.google.com/forum/#!forum/masstransit-discuss`).

THE SALIENT POINTS

- Messaging systems are used to build scalable, fault-tolerant systems based on reactive principles and loose coupling.
- When combining Domain-Driven Design (DDD) with messaging systems, use domain events as messages in the system to make domain concepts explicit in the code.
- Containers diagrams can give you early feedback about the design of your messaging system.
- Component diagrams can be used to model the flow of messages using the ubiquitous language (UL).

- Commands are messages that specify something should happen; they are handled in one place.
- Events are used to specify something has happened; events can be handled by multiple subscribers.
- A message bus is a distributed component that has an agent running inside each application that needs to send or receive messages.
- A messaging gateway is a technical pattern used to hide the complexity of an external system and add reliability.
- Messaging systems often synergize with the local data storage principle for scalability and fault tolerance. But that gives rise to eventual consistency.
- Eventual consistency is often a necessity for efficient scalability.
- Rolling forward into new states is a common pattern for dealing with eventual consistency that can lead to openings for new business-rule opportunities.
- UI composition of APIs exposed by components is the recommended practice for presenting information on web applications.
- Backward-compatible messaging versioning is a key pattern for ensuring that teams do not slow each other down when message formats need to change.
- You can use messaging bridges to connect disparate messaging technologies, but it's not always the simplest or most robust solution; HTTP-based alternatives like REST may be a better approach.

13

Integrating via HTTP with RPC and REST

WHAT'S IN THIS CHAPTER?

- An introduction to the idea of using HTTP to integrate bounded contexts
- An introduction to the REST protocol
- Guidance for choosing between RPC and REST when integrating with HTTP
- DDD-focused examples of implementing RPC with SOAP and plain XML
- Examples of implementing RPC using WCF and ASP.NET Web API
- A discussion of how to use REST with DDD to achieve the fault tolerance and scalability of messaging systems while still being domain focused
- An example of building a scalable, fault-tolerant, event-driven, RESTful distributed system using ASP.NET Web API
- Guidance for enabling loosely coupled, independent teams when integrating bounded contexts with HTTP

WROX.COM CODE DOWNLOADS FOR THIS THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 13 download and individually named according to the names throughout the chapter.

Hypertext Transport Protocol (HTTP) is a ubiquitous protocol that the billions of devices connected to the Internet understand. It can also be a discerning choice for integrating bounded contexts. Being so widespread, HTTP has clearly proven that it enables applications running on different hardware and software stacks to communicate relatively easily. This means that if you have bounded contexts using different technologies, HTTP can be very appealing. You saw in the previous chapter that although it is possible to integrate different messaging frameworks, there can be a lot of risky and time-consuming work involved. But because HTTP is a well-known standard, you can follow existing conventions when integrating with it.

Chapters 11, "Introduction to Bounded Context Integration," and 12, "Integrating via Messaging," showed you that integrating bounded contexts is not just about making software applications talk to each other. It's also about providing scalability and fault tolerance. You may be wondering why there are so many messaging frameworks and middleware solutions if HTTP satisfies these needs. That's difficult to answer with any certainty, but it does highlight the fact that HTTP is often overlooked when building event-driven distributed systems. However, REST is definitely starting to gain popularity as a choice for building distributed systems, and it is definitely an option you should at least consider. This chapter shows you why.

Although HTTP hasn't traditionally been used to build reactive, event-driven systems, it has been massively popular for integrating applications using remote procedure call (RPC). In contrast to REST, there are thousands of examples of applications that use RPC to integrate. This means there's a lot of real-world evidence showing its strengths and weaknesses. You learned about some of them in Chapter 11, and in this chapter, you will see concrete examples by building and comparing RPC and RESTful Domain-Driven Design (DDD) systems.

Whichever HTTP-based solution you choose, there are patterns and principles that synergize with DDD techniques to make domain concepts explicit. For example, Chapters 11 and 12 demonstrated how moving to an event-driven architecture based on domain events can have business and technical benefits. You'll see in this chapter that using domain events as the messages sent between bounded contexts via REST can again be very expressive.

Events don't always make sense as HTPP application programming interface (API) formats, though, especially when exposing your domain as APIs to external services. Consider an API that exposes a catalog of products: websites don't want a full history of events, they just want to see the latest snapshot showing the most up-to-date information. So this chapter also contains examples for exposing domain concepts that aren't events as HTTP APIs.

> **NOTE** *A lot of topics are covered in this chapter, so there is not enough space to go into full details about all of them. There are entire books written just about REST, for example. However, after reading this chapter, you will have seen enough theory, examples, and technologies to feel confident about building the types of systems presented in this chapter. As always, though, you are more than welcome to post any questions on the Wrox discussion forum at* `http://p2p.wrox.com/`.

A final topic presented in this chapter is enabling loosely coupled teams to efficiently iterate their bounded contexts. Chapter 11 showed how concepts like Service Oriented Architecture (SOA)

support this need in general, and Chapter 12 showed how to apply those concepts to messaging architecture. This chapter provides similar guidance for achieving loosely coupled teams when using HTTP as your integration protocol.

WHY PREFER HTTP?

When the whole world is using HTTP on all five of the devices they own, it must have its positives. Here are a few of the reasons why integrating your bounded contexts with HTTP might make a significant difference to the success of the projects you are involved in.

No Platform Coupling

Each application or component that integrates with HTTP may be built using any technology thanks to HTTP being a platform-agnostic protocol. Not only is this beneficial for creating loosely coupled applications, but it can help to create loosely coupled teams that have few dependencies on each other.

Using HTTP, each bounded context must honor its public contracts, which are the HTTP request and response formats. Providing the contracts are adhered to, teams are then free to mercilessly refactor, rewrite their applications in new technologies, or continue to add business value at their own pace. All that matters is that they honor their public contract so that integration with other bounded contexts remains intact.

Everyone Understands HTTP

HTTP is everywhere. Almost all programming languages and run times have a wealth of libraries and support for using HTTP. So when integrating with HTTP, there is a massive amount of support available to you. In this chapter, you learn that .NET has a number of frameworks, including Windows Communication Foundation (WCF) and ASP.NET Web API, for building HTTP-based integrations.

Another advantage to everyone understanding HTTP comes when you need to expand your team. It's easy to find a developer who understands HTTP, but it can often be more challenging to find a developer who understands messaging systems or particular messaging frameworks.

Lots of Mature Tooling and Libraries

On top of the modern frameworks and libraries for building HTTP-based integrations, some tooling is advanced. One example of this is the way Visual Studio generates classes for you when you point it to a particular type of web service. The classes provide methods that mimic the API of the HTTP web services, so you can write what appears to be standard object-oriented code yet is actually communicating across the network. This is demonstrated later in the WCF examples.

Dogfooding Your APIs

When all your communication is over HTTP, there may be no need to have dedicated channels for internal and external communication. In plain English, this means that you can build APIs that

bounded contexts use to communicate, and third parties can use those same APIs. In contrast, messaging systems are almost always for internal use only, so APIs have to be produced as well.

The practice of using the APIs internally that you share with clients and partners is known as *dogfooding*. Dogfooding is desirable because it helps you get the same experience as your customers. If your API contains pain points that are putting customers off, dogfooding might help you find and remove them. Of course, in some situations, dogfooding has drawbacks and might not be the best approach. One example might be when you need to have stronger performance guarantees internally than externally.

RPC

If you want to build distributed systems that integrate with HTTP, one option is to use RPC. As discussed in Chapter 11, RPC's "hide-the-network" abstraction can be useful when development speed is important or scalability needs are not too high. On the other hand, the inherent tight coupling associated with RPC can make scalability requirements and loosely coupled teams harder to achieve.

In the following section are examples that demonstrate the previously mentioned strengths and weaknesses of RPC. They allow you to start forming your own opinions and get a feel for where you might want to use RPC in the future.

> **NOTE** *Many companies, even those with massive scalability requirements, still use RPC over HTTP as their main integration strategy, so it can certainly be done. As a lot of these teams grow, though, they start to feel the pain points that manifest as more firefighting than feature delivery. Many of them then tend to move to event-driven and asynchronous alternatives.*

Implementing RPC over HTTP

You have a few choices when it comes to implementing RPC over HTTP. The traditional choice has been to use a protocol called SOAP (Simple Object Access Protocol), which adds another layer on top of HTTP. In recent years, however, SOAP has seen a massive decline in popularity. Nowadays, the more modern approach is to simply use plain eXtensible Markup Language (XML) or JavaScript Object Notation (JSON) as the payload in an RPC call. So that you can make informed decisions, you'll see both options in this section, starting with SOAP.

> **NOTE** *To work through the examples in this chapter, you need an installed copy of the freely available VS Express 2013 for Web. You can download it from MSDN (* `http://www.visualstudio.com/downloads/download-visual-studio-vs#d-express-web`*). Full editions of Visual Studio are also fine.*

SOAP

SOAP fully embraces the concept of RPC by including rich information in the payload, such as type and function meta data. This makes it easy to convert the contents of a SOAP message into a method call on the remote receiver. Due to this richness and being massively popular with the previous generation of developers, the tooling support for SOAP is advanced, as you'll see shortly.

To learn about SOAP and play with the advanced tooling that has been built around it, in this section you integrate two bounded contexts that form part of a social media application. For this scenario, envision that you are part of a start-up building a Twitter-like product that's gaining traction in the market and a rapidly increasing user base.

For this example, you are going to use RPC to help the development teams move faster. Currently they have a single, monolithic, Big Ball of Mud (BBoM) application where bounded contexts are merely libraries that have a binary dependency on each other. This is causing problems to ripple across the entire business when changes to one bounded context are breaking others. You're going to remove this problem by isolating each bounded context as a standalone application that can only be communicated with over HTTP. To make this transition as seamless and rapid as possible, you'll use RPC to replace in-process method calls from one bounded context to another with RPCs over the network (completely removing the binary dependency). This will demonstrate how RPC requires few changes to your code and makes the network almost invisible.

Designing for RPC

Designing for RPC involves deciding which method calls will be RPCs across the network. Apart from that, your code will mostly look the same as it did running on a single machine. Figure 13-1 shows the new design for the current scenario. Note how it replaces methods between two bounded contexts with RPCs across the network.

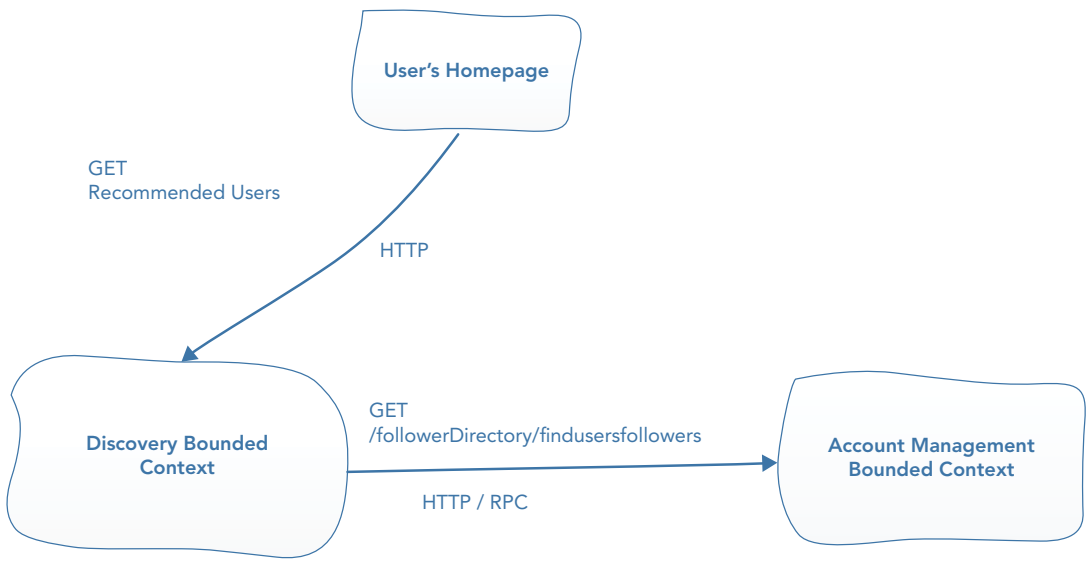


FIGURE 13-1: The "find recommended users" use case.

In the (fictitious) current system, the Discovery bounded context is calling `FindUsersFollowers()` on the `FollowerDirectory` class, which belongs to the Account Management bounded context. This is a binary dependency that requires in-process communication. You can see the code demonstrating this in Listing 13-1.

LISTING 13-1: Existing Solution with Binary Dependencies Between Bounded Contexts

```
namespace Discovery.Recommendations
{
    public class Recommender
    {
        public static List<Accounts> FindRecommendedUsers(string accountId)
        {
            var fd = AccountManageent.FollowerDirectory;

            // in-process call to another bounded context
            var followers = fd.FindUsersFollowers(accountId);

            var recommendations = ApplyCleverRecommendationsAlgorithm(followers);

            return recommendations;
        }
    }
}
```

Listing 13-1 shows the `FollowerDirectory.FindUsersFollowers()` method being called. This is the problematic method that couples two bounded contexts. It is going to be replaced with a similarly named RPC across the network, thereby removing the problematic binary dependency between the Discovery and Account Management bounded contexts.

Figure 13-1 also provides further background for the use case you are going to implement. You can see that the use case is triggered when a user logs in and arrives at her home page. When this happens, the business requirement is for users to see a list of recommended users whom they might want to follow. This is important to the business because it allows users to discover other users and hot topics so they continue to return to the site. An entire team of business people and developers is focused on helping users discover content. The team is known as the Discovery team, and the Discovery bounded context represents its area of the domain.

Implementing RPC over HTTP with WCF and SOAP

Integrating two bounded contexts using SOAP can initially be quite fast and relatively easy when using .NET's Windows Communication Foundation (WCF). You'll see this firsthand as you start to implement the use case shown in Figure 13-1 in the following sections.

> **WARNING** *Although the examples in this chapter show a project per-bounded context, this might not be a good idea for production systems. You should consider isolating domain logic from delivery mechanisms (web applications, desktop applications, commands lines, and so on).*

Creating a WCF Service

To get started, you need to create a blank Visual Studio solution that will be home to all the bounded contexts (in this SOAP example). You can call this solution `PPPDDD.SOAP.SocialMedia`. The first bounded context to be created is Account Management. To add the Account Management bounded context, you need to add a new WCF Service Application to the project called `AccountManagement`, as shown in Figure 13-2.

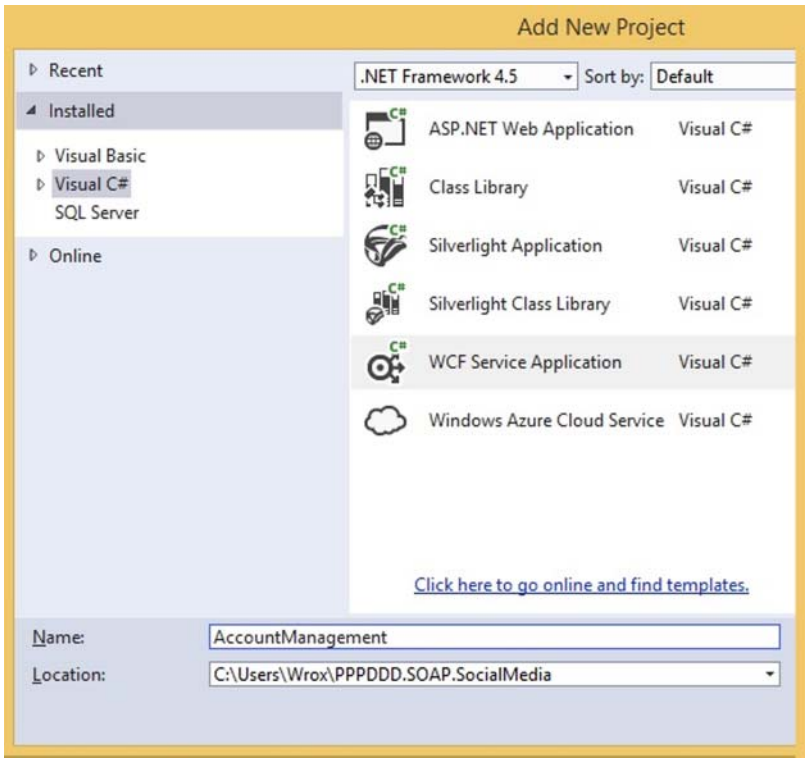


FIGURE 13-2: Adding the Account Management WCF Service.

> **NOTE** *You create the Account Management bounded context first because it exposes a web service called by the Discovery bounded context. As you'll see, the order of creation is important because Visual Studio can generate all the necessary proxy classes that make calling your web service easy if the web service already exists.*

In the old monolithic application, as you saw in Listing 13-1, there was a class called `FollowerDirectory` that had a method called `FindUsersFollowers`. To turn this

into an RPC call, you can simply add a WCF Service to the root of the project called `FollowerDirectory`. Then you can use WCF Service contracts to declare your RPCs, as shown in the next section.

> **NOTE** *To make following along with the examples easier, you should configure the* `AccountManagement` *project to always run on port 3100. You can set this in the project's properties on the Web tab.*

Service Contracts

After you've added the WCF Service, two files are added to the root of the project: `FollowerDirectory.svc.cs` and `IFollowerDirectory.cs`. The latter is what Visual Studio uses to generate a public SOAP contract (using the Web Service Description Language, or WSDL). The former is the implementation; your custom code goes in it and is run when RPC calls are made at run time. You see this in action shortly, so don't worry if it doesn't make perfect sense.

WCF has two annotations: `ServiceContract` and `OperationContract`. `ServiceContract` is added to an interface to signify that the class contains methods that can be called as RPCs across the network. `OperationContract` then signifies which methods on an interface decorated with `ServiceContract` are the RPCs. Therefore, to create the `FollowerDirectory.FindUsersFollowers()` RPC call, you should apply those two attributes, as shown in Listing 13-2.

LISTING 13-2: Denoting RPC Calls with OperationContract in WCF

```
using System;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.ServiceModel;

namespace AccountManagement
{
    [ServiceContract] // tell WCF this class contains RPCs
    public interface IFollowerDirectory
    {
        [OperationContract] // tell WCF this method is an RPC
        List<Follower> FindUsersFollowers(string accountId);
    }

    public class Follower
    {
        public string FollowerId {get; set; }

        public string FollowerName { get; set;}

        public List<string> SocialTags { get; set; }
    }
}
```

Listing 13-2 is all that Visual Studio and WCF requires from you to be able to generate the RPC infrastructure. You'll see later that Visual Studio automatically generates proxies of these classes on clients of the web service. This "networking for free" is what makes WCF and SOAP so appealing to many.

Before your service will work, you need to provide an implementation in the `FollowerDirectory.svc.cs` file. You can see a basic implementation in Listing 13-3 that generates a few dummy Followers in-memory and returns them. You can update your `FollowerDirectory` with this implementation.

LISTING 13-3: Basic Implementation of the FindUsersFollowers RPC

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;

namespace AccountManagement
{
    public class FollowerDirectory : IFollowerDirectory
    {
        public List<Follower> FindUsersFollowers(string accountId)
        {
            return GenerateDummyFollowers().ToList();
        }

        private IEnumerable<Follower> GenerateDummyFollowers()
        {
            for (int i = 0; i < 10; i++)
            {
                yield return new Follower
                {
                    FollowerId = "follower_" + i,
                    FollowerName = "happy follower " + i,
                    SocialTags = new List<string>
                    {
                        "programming", "DDD", "Psychology"
                    }
                };
            }
        }
    }
}
```

Testing WCF Services

You're now in a position to test that you really can call `FindUsersFollowers()`over the network as an RPC. Visual Studio makes this easy with the test client it provides. To run the test client, highlight the `FollowerDirectory.svc` item in the Solution Explorer (not `FollowerDirectory.sv.cs`) and press F5. You will then see the test client as illustrated in Figure 13-3.

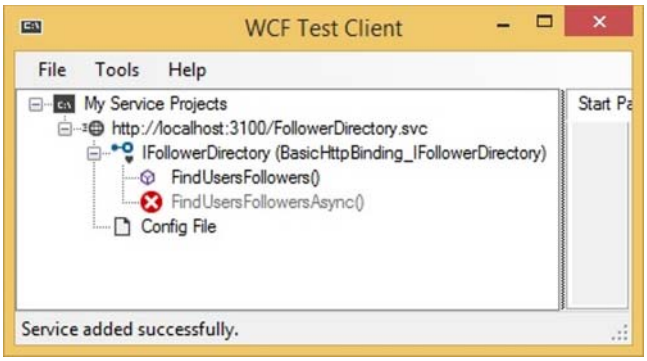


FIGURE 13-3: Visual Studio's WCF test client.

To test your new service, just double-click its name (`FindUsersFollowers`) in the left-hand Explorer pane, and then enter a value in the `Value` column for the row `accountId` in the right pane. After doing that, if you click the Invoke button, your `FindUsersFollowers` RPC will be carried out over the network, and the results will be displayed in the lower half of the right pane, as shown in Figure 13-4.

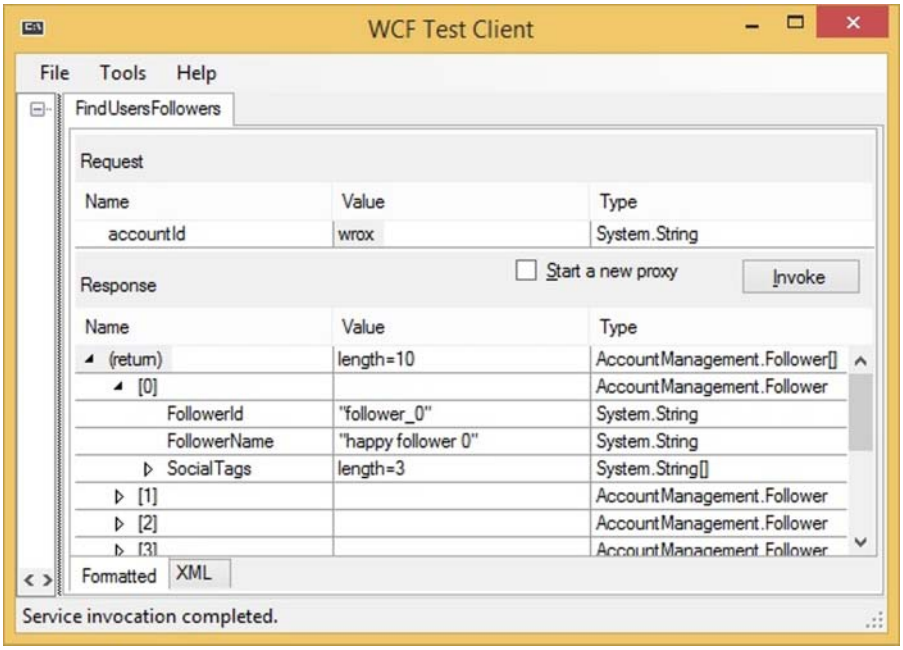


FIGURE 13-4: Invoking an RPC in WCF's test client.

If you want to see how the data was transmitted across the network, you can click on the XML tab at the bottom of the right pane. By doing that, you see the raw SOAP:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header />
  <s:Body>
    <FindUsersFollowersResponse xmlns="http://tempuri.org/">
      <FindUsersFollowersResult
xmlns:a="http://schemas.datacontract.org/2004/07/
AccountManagement" xmlns:i="http://www.w3.org/2001/
XMLSchema-instance">
        <a:Follower>
          <a:FollowerId>follower_0</a:FollowerId>
          <a:FollowerName>happy follower 0</a:FollowerName>
          <a:SocialTags
xmlns:b="http://schemas.microsoft.com/2003/10/
Serialization/Arrays">
            <b:string>programming</b:string>
            <b:string>DDD</b:string>
            <b:string>Psychology</b:string>
          </a:SocialTags>
        </a:Follower>
        <a:Follower>
        ...
```

Creating WCF Service Clients

You're now about to see that the WCF, SOAP, and Visual Studio combination does a lot of the hard work for you. You'll first create the Discovery bounded context project, and you'll then see how you can start making RPC calls to the Account Management bounded context just by pointing the Discovery bounded context to the uniform resource locator (URL) of the Account Management bounded context.

To begin, you need to add a new WCF Service Application to the solution called `Discovery` that represents the Discovery bounded context. Inside the Discovery bounded context, you then need to add a WCF Service called `Recommender`. `Recommender` provides the web services that the website uses to get the list of recommended users. You may find it helpful to quickly refer to the design in Figure 13-1.

`Recommender` has two responsibilities. First, it provides the API for clients to request recommendations. To fulfill that responsibility, it makes the RPC to the Account Management bounded context get an Account's followers. To implement that, you need to have the Account Management project running. (Highlight it in the Solution Explorer and press Ctrl+F5.) You can then test that it is running by directly accessing it in a web browser at `http://localhost:3100/FollowerDirectory.svc`. If the page has the heading "FollowerDirectory Service," things are working.

Next, you need to pass the URL to Visual Studio so it can generate the proxy classes. If you right-click on the `References` node for the Discovery project in the Solution Explorer and select Add Service Reference, the URL can be pasted into the `Address` field. All that's left is to change the Namespace (at the bottom of the Add Service Reference dialog) to `AccountManagement` and click Go. Your screen

should then resemble Figure 13-5, which shows the expanded `FollowerDirectory` node revealing the web service it has identified. Once you're happy, you can click OK.

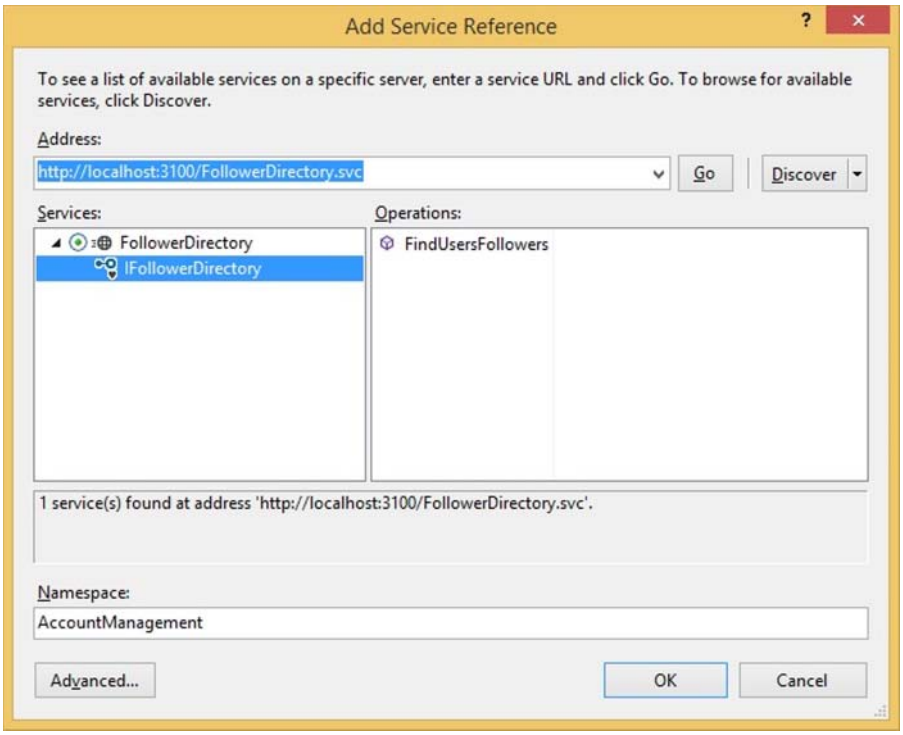


FIGURE 13-5: Adding a Service Reference in Visual Studio.

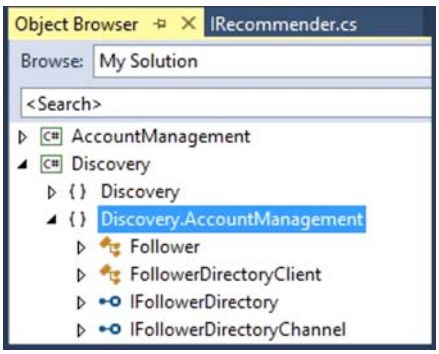


FIGURE 13-6: Generated proxy classes.

To see the generated proxy classes, you can inspect the `AccountManagement` item that was added to the `Service References` folder. Figure 13-6 shows the generated proxy classes. These are the classes that you will shortly instantiate in the Discovery bounded context and call methods on to invoke RPCs across to the Account Management bounded context (via SOAP/HTTP).

The last step is to build the Recommender web service that puts the generated proxy classes to work. Listing 13-4 and Listing 13-5 show a basic implementation of the `IRecommender` and `Recommender` that do just enough to demonstrate the RPC call. You need to add these to your `Discovery` project.

LISTING 13-4: IRecommender Exposing Web Service

```
using System;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.ServiceModel;

namespace Discovery
{
    [ServiceContract]
    public interface IRecommender
    {
        [OperationContract]
        List<string> GetRecommendedUsers(string accountId);
    }
}
```

LISTING 13-5: Recommender Implementing Web Service by Making RPC

```
using System;
using System.Linq;
using System.Collections.Generic;
using Discovery.AccountManagement;

namespace Discovery
{
    public class Recommender : IRecommender
    {
        public List<string> GetRecommendedUsers(string accountId)
        {
            // create an instance of the proxy class
            var accountManagementBC =
                 new AccountManagement.FollowerDirectoryClient();

            // call methods on the proxy - this initiates the SOAP/HTTP RPC call
            var followers = accountManagementBC.FindUsersFollowers(accountId);
            return FindRecommendedUsersBasedOnSocialTags(followers);
        }

        private List<string> FindRecommendedUsersBasedOnSocialTags(
            Follower[] followers)
        {
            /*
             * Real system would look at the users tags and find
             * popular accounts with similar tags by making more
             * RPC calls etc.
             */
            var tags = followers.SelectMany(f => f.SocialTags).Distinct();
            return tags.Select(t => t + "_user_1").ToList();
        }
    }
}
```

In Listing 13-5, an instance of `AccountManagement.FollowerDirectoryClient` is created. It is a proxy class that Visual Studio generated when adding the Service Reference. When its `FindUsersFollowers()` method is called, it fires an RPC across the network and calls into the code you added in the Account Management bounded context. The main takeaway here is that WCF and Visual Studio took care of all the network-related plumbing. Most of the code you added would look very similar even if there was no network involved.

You can test that everything is successfully working by setting both projects to start up (as shown in the previous chapter by right-clicking the solution in the Solution Explorer and choosing Set Startup Projects) and then pressing F5 on the `Recommender.svc` in the Solution Explorer. The WCF test client pops up again, and this time you need to invoke `GetRecommendedUsers()`, as demonstrated in Figure 13-7.

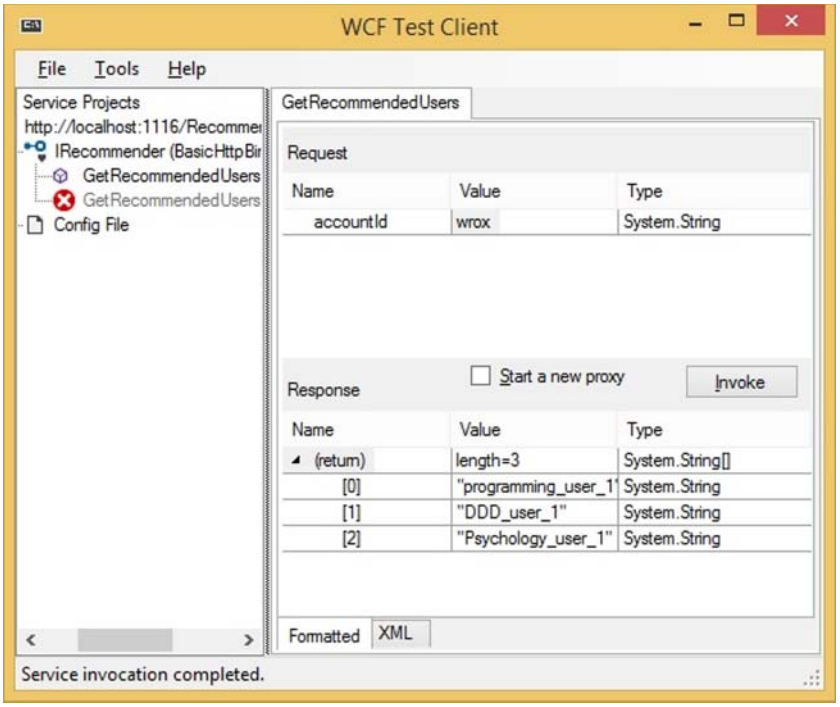


FIGURE 13-7: The RPC must have occurred.

The result of the RPC call in Figure 13-7 is the hard-coded data that is returned by the web service you created. This indicates that the RPC call between the two bounded contexts successfully occurred. Conclusively, this example's aim of integrating the Account Management and Discovery bounded contexts without a binary dependency has now been achieved.

SOAP's Decline

Although there are many existing public SOAP APIs, new APIs just aren't being built with SOAP anymore. One of SOAP's big pain points is the complexity and verbosity of its message format,

which you saw a glimpse of earlier. People are critical of needless complexity, and they often cite SOAP as a perfect example of unnecessary complexity. Accordingly, you should be careful about exposing public SOAP APIs, but you shouldn't feel too concerned about using SOAP internally if it suits your needs.

The modern preference for RPC over HTTP is to use lightweight, plain XML or JSON payloads. The next section shows examples of this using ASP.NET Web API.

Plain XML or JSON: The Modern Approach to RPC

To see how you can integrate over HTTP without the complexity and verbosity of the SOAP format, in this section, you re-create the social media SOAP integration instead using the relatively lightweight JSON. The following is the JSON version of the SOAP payload shown earlier. You may want to go back and compare to fully appreciate how the JSON is far more compact.

```
{
    "followers": [
        {
           "accountId": "34djdlfjk2j2",
           "socialTags": [
               "ddd", "soa", "tdd", "kanban"
           ]
        },
        ...
    ]
}
```

> **NOTE** *Because this example is a reimplementation of the previous SOAP/WCF example, the design in Figure 13-1 also applies here. You may want to quickly refer back to the design for a refresher.*

Implementing RPC over HTTP with JSON Using ASP.NET Web API

The ASP.NET Web API is Microsoft's latest framework for creating web services. Later you will see how it can be a good choice for building RESTful APIs. But for now you'll see how it can make life easy when building JSON RPC APIs. As a starting point, you need a new blank Visual Studio solution called `PPPDDD.JSON.SocialMedia`. This solution needs to be populated with an ASP.NET Web Application project called `AccountManagement`. As you go through the creation process for the project, you need to select the Empty template and check the Web API check box. Once it's created, you need to configure the project to always start on port 3200.

Controllers contain the code that will be run when web requests are made to your Web API. You can see controllers as an opportunity to express domain concepts from the ubiquitous language (UL), although you should be careful about putting domain logic in them. Some developers conceptualize controllers as Application Services.

> **NOTE** *Application Services and the service layer are covered in detail in Chapter 25, "Commands: Application Service Patterns for Processing Business Use Cases."*

To start with this example, you need to add a class called `FollowerDirectoryController` to the `Controllers` folder in the root of the project; Web API requires that controllers be placed in this folder. The code for `FollowerDirectoryController` is shown in Listing 13-6.

LISTING 13-6: FollowerDirectoryController

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace AccountManagement.Controllers
{
    public class FollowerDirectoryController : ApiController
    {
        public IHttpActionResult GetUsersFollowers(string accountId)
        {
            var followers = GenerateDummyFollowers().ToList();
            return Json(followers);
        }

        private IEnumerable<Follower> GenerateDummyFollowers()
        {
            for (int i = 0; i < 10; i++)
            {
                yield return new Follower
                {
                    FollowerId = "follower_" + i,
                    FollowerName = "happy follower " + i,
                    SocialTags = new List<string>
                    {
                        "programming", "DDD", "Psychology"
                    }
                };
            }
        }

    }

    public class Follower
    {
```

```
            public string FollowerId { get; set; }

            public string FollowerName { get; set; }

            public List<string> SocialTags { get; set; }
        }

    }
```

For demonstrative purposes, the `FollowerDirectoryController` in Listing 13-6 returns a list of hard-coded `Followers` (as JSON). In a real application, this class would likely perform database lookups or API calls to get the required follower information.

After starting the application (by pressing F5), you can test the new web service by hitting it in the browser. Using Web API's default conventions, it is accessible at `http://localhost:3200/api/followerdirectory/getusersfollowers?accountId=123`, where the value for the `accountId` parameter is variable. (It can be anything in this example.) Figure 13-8 shows an example of hitting the API from a browser and viewing the JSON response.

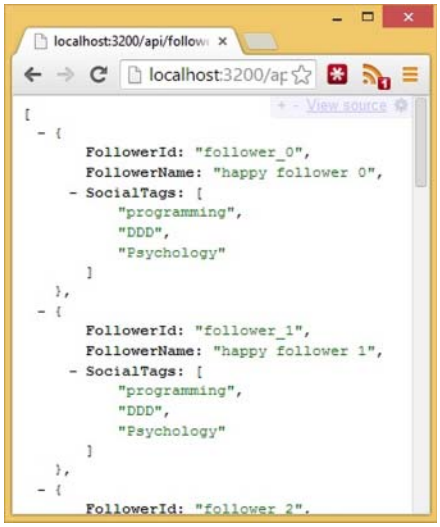


FIGURE 13-8: Viewing the output of a Web API controller in a browser.

> **NOTE** *It helps to install a plug-in that formats JSON when rendered in a browser. For Chrome, there is such a plug-in called JSONView (*`https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefhakgolnmc?hl=en`*). Firefox, Internet Explorer, and other browsers have similar plug-ins.*

You can see with this approach that you had to do a tiny bit of extra work setting up the controller compared to SOAP and WCF, but the data sent across the wire is so much cleaner and lighter that it's easy to understand what is happening. This is also a bonus when it comes to debugging problems.

Lacking the richness of meta data provided by SOAP, it is not possible to automatically generate proxy classes for a plain JSON API. It still doesn't have to be a lot of extra work, though, as you'll now see.

To create a client of your JSON API, you need to add a new ASP.NET Web Application, called `Discovery`, to represent the Discovery bounded context. Inside the `Discovery` project, you need to add a class called `RecommenderController` in the `Controllers` folder. The code for this class is shown in Listing 13-7. For it to compile, you need to install `HttpClient` and `ServiceStack.Text` by running the following commands in the Nuget Package Manager Console:

```
Install-Package Microsoft.AspNet.WebApi.Client -Project Discovery
Install-Package ServiceStack.Text -Project Discovery
```

LISTING 13-7: RecommenderController

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using ServiceStack.Text;

namespace Discovery.Controllers
{
    public class RecommenderController : ApiController
    {
        public List<string>  GetRecommendedUsers(string accountId)
        {
            var accountManagementUrl =
                "http://localhost:3200/api/" +
                "followerdirectory/getusersfollowers?" +
                "accountId=" + accountId;

            var response = new WebClient().DownloadString(accountManagementUrl);
            var followers = JsonSerializer
                               .DeserializeFromString<List<Follower>>(response);
             // automatically converted to JSON by Web API
            return FindRecommendedUsersBasedOnSocialTags(followers);
        }

        private List<string> FindRecommendedUsersBasedOnSocialTags(
            List<Follower> followers)
        {
            /*
             * Real system would look at the users tags and find
             * popular accounts with similar tags by making more
             * RPC calls.
             */
            var tags = followers.SelectMany(f => f.SocialTags).Distinct();
```

```
            return tags.Select(t => t + "_user_1").ToList();
        }

    }

    /* class not shared between bounded contexts to meet
     * requirement of no source code dependencies
     */
    public class Follower
    {
        public string FollowerId { get; set; }

        public string FollowerName { get; set; }

        public List<string> SocialTags { get; set; }
    }
}
```

As Listing 13-7 shows, the logic for this implementation is similar to the WCF approach in the previous example. However, with this solution, you have to do the manual work of making the HTTP request and parsing the response yourself. As you can see, though, there are feature-rich libraries, provided by Microsoft and the community, that do a lot of the laborious work for you.

> **WARNING** *Listing 13-7 takes the simplest possible approach to integration. In a real solution, you might want to use asynchronous web clients, and you might want to check for error conditions. You might also want to set headers, such as Accept specifying your preferred format (in cases where the remote service supports multiple formats like XML, JSON, HTML, and CSV).*

To test that everything works as intended, you need to set both projects as start-up projects (as shown in previous examples). You then need to navigate to the URL of the `GetRecommendedUsers` API you just created. The URL is `http://localhost:{port}/api/recommender/getrecommendedusers?accountId=123` depending on which port the Discovery bounded context is using on your machine. A browser automatically pops up informing you of the port number when you press F5 inside the solution (or you can manually specify a port, as shown earlier in this chapter).

Choosing a Flavor of RPC

Integrating bounded contexts with RPC over HTTP was just demonstrated using two common approaches. With WCF and SOAP you can add a few attributes to your domain model and suddenly it becomes a distributed system free from binary dependencies between bounded contexts. In turn, this allows teams to be more independent and not have to worry about breaking other bounded contexts. One problem with SOAP, though, is that the format is complex and verbose; in many cases, RPC is used for simple integrations, so this doesn't seem logical. This is why plain XML or JSON is the more popular choice today.

Both options, however, have the flaws inherent to RPC that were mentioned in Chapter 11. First, they can be harder to scale efficiently. Looking back at the diagram in Figure 13-1, consider a forthcoming request from the business to improve the speed at which recommended followers are displayed onscreen. Both the Discovery bounded context and the Account Management bounded context may need to be scaled to provide an overall performance improvement. If the chain or RPCs spanned three bounded contexts, then three bounded contexts may need to be scaled due to the temporal coupling.

In terms of fault tolerance, there are also worrying signs. If the Account Management bounded context goes down, the Discovery bounded context also goes down because it cannot RPC across and get the followers. Again, this is due to the temporal coupling.

It may seem like you need to choose between integrating with HTTP for loose platform coupling and having a scalable, fault-tolerant system that uses a messaging framework like NServiceBus. But that's completely untrue. The next section of this chapter shows that you can have the scalability and fault tolerance of a messaging system and the loose platform coupling of HTTP by combining the principles of reactive programming with REST.

REST

In this section, you rebuild the social media integration for a third time. This time, however, you completely redesign for scalability, fault tolerance, and development efficiency using event-driven REST. This third design iteration still relies only on HTTP instead of a heavyweight messaging framework. But this version still follows the reactive, SOA, and loose coupling principles presented in Chapter 11.

REST is a misunderstood and misused term, though. Before you build any RESTful applications, it is crucial that you understand what REST really is.

Demystifying REST

REST was introduced to the world by Roy Fielding. He created REST as an architectural style based on the principles that make the Internet so successful. REST has a number of fundamental concepts, including resources and hypermedia, which provide the platform for evolvable clients and servers.

> **NOTE** *Roy Fielding introduced REST to the masses with his now infamous "Architectural Styles and the Design of Network-Based Software Architectures" thesis. You can view it online at* `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

Resources

HTTP requests to a RESTful system are for resources. Responses contain the requested resource (if successful). Resources can be things like documents, such as web pages, or media, such as MP3 files. Resources work well with DDD because concepts in your domain can be expressed as resources—further spreading the UL. As a basic example, in a financial domain, there could be transactions that transfer funds from one account to another. The UL would contain an entry for each type of transaction, such as B2B Transaction or Personal Transaction. These transactions could be exposed as resources accessible from the uniform resource identifiers (URIs aka URLs) `http://pppddddemo.com/`

`B2bTransactions` or `http://pppddddemo.com/PersonalTransactions`. This is completely different from RPC, where requests and responses simulate method calls using imperative naming.

> **WARNING** *There is a difference between URIs and URLs that's important to be aware of. You shouldn't think of them as meaning exactly the same. Essentially, URLs are just one type of URI; there are other types, although they are less common. You can learn more about the differences on Wikipedia (*`http://en.wikipedia.org/wiki/Uniform_resource_identifier`*).*

Resources have a one-to-many relationship with representations. In other words, when requesting a resource, you can specify a different protocol or a different content type, such as JSON, XML, or HTML. Each response will be the same resource but will be presented differently according to the syntactic rules of the requested format. You will see shortly that clients of RESTful APIs choose a format by specifying the required Multi Media Encoding (MIME) type in HTTP's "Accept" header.

Here are a couple other key details relating to resources:

- There is a many-to-one mapping between URIs and resources. Multiple URIs, therefore, can point to the same resource.
- Resources can be hierarchical. For instance, to expose a user's address, you could use a URI such as `/accounts/user123/address`. Each segment in the path is a child resource of the segment to its left, just like a path in a file system.

Hypermedia

Humans browsing the web go from web page to web page by clicking links. This is hypermedia in action. By returning hyperlinks in resources, computers, too, can move from resource to resource simply by following links. This is demonstrated later in the chapter.

Hypermedia presents another opportunity for DDD practitioners to express their domain more explicitly. Imagine a car insurance policy. Each step of the application process could be expressed as links in hypermedia to the next possible steps—expressed using the UL. Not only does this express domain concepts, but it can be used to model workflows or domain processes.

Using hypermedia in machine-to-machine communication means that clients of a RESTful API are not coupled to its URIs. This leads to decoupled clients and servers, free to evolve independently. This is one of the fundamental reasons people have for using REST, because SOAP-based solutions tend to be brittle due to tightly-coupled clients and servers.

Statelessness

Application state, such as the items in a user's shopping cart, arguably should not be stored on the server in a RESTful application. This provides a foundation for fault tolerance and scalability because clients do not have to keep hitting the same machine that contains the state. Application state should therefore be kept on the client and sent to the server every time the server requires it. Going back to the shopping cart example, in a stateless REST API, the cart items could be stored in cookies and sent to the server with every request. Any problems a server may have, therefore, should not preclude other servers stepping in to take over from it.

> **NOTE** *In practice, statelessness can involve a number of trade-offs, and you may need to make pragmatic decisions. However, having the intention of being stateless where possible is a worthwhile philosophy.*

REST Fully Embraces HTTP

HTTP has a number of conventions that provide the basis for scalability, fault tolerance, and loose coupling. Because REST is based on these principles that make the web successful, it is essential that you at least have a basic understanding of HTTP's features before you build RESTful applications.

Verbs

HTTP provides a uniform interface for interacting with resources. For example, to fetch a resource, you send a GET request to its URI. To delete the same resource, you send a DELETE request to the URI. You can use PUT requests to create a resource at the desired URI. For adding items to a collection, you can use the POST verb. Examples of how these common verbs can be used to interact with resources are shown in Table 13-1.

TABLE 13-1: Using HTTP Verbs to Create, Read, Update, and Delete Resources

URI	VERB	ACTION
`/accounts/user123`	GET	Read/fetch the resource
`/accounts/user123`	DELETE	Delete the resource
`/accounts/user123`	PUT	Create the resource
`/accounts/user123/addresses`	POST	Update the resource

> **WARNING** *Table 13-1 is not a strict definition of the semantics of HTTP verbs. Most notably, there is often a lot of fluidity and debate about when to use PUT and POST. For more information, "The RESTful Cookbook" contains a relevant discussion (*`http://restcookbook.com/HTTP%20Methods/put-vs-post/`*).*

By having a single set of verbs that are applied uniformly across the entire Internet, it is easy to build generic API clients and infrastructure components, such as caches, that understand the web's conventions. Think about it—every programming language has libraries for working with HTTP. This is the power of common conventions, and you can also harness them when integrating bounded contexts.

Status Codes

Complementary to HTTP's verbs are its status codes. As with verbs, having a common set of status codes means any agent on the web understands the conventions. For example, whenever you make a

request to a URI that doesn't exist, you get an HTTP 404 status code back because it is a common standard that almost all systems adhere to.

HTTP status codes are grouped by their first digit, as shown in Table 13-2. Within each group are more specific status codes.

TABLE 13-2: HTTP Status Code Groups

STATUS CODE GROUP	DEFINITION	EXAMPLE
1xx	Informational	This is rarely used.
2xx	Success	The resource you requested is returned.
3xx	Redirection	The resource you requested has been moved to another address.
4xx	Client error	You supplied an invalid parameter value.
5xx	Server error	There is a bug in the API code preventing the resource from being returned.

Wikipedia has an accessible introduction to HTTP status codes if you would like to learn more (`http://en.wikipedia.org/wiki/List_of_HTTP_status_codes`).

Headers

Aside from the URI and body of an HTTP request/response, you're probably familiar with headers that provide extra information. RESTful systems frequently use HTTP's caching headers that are covered later in this chapter.

Most RESTful applications require some level of security. Because a property of REST is statelessness, it's often recommended that authentication and authorization details are communicated in headers using protocols such as OAuth.

For more information on headers that you can use in HTTP requests and responses, Wikipedia has an accessible, yet detailed entry (`http://en.wikipedia.org/wiki/List_of_HTTP_header_fields`).

> **NOTE** *Authentication and authorization of RESTful systems is beyond the scope of this book. An excellent free resource for learning about them online is the "The RESTful CookBook" (*`http://restcookbook.com/Basics/loggingin/`*).*

What REST Is Not

REST can be a good choice for many projects and a suboptimal choice for others. Whatever choice you make, it's important to name things accordingly for accurate communication. Unfortunately, REST is a much misused term. So before calling your API RESTful, you should check that as a bare minimum it centers on hypermedia and resources.

After numerous high-profile abuses of the term REST, in 2008 Roy Fielding was compelled to write a blog post demanding that people meet some basic requirements for their API to be called RESTful (`http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`). Another countermeasure to the abuse of the term REST is the Richardson Maturity Model (`http://martinfowler.com/articles/richardsonMaturityModel.html`). This is basically a barometer indicating how close your API is to being RESTful.

REST for Bounded Context Integration

The context for the remainder of this chapter is a redesign of the RPC examples earlier in the chapter. With some killer new features and viral marketing campaigns, user sign-ups for the fictitious social media start-up have again increased exponentially. The business wants to cash in on its success by adding premium accounts that are promoted to regular users. Premium accounts are a way for companies to gain followers on the social media website so they can profit from enhanced brand loyalty.

Unfortunately, as happens on many occasions, the RPC-based integration is not scaling well. It was the perfect choice initially for its time-to-market advantages but is now preventing features from being delivered because developers spend too long fire-fighting. So before you can add new features, you need to stabilize the system to support the rapid growth of the business.

This new version of the system is based on an event-driven architecture. Interestingly, though, instead of taking the common message bus approach (as per the previous chapter), this system uses REST and HTTP to preclude technology/vendor lock-in for the lifetime of the system.

Designing for REST

As suggested in the previous chapter, a small amount of design can create a shared vision and a deeper understanding of how the system being built addresses its functional and nonfunctional requirements. Some steps for designing a system that uses REST for integration will differ from those used to design a messaging system. Mostly the steps will be similar, though, including the first step: start with the domain.

DDD

Expressing business policies, using the UL, in a set of sketches is again a useful first step in designing a system. It nearly always makes sense to work out what problem you need to solve and what domain processes you need to model before you decide on a technical solution.

Figure 13-9 is a component diagram illustrating the new event-driven design of the Recommended Accounts use case. As with the messaging solution in the previous chapter, it focuses on domain commands and events. In fact, this design could be for a messaging system, because the diagram focuses on the flow of messages during the business use case and is independent of technology choices.

In Figure 13-9, you can see two key domain events. First, there is Began Following. Throughout the company, every member of staff understands that a Began Following domain event occurs when one account starts following another. This is part of the UL and one of the core concepts of the business. The other domain event shown is Premium Recommendations Identified. This is also part of the UL, representing occurrences where the Discovery bounded context has identified a premium account

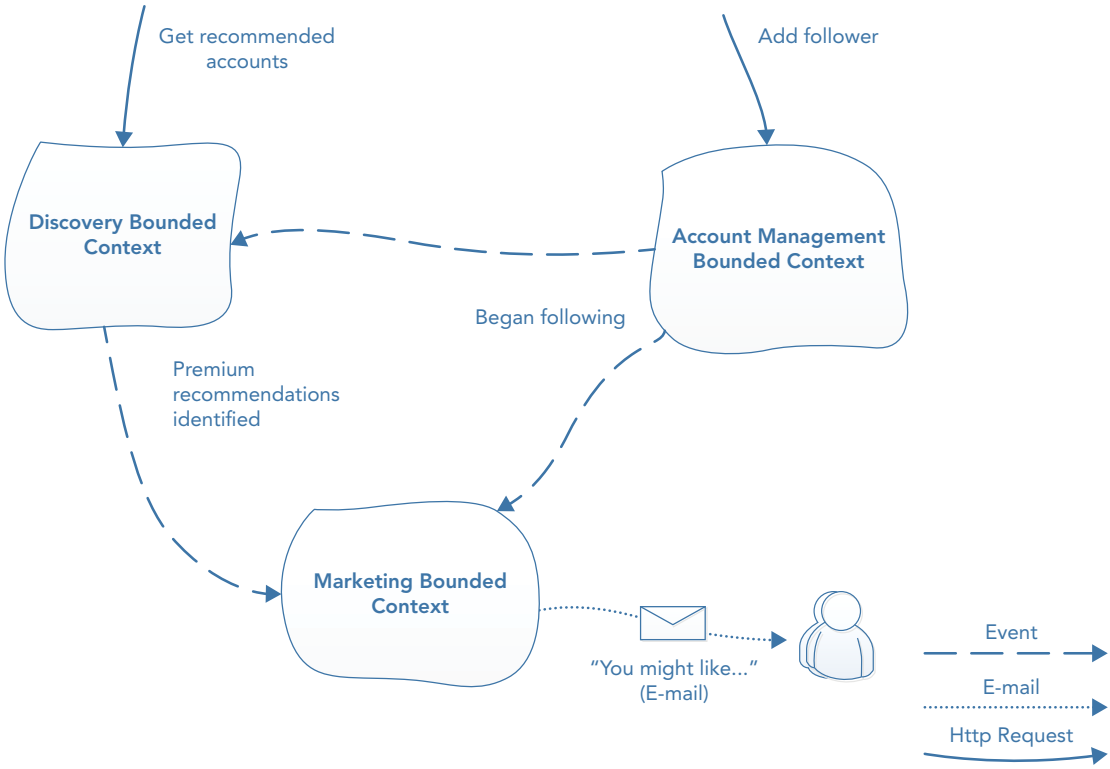


FIGURE 13-9: Component diagram for the Recommended Accounts use case.

that a regular account may like to follow. This is a crucial domain concept, because promoting premium accounts to regular users is central to the new business model.

SOA

Chapter 11 showed you that following the principles of SOA can lead to loosely coupled bounded contexts. In turn, this can provide the platform for high-performing teams. SOA's principles are technology agnostic, so you can apply them when building systems with HTTP.

By isolating bounded contexts each owned by a single team, loose coupling is within reach. Each team is free to develop its features in line with its business priorities, free of cross-team distractions or dependencies. All that changes when using SOA with HTTP compared to messaging is that the contract between teams is no longer classes in code, but the format of HTTP requests and responses. Upcoming examples fully demonstrate this.

Event-Driven and Reactive

You saw in Chapters 11 and 12 that asynchronous messaging, based on reactive principles, was the recommendation for building fault-tolerant scalable systems. This is also the case when integrating with REST and HTTP. As you are probably aware, HTTP doesn't inherently support publish/

subscribe, so there's no way to push out events to subscribers as they occur like you can with a message bus. Instead, with REST, clients can poll for changes. Polling generally has negative connotations in terms of scaling, but utilization of HTTP's caching conventions can negate this problem.

> **NOTE** *Technically, it is possible to build scalable, event-driven systems with REST without resorting to polling. Each event publisher can keep a list of subscribers and a URL for each of them where a message can be published to. This approach tends to be more complex and often stirs up debates about statelessness.*

Figure 13-10 is the containers diagram for the new Reactive, RESTful social media system that will be built in the remainder of this chapter. Note how there are some similarities to a messaging system: each component is small so that it can be scaled independently according to business needs; bounded contexts do not share dependencies such as databases. Additionally, thanks to HTTP, each team can use any technologies it prefers within its bounded contexts (which was more difficult to achieve in the previous chapter). These traits support scalability, fault tolerance, and development velocity.

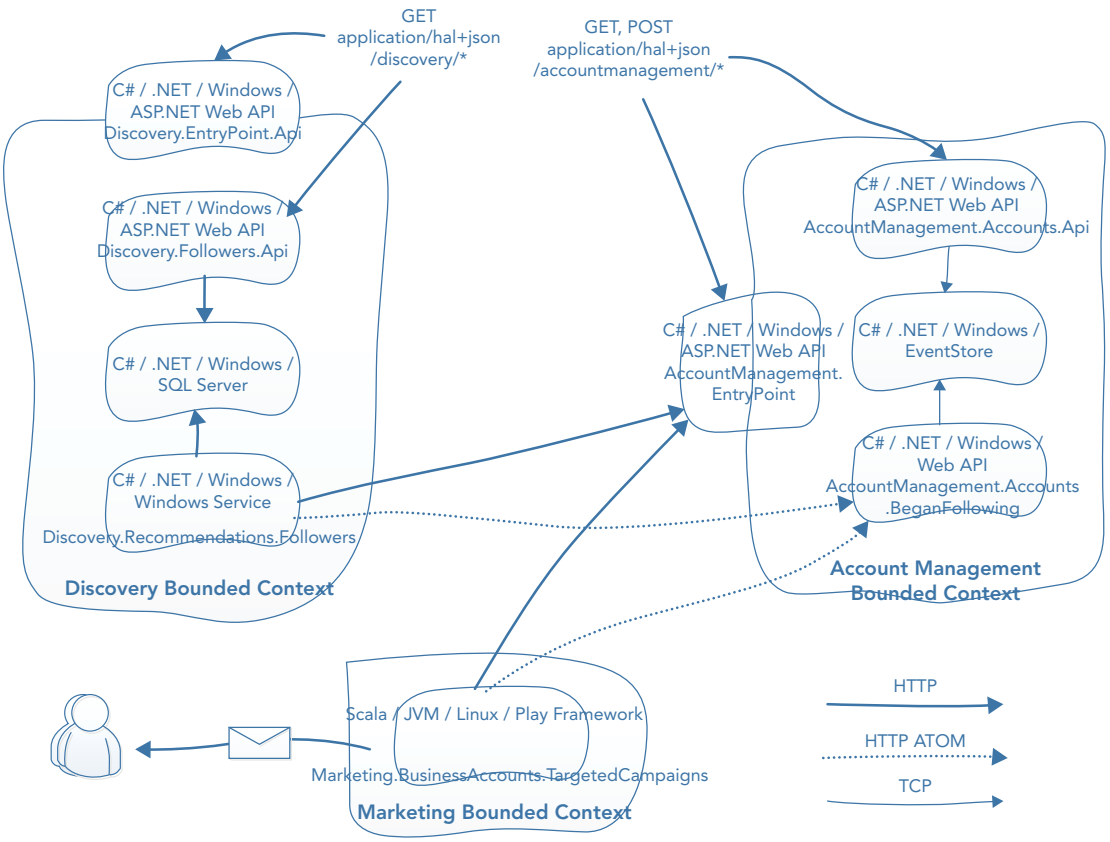


FIGURE 13-10: Containers diagram of Discovery, Account Management, and Marketing bounded contexts.

An important design consideration for scalability is the granularity of projects. For your HTTP APIs, you could put all your endpoints in one project, but then you couldn't easily deploy them independently based on their individual scalability needs. The loose recommendation in this chapter is to start with one project per resource. Examples of this in Figure 13-10 are the stand-alone projects for the entry point resource and the Accounts resource.

When you have nested resources, you may also want to consider moving them into their own project sometimes. The main factors for doing so will usually be complexity of having extra projects traded off against the ability to scale APIs independently. You may even want to move individual request handlers into their own project if specific use cases have demanding scalability requirements.

HTTP & Hypermedia

Hypermedia is fundamental to REST because it is the factor that enables clients and servers to evolve independently of one another. So it can be useful to think a little about the hypermedia contracts up front before you start to build the system. (You can still iterate as you go along.) A good way to design REST workflows is to use sequence diagrams like the one shown in Figure 13-11 that demonstrates the event-driven Add Follower use case.

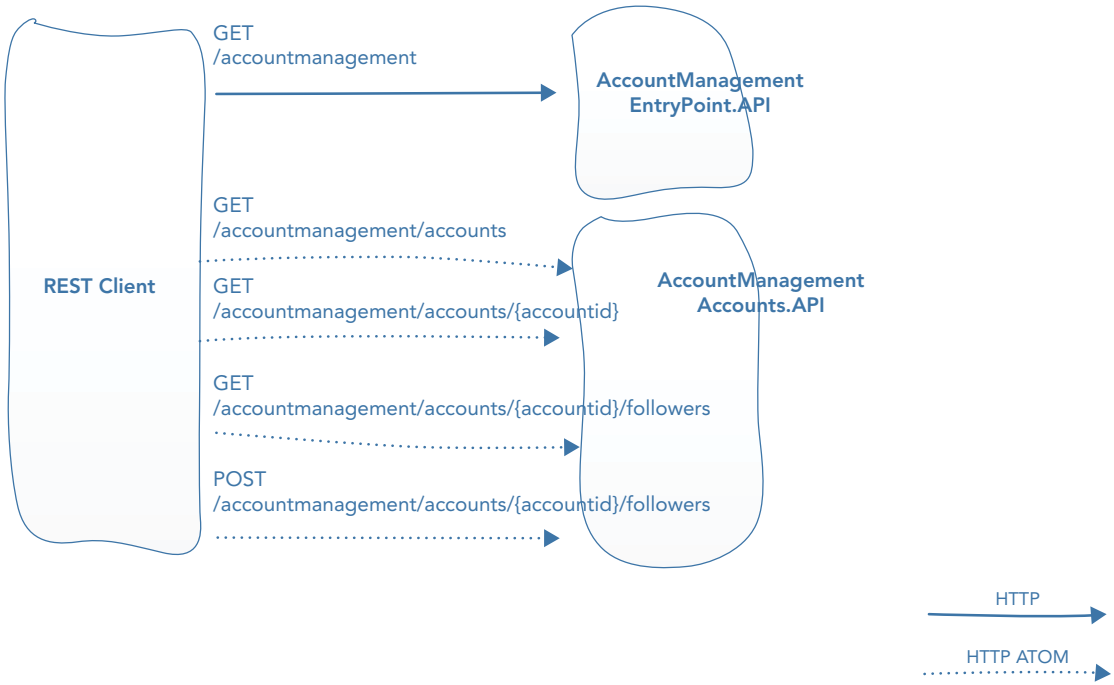


FIGURE 13-11: Flow of HTTP requests for the Add Follower use case.

As you can see, clients are only coupled to the entry point URI. From there, they make requests to other services by following hypermedia links provided in responses. For instance, clients wanting to initiate the Add Follower use case (akin to a domain command) on the Account Management

bounded context are only coupled to the URI of the entry point resource—/accountmanagement. From then on, clients merely follow links in the hypermedia, returned as HTTP responses, until the Followers resource is reached.

In the containers diagram shown in Figure 13-10, there is an indication next to each HTTP endpoint of its content type. Most of them show application/hal+json. HAL stands for Hypertext Application Language; it is essentially just well-known existing content types—XML and JSON—with conventions for representing hypermedia links. You can learn about the HAL standard in depth on Mike Kelly's blog (http://stateless.co/hal_specification.html). The examples in the remainder of this chapter use application/hal+json, so you will still learn the basics just by reading this chapter.

Another content type shown on the containers diagram is application/atom+xml, which denotes Atom. Atom is a common standard for producing RSS feeds and thus is a great fit for representing lists of events. This is precisely how it is used by the Account Management bounded context—to represent the list of Began Following events that have occurred.

Using Atom as a feed of events is the main building block for building event-driven distributed systems with REST in this chapter. It doesn't have to be Atom, but Atom's popularity means you should definitely consider it.

An example of polling an Atom feed for domain events is shown in Figure 13-12. This diagram shows the flow of HTTP messages involved for polling the Began Following Atom feed that you will build later in this chapter.

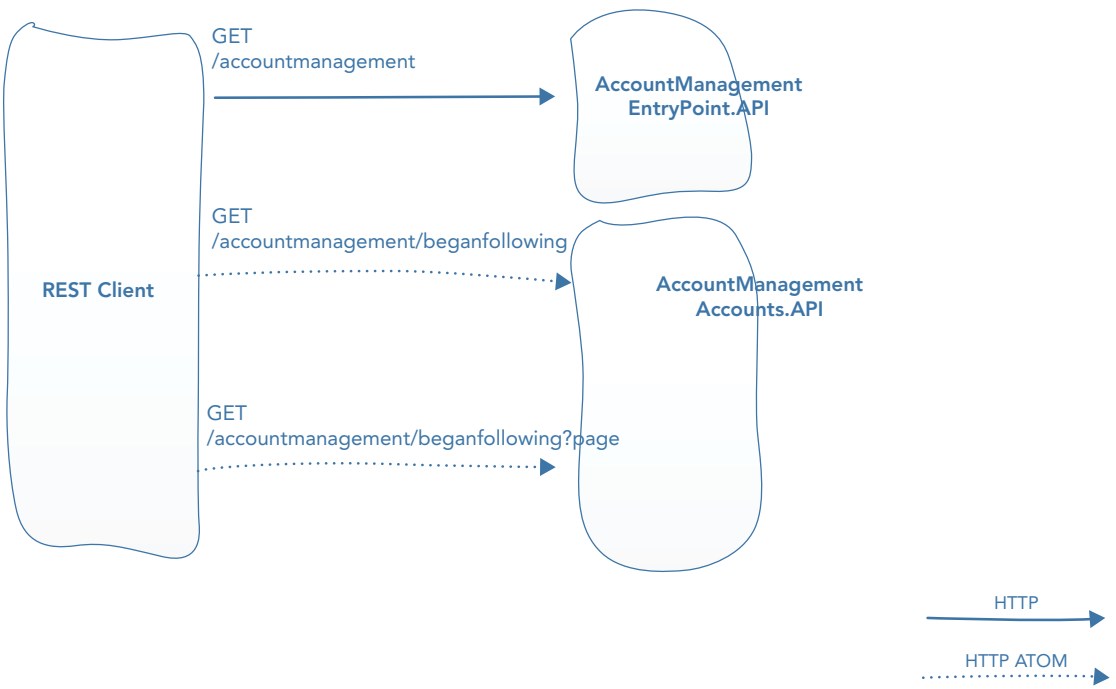


FIGURE 13-12: Flow of HTTP requests for polling and consuming the Began Following event feed.

Building Event-Driven REST Systems with ASP.NET Web API

Now that you've heard all the theory, it's time to start building the RESTful, event-driven version of the social media application. To keep the examples concise and focused on essential patterns, you will just be building the Account Management bounded context and some of the Discovery bounded context. From these examples, you will learn enough to begin using the concepts on your own projects.

> **NOTE** *Even though you will not be implementing all the bounded contexts shown in the diagrams in this section, it was important to show them on the designs so that you would have an appreciation for designing larger systems. Rest assured, the remaining examples cover all the important concepts. Building the full system would just be unnecessary repetition.*

An outside-in approach will be taken to build the RESTful social media system; you will start by adding the hypermedia entry point to the Account Management bounded context. You'll then create the Accounts API. After that you'll create the Atom feed that publishes Began Following events. Finally, you'll create the consumer of the Began Following feed that resides in the Discovery bounded context.

During the upcoming examples, all code will live in the same Visual Studio solution for convenience. But when you're building RESTful systems that integrate over HTTP, this is not necessary. In fact, you may be using different technologies where it is not even possible to share a solution. It's up to you to decide what you think is best; having completely separate code repositories for each part of the system encourages looser coupling, but keeping code close together may help to share the bigger picture.

To begin building the new RESTful social media system, you can start by creating a new blank Visual Studio solution called `PPPDDD.REST.SocialMedia`.

Hypermedia-Driven APIs

Hypermedia is at the heart of REST. Accordingly, you will now see how to build hypermedia APIs in .NET with ASP.NET Web API. Although the implementation details will vary from framework to framework, the concepts are framework agnostic and will apply to any tools you may decide to build hypermedia APIs with.

As you've seen, the key benefit of hypermedia is that it decouples clients and servers, allowing independent evolution. But clients must know something up front about the API. This is the role of the entry point resource that clients *should* couple themselves to.

Entry Point Resource

When clients want to interact with a REST API, they start by requesting the entry point resource. From then on, they mostly just follow links in the hypermedia that is returned. Choosing where to locate your entry point(s) has a number of considerations that you should take into account. For instance, the design in Figure 13-10 chooses to have a single entry point per-bounded context. But

you could have a single entry point for the entire system or go more fine-grained and have an entry point per top-level resource. Ultimately, you have to decide on a per-project basis how much of the system you want to expose via entry point resources.

Designing an entry point involves identifying the initial resources and transitions that should be available to consumers of the API. The Account Management bounded context's entry point will be the list of top-level resources. In this example, that is just the Accounts resource. From the Accounts resource, hypermedia will link to individual accounts, and from individual accounts to details of those accounts, exposed as child resources, such as its followers. This is just repeating what you saw in the sequence diagram earlier in the chapter.

As you're starting to see, links are the building blocks of hypermedia. But for API clients to follow links, they need to be able to decide which link provides the transition they are looking for. This is the role of link relations, which indicate what the link represents (or, more specifically, its relationship to the current resource). For example, for a resource that has many pages, to be hypermedia-compliant, it would need a link with the relation Next, which clients can follow to the next page. You'll see a number of links and relations in the upcoming examples.

> **NOTE** *To learn more about common link relations, see the official Internet Assigned Numbers Authority (IANA) documentation, which contains a comprehensive list (*`http://www.iana.org/assignments/link-relations/link-relations.xhtml`*). It is also permissible to use custom link relations. In doing so, you are trading the benefits of common conventions that everyone understands for less understood conventions that convey your specific domain more clearly.*

To build the API that produces the Account Management entry point, you need to start by adding a new ASP.NET application to the solution called `AccountManagement.EntryPoint`. This follows the convention of naming API projects based on the format `{bounded context}.{Resource}`. When adding the project to your solution, be sure to select the empty template, and be sure to check the Web API check box.

> **NOTE** *Throughout this section, the empty template and Web API check box should always be selected when adding a new ASP.NET Web Application project in Visual Studio.*

One decision still needs to be made before you can add the endpoint that produce the entry point resource. You need to choose a media type that supports hypermedia.

HAL

Traditionally, XHTML has been used as the hypermedia format for REST APIs. Unfortunately, it's undesirable due to its verbosity (especially if you're trying to escape from SOAP). Fortunately,

though, a relatively new standard is available and gathering traction. This new standard is HAL, which was briefly introduced earlier in the chapter, and it comes in two main flavors: XML and JSON. Essentially, both of those well-known formats have been extended with specific conventions for representing links. This provides the hypermedia benefits of XHTML without the verbosity as the following example demonstrates.

```
{
  "_links": {
    "self": {
      "href": "http://localhost:4100/accountmanagement"
    },
    "accounts": {
      "href": "http://localhost:4101/accounts"
    },
 }
}
```

The entry point resource shown in the preceding snippet demonstrates the conventions for representing links in HAL (JSON). All links must be defined within an element at the root of the resource called `_links`. Each link begins with its relation (`self` and `accounts` in the example). Each link also contains an href, which is the URI of the resource it points to. Only the `self` link, which points to the current resource, is mandatory; all other links are optional.

To build the entry point resource API, you first need to configure the project to start on port 4100. (You can set this in the project's properties on the Web tab.) You then need to configure a URI for the entry point by adding a route inside Web API's `WebAPIConfig`, as shown in Listing 13-8. Your `WebApiConfig` file will be located in the `App_Start` folder that sits in the root of the project.

LISTING 13-8: Adding the Explicit Entry Point Route to WebApiConfig

```
using System;
using System.Web.Http;

namespace AccountManagement.EntryPoint.Api
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "Entry Point",
                routeTemplate: "accountmanagement",
                defaults: new { controller = "EntryPoint", action = "Get" }
            );
        }
    }
}
```

If you're not familiar with Web API's routing syntax, in Listing 13-8, the definition of the `EntryPoint` route ensures that whenever a request comes in for the path `/accountmanagement`, the `Get()` method on a class called `EntryPointController` is invoked. Before you can implement that controller, you need to install a Nuget package that adds support for HAL in Web API. As you will see, this is an incredibly usable library that takes all the effort out of creating HAL APIs. To install `WebApi.Hal` into the `AccountManagement.EntryPoint.Api` project, you need to run the following command in the Nuget Package Manager Console:

```
Install-Package WebApi.Hal -Project AccountManagement.EntryPoint.Api
```

> **NOTE** *Web API supports conventional routing to avoid having to explicitly specify each route. This example chooses to specify routes explicitly to make it easier to follow along. It's completely up to you choose the approach you prefer.*

Once `WebApi.Hal` is installed, you need to tell Web API to make HAL (JSON) the default media type by updating your `Global.asax.cs`, as per Listing 13-9. This also sets HAL (XML) as the second preference. The two formatters, `JsonHalMediaTypeFormatter` and `XmlHalMediaTypeFormatter`, belong to the `WebAPI.Hal` package you just installed.

LISTING 13-9: Setting HAL as the Default Content Type Using WebApi.Hal's Formatters

```
using System;
using System.Web.Http;
using WebApi.Hal;

namespace AccountManagement.EntryPoint.Api
{
    public class WebApiApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(WebApiConfig.Register);

            // default media type (HAL JSON)
            GlobalConfiguration.Configuration.Formatters.Insert(
                0, new JsonHalMediaTypeFormatter()
            );

            // alternative media type (HAL XML)
            // accept header must equal application/hal+xml
            GlobalConfiguration.Configuration.Formatters.Insert(
                1, new XmlHalMediaTypeFormatter()
            );
        }
    }
}
```

All dependencies are installed and configuration is now complete, paving the way for you to complete the Entry Point API by implementing the `EntryPointController`. You can achieve this by first

adding a class called `EntryPointController` to the `Controllers` folder that sits at the root of the project. Once you've added the class, you can then replace the contents of the file with the code in Listing 13-10.

LISTING 13-10: EntryPointController for the AccountManagement.EntryPoint.Api Project

```
using System;
using System.Collections.Generic;
using System.Web.Http;
using WebApi.Hal;

namespace AccountManagement.EntryPoint.Api.Controllers
{
    public class EntryPointController : ApiController
    {
        private const string EntryPointBaseUrl = "http://localhost:4100/";
        private const string AccountsBaseUrl = "http://localhost:4101/";

        [HttpGet]
        public EntryPointRepresentation Get()
        {
            return new EntryPointRepresentation
            {
                Href = EntryPointBaseUrl + "accountmanagement",
                Rel = "self",
                Links = new List<Link>
                {
                    new Link
                    {
                        Href = AccountsBaseUrl + "accountmanagement/accounts",
                        Rel = "accounts",
                    },
                }
            };
        }
    }

    public class EntryPointRepresentation : Representation
    {
        protected override void CreateHypermedia(){}
    }
}
```

The code in Listing 13-10, when executed, will return the entry point resource as HAL-JSON (by default). This is because `EntryPointRepresentation` inherits from the `Representation` base class—a class that the `WebApi.Hal` library provides. When a class inheriting from `Representation` is returned from a controller method, the `JsonHalMediaTypeFormatter` will convert it to HAL-JSON.

Inside `Get()`, the code declaratively maps onto the response format. You can see two links are being generated in this code: `Href` and `Links` (a collection with just one link). These are the two links that will appear in the response. You can see all this in action by testing out what you have so far.

Testing HAL APIs with the HAL Browser

A huge benefit of building hypermedia APIs on top of common standards is that it is easy to create generic clients that can explore APIs built using those standards. One such tool for HAL is the Hal browser, a small web application that allows you to consume and interact with HAL APIs. You'll now see how to use the HAL browser to test the Entry Point API you have just built.

To install the HAL browser, you have to download it (`https://github.com/mikekelly/hal-browser/archive/master.zip`) and then unzip it. Finally, you need to copy the contents of the unzipped folder into the root of your `AccountManagement.EntryPoint.Api` project's folder in Windows Explorer. If you open Windows Explorer at the root of the `AccountManagement.EntryPoint.Api` project, there should be a file called `browser.html`. If you don't see it, the file is in the wrong location.

If you are happy that the files are copied into the correct location, you can press F5 to run your project. In the browser that Visual Studio starts up for you, you can access the HAL browser by navigating to `http://localhost:4100/browser.html`. (This assumes that you configured this project to always use port 4100, as explained previously.) After navigating to `browser.html`, if everything is working correctly, you will see the HAL browser, as per Figure 13-13.

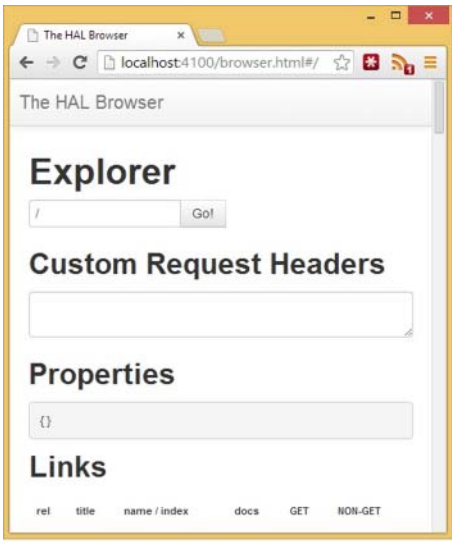


FIGURE 13-13: Accessing the HAL browser.

> **NOTE** *If you are experiencing any problems with the HAL browser or any of the other tasks in this chapter, you are more than welcome to post your questions on the Wrox discussion forum at* `http://p2p.wrox.com/`. *You can also inspect this chapter's code download for a reference point.*

When you do access the HAL browser, you'll notice it doesn't show your entry point resource. This is because the HAL browser looks for API entry points at the default path (/). To remedy this, simply enter **`/accountmanagement`** into the HAL browser's navigation bar (below the Explorer label) and click GO. You should then see the raw entry point resource (on the right side) and the interactive tools (on the left side), as shown in Figure 13-14.

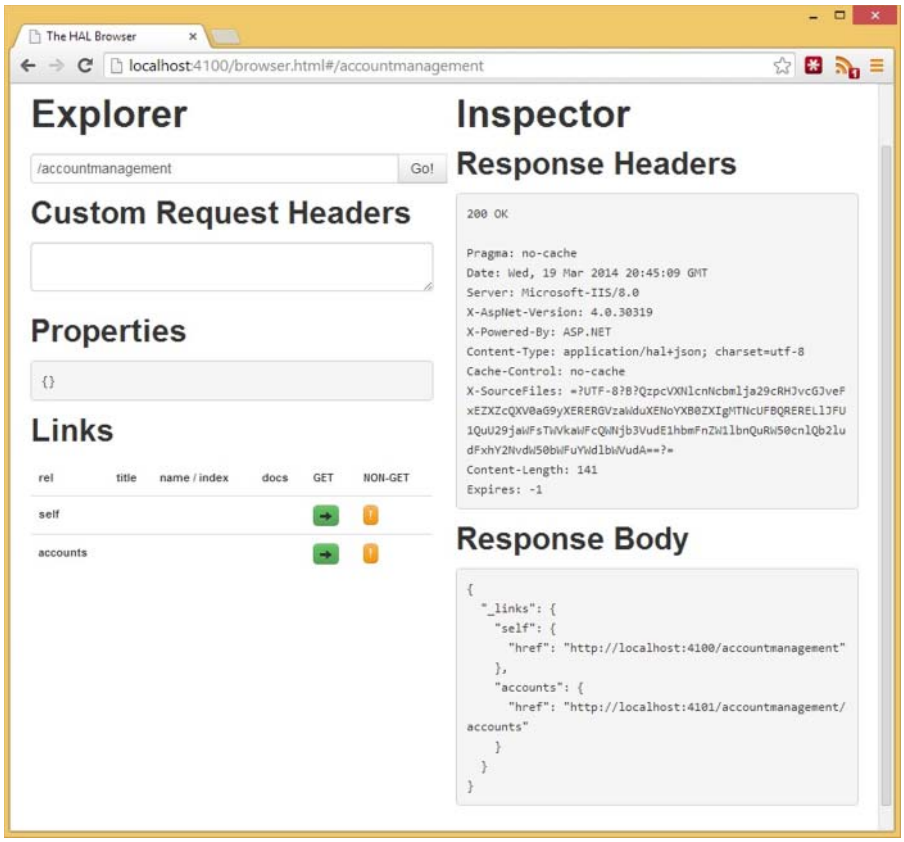


FIGURE 13-14: Viewing the entry point resource in the HAL browser.

At the moment, the HAL browser provides little benefit because the links in the entry point resource point to nonexistent resources. So this is your next task: to implement the Accounts API. As the entry point resource in Figure 13-14 shows, the Accounts resource needs to be accessible at `http://localhost:4101/accountmanagement/accounts`.

URI Templates

In building the Accounts API, you will see how to handle a common concern people have for hypermedia: inefficient navigation. Consider an API that exposes lots of data, such as thousands or millions of accounts. Clients may have to navigate hundreds of links to find the resource they want by

successfully following links to the next page. Obviously, this can be massively inefficient for the client and the server—especially for public APIs with many concurrent clients. This problem is solved by using URI templates.

The following sample shows the Accounts resource, as HAL (JSON), that you are going to create an API for shortly. Look for the URI template; it's the link whose `templated` attribute is set to `true`:

```
{
  "_links": {
    "self": {
      "href": "http://localhost:4101/accountmanagement/accounts"
    },
    "alternative": {
      "href": "http://localhost:4101/accountmanagement/accounts?page=1"
    },
    "account": [
      {
        "href": "http://localhost:4101/accountmanagement/accounts/{accountId}",
        "templated": true
      },
      {
        "href": "http://localhost:4101/accountmanagement/accounts/123"
      },
      ...
}
```

To represent URI templates, not only do you need to set the `templated` attribute to `true`, but you must also add placeholder sections in the URI. In the Accounts resource response just shown, you can see the placeholder is `{accountId}`. Clients of the API can then replace the placeholder with the ID of the account they are looking for. In doing so, you cut down hundreds of potential requests into just a single one. Creating URI templates with `WebApi.Hal` requires little effort, as you will now see while creating the Accounts API.

You can create the Accounts API by adding a new project called `AccountManagement.Accounts.Api`. You need to set this as a start-up project and ensure that it runs on port 4101. Once it's created, you can then add a reference to `WebApi.Hal` by running the following command in the Nuget Package Manager console:

```
Install-Package WebApi.Hal -Project AccountManagement.Accounts.Api
```

The final configuration step is to set HAL as the default content type, as shown previously in Listing 13-9.

Two URIs will be exposed by the Accounts API. Initially, clients will click the Accounts resource at /accountmanagement/accounts, which represents the entire list of accounts. From there, clients will then navigate to individual accounts using the URI template contained within the Accounts resource—`accountmanagement/accounts/{accountId}`. To declare these routes in your project, the `WebApiConfig` in your `AccountManagement.Accounts.Api` project should resemble Listing 13-11.

LISTING 13-11: Route Declarations for the Accounts API in WebApiConfig

```
using System;
using System.Web.Http;

namespace AccountManagement.Accounts.Api
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "Accounts Collection",
                routeTemplate: "accountmanagement/accounts",
                defaults: new { controller = "Accounts", action = "Index" }
            );

            config.Routes.MapHttpRoute(
                name: "Individual Account",
                routeTemplate: "accountmanagement/accounts/{accountId}",
                defaults: new { controller = "Accounts", action = "Account"}
            );
        }
    }
}
```

As the route declarations in Listing 13-11 show, a class called `AccountsController` needs to be added to the `Controllers` folder (still inside `AccountManagement.Accounts.Api`). It needs two methods—`Index()` and `Accounts()`—as dictated by the route definitions. Starting with just `Index()`, the `AccountsController` should initially contain the code shown in Listing 13-12.

LISTING 13-12: AccountsController

```
using System;
using System.Collections.Generic;
using System.Web.Http;
using WebApi.Hal;

namespace AccountManagement.Accounts.Api.Controllers
{
    public class AccountsController : ApiController
    {
        private const string EntryPointBaseUrl = "http://localhost:4100/";
        private const string AccountsBaseUrl  =
                              "http://localhost:4101/ accountmanagement/";
```

(*continued*)

LISTING 13-12: *(continued)*

```
        [HttpGet]
        public AccountsRepresentation Index()
        {
            return new AccountsRepresentation
            {
                Href = AccountsBaseUrl + "accounts",
                Rel = "self",
                Links = new List<Link>
                {
                    new Link
                    {
                        Href = AccountsBaseUrl + "accounts?page=1",
                        Rel = "alternative",
                    },
                    new Link
                    {
                        // automatically identified as a template
                        Href = AccountsBaseUrl + "accounts/{accountId}",
                        Rel = "account",
                    },
                    new Link
                    {
                        Href = AccountsBaseUrl + "accounts/123",
                        Rel = "account",
                    },
                    new Link
                    {
                        Href = AccountsBaseUrl + "accounts/456",
                        Rel = "account",
                    },
                    new Link
                    {
                        Href = AccountsBaseUrl + "acccounts?page=2",
                        Rel = "next"
                    },
                    new Link
                    {
                        Href = EntryPointBaseUrl + "accountmanagement",
                        Rel = "parent"
                    }
                },
            };
        }
    }

    public class AccountsRepresentation : Representation
    {
        protected override void CreateHypermedia()
        {
        }
    }
}
```

Most of the code in Listing 13-12 will be familiar from Listing 13-10, but it is worthwhile noting the additional links. The `alternative` link relation represents links that have a different URI but point to the same resource (`self`). One benefit of this is that clients can cache more efficiently by treating both URIs as the same resource. Below the `alternative` link is the templated `account` link, which `WebApi.Hal` automatically marks as templated because the `href` contains a placeholder.

Before you can test the new API in the HAL browser, you need to enable Cross-Origin Resource Sharing (CORS). This is because the Accounts resource is provided by a different vhost (port 4101 as opposed to 4100). The instructions for enabling CORS are detailed on the ASP.NET website (`http://www.asp.net/web-api/overview/security/enabling-cross-origin-requests-in-web-api`). Basically, you need to do the following:

1. Add the CORS package to the project by running the following command inside the Nuget Package Manager Console:

```
Install-Package Microsoft.AspNet.WebApi.Cors -Project AccountManagement.Accounts.Api
```

2. Configure CORS in `WebApiConfig`, as per Listing 13-13.

LISTING 13-13: ENABLING CORS IN ASP.NET WEB API

```
using System;
using System.Web.Http;
using System.Web.Http.Cors;

namespace AccountManagement.Accounts.Api
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            var cors = new EnableCorsAttribute("*", "*", "*");
            config.EnableCors(cors);

     // unchanged code
```

If you run the HAL browser as before, starting at the entry point and following the link to the Accounts resource, you will see the URI template link, as per Figure 13-15.

> **WARNING** *When you run the application after adding CORS, you may see an error indicating that a version of* `System.Web.Http` *could not be loaded. To resolve this, you need to set up an assembly binding. When you build the solution and then double-click the warning that begins* `Found conflicts between...` *(in the Error List toolbar), Visual Studio adds the necessary bindings for you.*

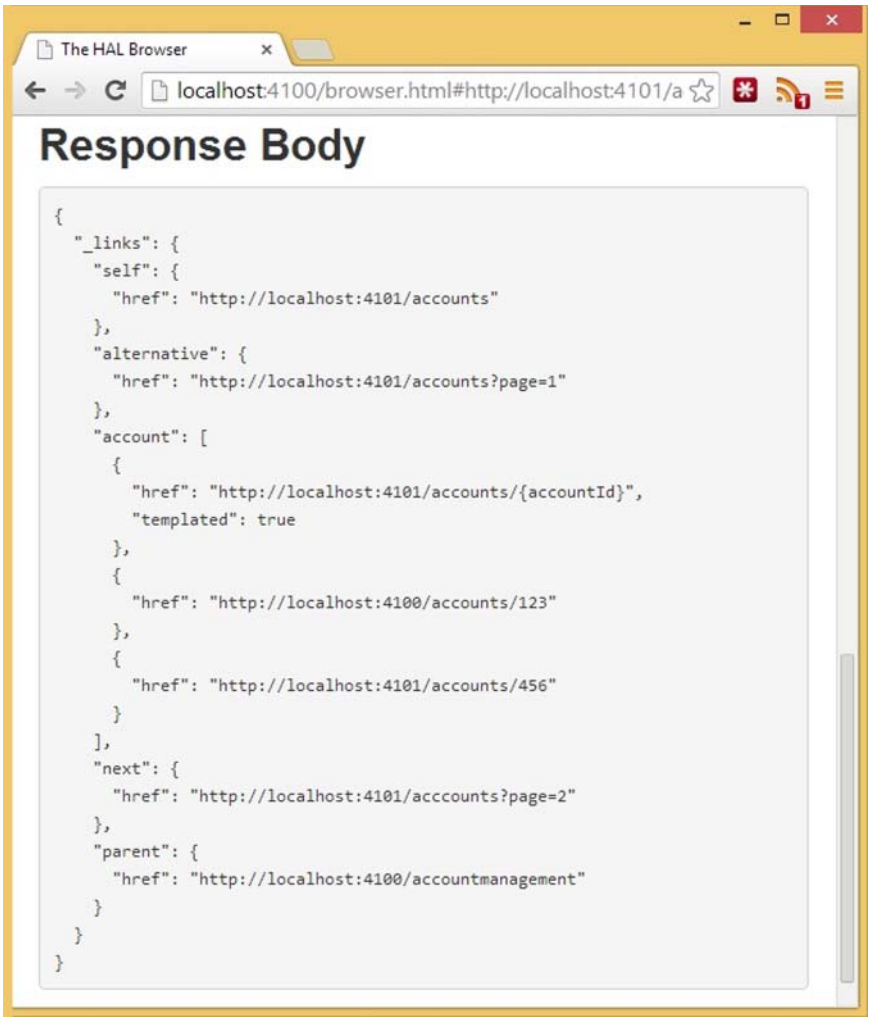


FIGURE 13-15: Following the Accounts link in the HAL browser.

URI templates are not all or nothing; you can combine them with normal links, even for the same resource. Figure 13-15 contains two nontemplated "account" links that illustrate this. They point directly to Account resources. For those links to work, though, you need to add an `Account()` method to the `AccountsController`. (This is determined by the route entry shown in Listing 13-11.)

An initial implementation of `Account()`, which returns just canned data, is shown in Listing 13-14. You can add this to your `AccountsController` below `Index()`. You also need to add the `AccountRepresentation` and `Account` classes from Listing 13-15. (You can put them all inside the `AccountsController.cs` file.)

LISTING 13-14: Account Method That Generates Account Resources

```
[HttpGet]
public AccountRepresentation Account(string accountId)
{
    // canned data.. for now
    return new AccountRepresentation
    {
        Href = AccountsBaseUrl + "accounts/" + accountId,
        Rel = "self",
        AccountId = accountId,
        Name = "Account_" + accountId,
        Links = new List<Link>
        {
            new Link
            {
                Href = AccountsBaseUrl + "accounts",
                Rel = "collection",
            },
            new Link
            {
                Href = AccountsBaseUrl + "accounts/" + accountId + "/followers",
                Rel = "followers",
            },
            new Link
            {
                Href = AccountsBaseUrl + "accounts/" + accountId + "/following",
                Rel = "following",
            },
            new Link
            {
                Href = AccountsBaseUrl + "accounts/" + accountId + "/blurbs",
                Rel = "blurbs",
            }
        }
    };
}
```

LISTING 13-15: Account Representation Class Used to Generate Account Resource

```
public class AccountRepresentation : Representation
{
    public string AccountId { get; set; }
    public string Name { get; set; }

    protected override void CreateHypermedia()
    {
    }
}
```

A resource's fields (as opposed to its links) are represented as standard JSON in an HTTP response. In Listing 13-15, you can see that an `AccountRepresentation` has the properties `AcountId` and

`Name`. Therefore, if you navigate to an Account resource using the Hal browser, you will see these properties represented as plain JSON. Figure 13-16 also shows this.

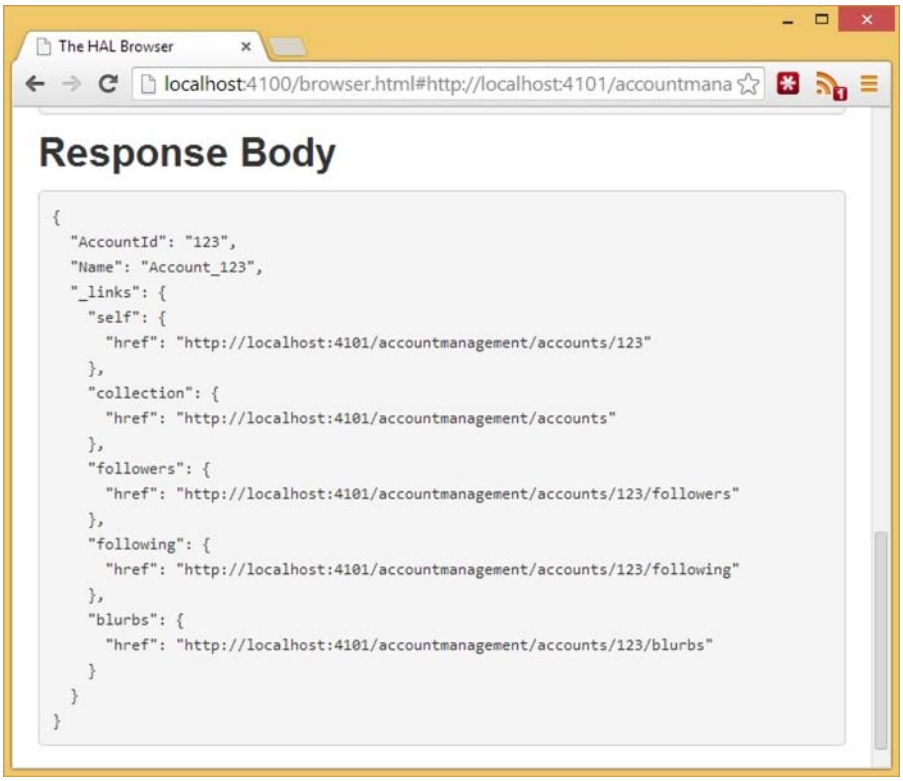


FIGURE 13-16: Resource's data fields are represented as plain JSON.

Figure 13-16 also shows some noteworthy links. In particular, the three links that point to child resources of this account are `followers`, `following`, and `blurbs`. You're going to build the followers endpoint in the next section, but the other two links are just to give you an idea of how a resource with many child resources could look. Building the followers endpoint is where you will start to build the event-driven parts of the system using the Event Store.

Persisting Events with the Event Store

Most of the infrastructure is in place for you to learn about building event-driven systems with REST. The first part of your learning involves storing events. There are many ways you can achieve this, such as writing events to a text file log or using a table in a SQL database. But in this example, you will see a purpose-built tool—the Event Store—that is the work of popular DDD practitioner Greg Young and his team (`www.geteventstore.com`).

> **NOTE** *It might be useful to refer to the design diagrams in Figure 13-9 to 13-11 to remind yourself of how all the pieces you've built fit so far fit together and what their purpose is. It's easy to lose sight of the bigger picture when you're focused on lower-level details.*

To learn about storing events, you need to build a new endpoint in your Accounts API that returns the followers of an account. This endpoint supports the ability to add new followers to the collection by posting to it. This flow was previously illustrated in Figure 13-11. As usual, the first step to creating a new endpoint for exposing resources is to start with the route definition. Listing 13-16 shows how your `WebApiConfig` should be updated to add the route definition for the Followers endpoint.

LISTING 13-16: Updated WebApiConfig with Followers Route Definition

```
using System;
using System.Web.Http;

namespace AccountManagement.Accounts.Api
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            config.MapHttpAttributeRoutes();

            // ... unchanged account route definitions as before

            // new route definition to be added
            config.Routes.MapHttpRoute(
               name: "Account Followers",
               routeTemplate: "accountmanagement/accounts/{accountId}/followers",
               defaults: new { controller = "Followers", action = "Index" }
            );
        }
    }
}
```

Listing 13-16 shows that the Followers resource will be served up by a method called `Index()` on a controller called `FollowersController`. You can create that controller by adding a class called `FollowersController`, in your project's `Controllers` folder, that resembles Listing 13-17.

LISTING 13-17: Initial Implementation of FollowersController

```
using System;
using System.Collections.Generic;
using System.Web.Http;
using WebApi.Hal;
```

(*continued*)

LISTING 13-17: *(continued)*

```
namespace AccountManagement.Accounts.Controllers
{
    public class FollowersController : ApiController
    {
        private const string AccountsBaseUrl =
                                "http://localhost:4101/accountmanagement/";

        [HttpGet]
        public FollowersRepresentation Index(string accountId)
        {
            return new FollowersRepresentation
            {
                Href = AccountsBaseUrl + "accounts/" + accountId + "/followers",
                Rel = "self",
                Links = new List<Link>
                {
                    new Link
                    {
                        Href = AccountsBaseUrl +
                                 "accounts/" + accountId + "/followers?pages=2",
                        Rel = "next",
                    },
                },
                followers = GetFollowers(accountId)
            };
        }

        private List<Follower> GetFollowers(string accountId)
        {
            // replace with DB lookup, etc. (in real application)
            return new List<Follower>
            {
                new Follower
                {
                    AccountId = "fl1",
                },
                new Follower
                {
                    AccountId = "fl2",
                },
                new Follower
                {
                    AccountId = "fl3",
                }
            };
        }
    }

    public class FollowersRepresentation : Representation
    {
        public List<Follower> followers { get; set; }

        protected override void CreateHypermedia()
```

```
        {
        }
    }

    public class Follower
    {
    public string AccountId { get; set; }
    }
}
```

The implementation of `Index()` in Listing 13-17 merely returns a canned response, parameterised with the passed-in Account ID. This is not the important part of this example—persisting an event in response to data being posted to the endpoint is the important and exciting part. You can see that process being triggered in Listing 13-18, which shows the code that needs to be added to your `FollowersController` directly below `GetFollowers()`. Also, you need to add the `BeganFollowing` class that is shown in Listing 13-19.

LISTING 13-18: Followers Controller Persisting Events for POST Requests

```
[HttpPost] // respond to POST requests only
[ActionName("index")] // Web API will not allow duplicate names
public IHttpActionResult IndexPOST(string accountId, Follower follower)
{
   // accountId will be taken from querystring - it is a simple type
   // follower will be taken from request body - it is a complex type

   var evnt = new BeganFollowing
   {
       AccountId = accountId,
       FollowerId = follower.AccountId
   };
   EventPersister.PersistEvent(evnt);
   return RedirectToRoute("Account Followers", new { accountId = accountId });
}
```

LISTING 13-19: Class Representing BeganFollowing Domain Event

```
// representing the domain event
public class BeganFollowing
{
    public string AccountId { get; set; }

    public string FollowerId { get; set; }
}
```

WARNING *Model-binding to domain classes or data structures used for persistence can lead to tight coupling. It is shown here for simplicity but is not recommended for important production applications.*

> **NOTE** *The Followers endpoint was added to the* `AccountManagement.Accounts.Api` *project for convenience. Depending on your scalability requirements, you may want to put child resources in their own project so you can scale them independently.*

`IndexPOST()`, shown in Listing 13-18, responds to post requests for `/accountmanagement/accounts/{accountId}/followers`. You can see this because the method is attributed with the `HttpPost` attribute. Because of another method called `Index` () in the same file, the `ActionName` attribute indicates that it should still respond to requests for the `Account Followers` route even though this method is named `IndexPOST()`.

The crucial event-persistence mechanics are not included in Listing 13-18. You can, though, see a call to `EventPersister.PersistEvent()`. This is the class that handles event persistence. You can see the contents of it in Listing 13-20, which shows the bare-minimum functionality for persisting events to the Event Store. You need to add this class to your project. For convenience, you can put it at the bottom of the `AccountsController.cs` file (but outside of the `AccountsController` class). Before adding the `EventPersister`, though, you need to install the Event Store C# client with the following command:

```
Install-Package EventStore.Client -Project AccountManagement.Accounts.Api
```

LISTING 13-20: EventPersister—A Custom Utility for Persisting Events to the Event Store

```
public static class EventPersister
{
    private static IPEndPoint defaultEsEndpoint =
     new IPEndPoint(IPAddress.Loopback, 1113);

    private static IEventStoreConnection esConn =
      EventStoreConnection.Create(defaultEsEndpoint);

    static EventPersister()
    {
        esConn.Connect();
    }

    public static void PersistEvent(object ev)
    {
        var commitHeaders = new Dictionary<string, object>
        {
            {"CommitId", Guid.NewGuid()},
        };

        esConn.AppendToStream(
            "BeganFollowing", ExpectedVersion.Any, ToEventData(Guid.NewGuid(),
            ev, commitHeaders
        ));
```

```
        }

        private static EventData ToEventData(Guid eventId, object evnt,
                                    IDictionary<string, object> headers)
        {
            var data = Encoding.UTF8.GetBytes(
                JsonSerializer.SerializeToString(evnt)
            );

            var metadata = Encoding.UTF8.GetBytes(
                JsonSerializer.SerializeToString(headers)
            );

            var typeName = evnt.GetType().Name;

            return new EventData(eventId, typeName, true, data, metadata);
        }
    }
```

For the `EventPersister` to compile, you need to include the following `using` statements:

```
using EventStore.ClientAPI;
using Newtonsoft.Json;
using System.Net;
using System.Text;
```

Of the code shown in Listing 13-20, there are two key details to focus on: the event is converted to JSON (and then binary), and it is appended to a stream—the `BeganFollowing` stream in this case. You will get a better understanding of how all of this works once the Event Store is up and running.

> **NOTE** *Event Sourcing is a deep and exciting topic. Chapter 22, "Event Sourcing," covers it in detail.*

Installing and Starting the Event Store

This example uses version 2.0.1 of the Event Store (`http://download.geteventstore.com/binaries/EventStore-OSS-Win-v2.0.1.zip`). Once you've downloaded it, you just need to extract the archive into a directory and then run the following PowerShell command from that directory (as Administrator):

```
./EventStore.SingleNode.exe --db .\ESData
```

To confirm that the Event Store has started up successfully, you should be able to access the management application by going to `http://localhost:2113`. As confirmation of success, you are presented with the welcome page shown in Figure 13-17. If you don't see the welcome screen, double-check that you ran PowerShell as Administrator. Also, look to see if there were any errors printed on the

PowerShell console. If you do see the welcome page, that means the Event Store is running and is now patiently waiting to persist all your events.

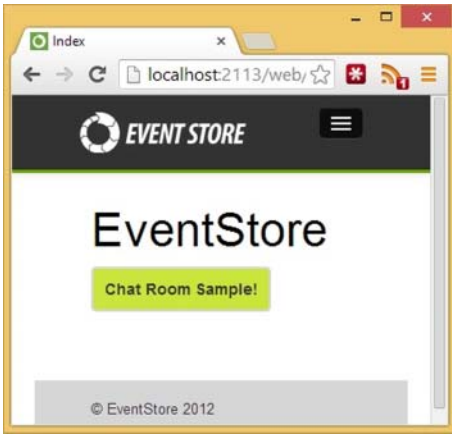


FIGURE 13-17: The Event Store's admin UI.

> **NOTE** *For more information about the Event Store, including running in different environments, such as Mono and EC2, the official docs on GitHub are the best place to look (*`https://github.com/eventstore/eventstore/wiki/Running-the-Event-Store#on-windows-and-net`*).*

> **NOTE** *The Event Store shuts down when you log out of Windows and does not restart. If you work through the examples in this chapter over multiple sessions, you need to run the same command to start the Event Store each time you log in to Windows.*

Viewing Persisted Events with the Event Store Admin UI

You just saw a glimpse of the Event Store admin UI, and now you will explore some of its key features as you view events that are being created via the Accounts API. To get events into the system, you need to post details of new followers. For demonstration purposes, you can do all this through the HAL browser by going to the entry point (`/accountmanagement` in the Hal browser) after you have started the system. You then need to follow the link to the Accounts resource. From the Accounts resource, you need to follow the link to one of the dummy accounts, and from there follow the link to its followers resource.

If you want to post to the followers resource, you need to click the orange button in the `NON-GET` column for the row that represents the `self` link. This button is shown in Figure 13-18. Clicking

this button opens a dialog enabling you to construct a JSON payload that will be posted to the endpoint. Figure 13-19 shows correctly formatted JSON being entered into this dialog containing the details of a new follower. Once you've entered some JSON, click the OK button, and your JSON is posted.



FIGURE 13-18: The NON-GET button in the HAL browser.

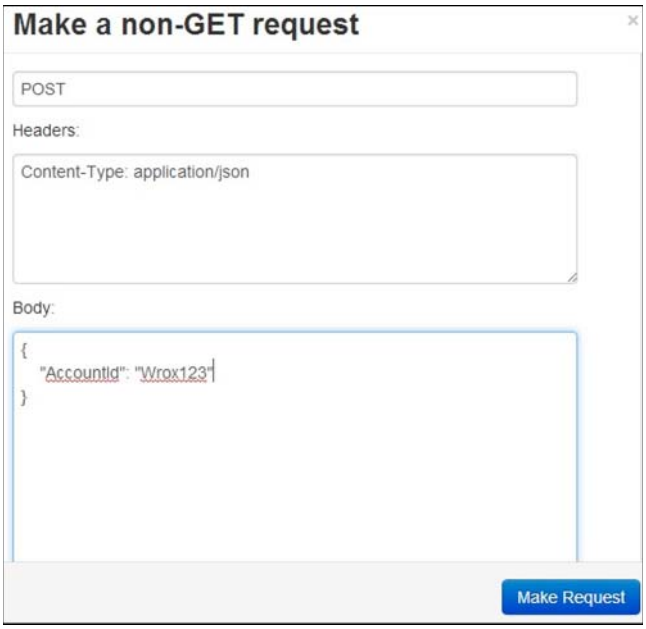


FIGURE 13-19: Constructing JSON on the NON-GET dialog in the HAL browser.

Providing you got a `200` response back from posting the JSON, you can now view the event using the Event Store's admin UI. After navigating to `http://localhost:2113/` and choosing the Streams menu item, you should see a stream called `BeganFollowing` that was created when you posted from the HAL browser. You can click on its name to view events in that stream. From there, you can inspect individual events, as shown in Figure 13-20.

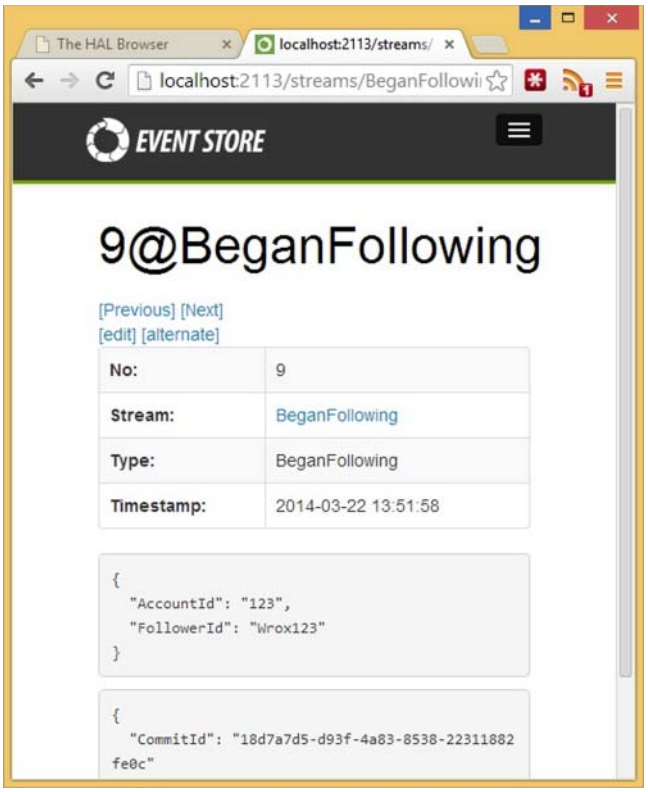


FIGURE 13-20: Viewing an event in the Event Store.

> **NOTE** *When you try to access some of the Event Store features in the web UI, you are asked for a password. The default username is* `admin`*, and the default password is* `changeit`*, as described on the Event Store wiki (*`https://github.com/EventStore/EventStore/wiki/Default-Credentials`*).*

Publishing Events to an Atom Feed

Atom is a discerning choice for exposing events in many RESTful systems because it is an extremely common format, as discussed earlier in the chapter. In this example, you will see how to use the tools baked into the .NET framework for creating and publishing an Atom feed.

Applications that publish events as an Atom feed are akin to message-publishing components in a messaging system. Accordingly, for an event-driven REST system, you can use a similar naming convention that communicates domain concepts, such as `{BoundedContext}.{BusinessComponent}.{Component}`.

To create the component that publishes the Began Following domain event, you can start by adding a new ASP.NET Web Application to the project called `AccountManagement.RegularAccounts.BeganFollowing`. You need to configure this application to run on port 4102 and make it a start-up project.

Creating a Basic Atom Feed in .NET

To create an Atom feed using official libraries that are part of the .NET work framework, you first need to add a reference to `System.ServiceModel` in the new `AccountManagement.RegularAccounts.BeganFollowing` project. After setting up a route definition, shown in Listing 13-21, you can then use classes from `System.ServiceModel` to create an Atom feed using events retrieved from the Event Store, as shown in Listing 13-22. The code in Listing 13-22 needs to be added as a new controller in the `Controllers` folder.

LISTING 13-21: BeganFollowing Route Entry

```
config.Routes.MapHttpRoute(
name: "Began Following",
       routeTemplate: "accountmanagement/beganfollowing",
       defaults: new { controller = "BeganFollowing", action = "Feed" }
);
```

LISTING 13-22: BeganFollowing Controller Producing an RSS Feed

```
using System;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using System.ServiceModel.Syndication;
using System.Xml;
using System.Text;
using System.IO;
using EventStore.ClientAPI;
using Newtonsoft.Json;
using System.Collections.Generic;
using System.Linq;

namespace AccountManageent.RegularAccounts.BeganFollowing
{
    public class BeganFollowingController : ApiController
    {
        private const string BeganFollowingBaseUrl = "http://localhost:4102/";

        [HttpGet]
        public HttpResponseMessage Feed()
        {
           // create feed
```

(*continued*)

LISTING 13-22: *(continued)*

```
            var feedUri = new Uri(BeganFollowingBaseUrl + "beganfollowing");
            var feed = new SyndicationFeed(
             "BeganFollowing", "Began following domain events", feedUri
            );
            feed.Authors.Add(
             new SyndicationPerson("accountManagementBC@WroxPPPDDD.com")
            );
            feed.Items = EventRetriever.RecentEvents("BeganFollowing")
                                       .Select(MapToFeedItem);

            // set feed as response - always atom+xml - no HAL
            var response = new HttpResponseMessage(HttpStatusCode.OK);
            response.Content = new StringContent(
             GetFeedContent(feed), Encoding.UTF8, "application/atom+xml"
            );
            return response;
        }

        private string GetFeedContent(SyndicationFeed feed)
        {
            using(var sw = new StringWriter())
            using(var xw = XmlWriter.Create(sw))
            {
                feed.GetAtom10Formatter().WriteTo(xw);
                xw.Flush();

                return sw.ToString();
            }
        }

        private SyndicationItem MapToFeedItem(ResolvedEvent ev)
        {
            return new SyndicationItem(
                "BeganFollowingEvent",
                Encoding.UTF8.GetString(ev.Event.Data),
                new Uri(
                  RequestContext.Url.Content("/beganfollowing/" +
                  ev.Event.EventId
                 )),
                ev.Event.EventId.ToString(),
                 DateTime.Now // Event store client does not return timestamp yet
            );
        }
    }
}
```

As you can see in Listing 13-22, an Atom feed is created using the `SyndicationFeed` class. The created feed is then set as the response of the HTTP request via an `XmlWriter`. On the response object, `application/atom+xml` is set as the content type. This will be passed directly as the value for the HTTP `Content-Type` response header. You can also see that individual events, retrieved from the

Event Store (`EventRetriever.RecentEvents()`), are converted into `FeedItems`. But what you can't see in Listing 13-22 is how to retrieve the events from the Event Store. That is shown next.

Retrieving Events from the Event Store

In Listing 13-22, individual feed items are generated by retrieving events from the Event Store using a custom utility class: `EventRetriever`. The contents of `EventRetriever` are shown in Listing 13-23 and need to be added to your project. To make life easy, you can pop it in the bottom of the file containing the `BeganFollowingController` if you don't want to create another file.

LISTING 13-23: EventRetriver—A Small Utility for Retrieving Events from Event Store

```
public static class EventRetriever
{
    private static IPEndPoint defaultEsEndpoint =
     new IPEndPoint(IPAddress.Loopback, 1113);

    private static IEventStoreConnection esConn =
     EventStoreConnection.Create(defaultEsEndpoint);

    static EventRetriever()
    {
        esConn.Connect();
    }

    public static IEnumerable<ResolvedEvent> RecentEvents(string stream)
    {
        var results = esConn.ReadStreamEventsForward(stream, 0, 20, false);
        return results.Events;
    }
}
```

`EventRetriever` is a utility class that wraps the Event Store C# client. It is hard-coded to retrieve the past 20 events, starting from the most recent. This is enabled by using `ReadStreamEventsForward`, which starts with the most recent events and works backward. There's a lot of functionality provided by the Event Store C# client that isn't covered in this book, so if you're thinking about using the Event Store and the C# client, the Event Store website contains lots of useful information.

For the `EventRetriever` to compile, you need to add a reference to the Event Store C# client in this project as well. The following command takes care of installing it for you:

```
Install-Package EventStore.Client -Project AccountManagement.RegularAccounts.
BeganFollowing
```

To test that your Atom feed is working as expected, you first need to update the entry point resource (in the `AccountManagement.EntryPoint.Api` project) to provide a link to the Atom feed (remember, clients should not be coupled to resources, only the entry point). Listing 13-24 shows the updated entry point resource containing the required link. Once the resource is added to your

project, you can test the feed by viewing it directly in a browser (accessing a resource directly is okay if you're just testing it): `http://localhost:4102/accountmanagement/beganfollowing`.

LISTING 13-24: Updated Entry Point with a Link to the BeganFollowing Atom Feed

```
return new EntryPointRepresentation
{
    Href = EntryPointBaseUrl + "accountmanagement",
    Rel = "self",
    Links = new List<Link>
    {
        new Link
        {
            Href = AccountsBaseUrl + "accountmanagement/accounts",
            Rel = "accounts",
        },
        new Link
        {
            Href = "http://localhost:4102/accountmanagement/beganfollowing",
            Rel = "beganfollowing"
        }
    }
};
```

Archiving Feeds

In a highly scalable system with potentially millions of users, there may be hundreds or thousands of events every second. Having a single Atom feed for all these events would quickly become unusable. This could result in a massive waste of network bandwidth, as well as other inefficiency-related issues. A common solution is to display a fixed number of events per-feed, and once a capacity is reached to then archive the feed. Importantly, each feed contains hypermedia links to the previous and next archives (if they exist). For more information, the Internet Engineering Task Force (IETF) has a request for comments (RFC) titled "Feed Paging and Archiving" (`https://tools.ietf.org/html/rfc5005`).

Creating an Event Subscriber/Atom Feed Consumer

Consuming an Atom feed that exposes domain events is akin to subscribing to messages in a messaging system. However, consuming an Atom feed inverts the process of receiving pushed messages by polling and pulling them instead. It's a little more work up-front for developers, but it definitely has compelling advantages.

When creating a consumer of an Atom feed, you can again take advantage of Atom's popularity by using official libraries in the .NET framework. You'll see this shortly as you build the first part of the Discovery bounded context that polls the Began Following Atom feed. The project you create for this does not need to be a web project. Instead, you can create a new C# Class Library called `Discovery.Recommendations.Followers` (as per the containers diagram in Figure 13-10). As before, to take advantage of .NET's really simple syndication (RSS) libraries, you need to add a reference to `System.ServiceModel`. You also need to configure this project as a start-up project.

> **NOTE** *If you're not using .NET, your platform will still likely contain libraries or open source projects that facilitate the creation of Atom feeds. Java, and thus Scala, for example, have the Rome Tools library (*`http://rometools.github.io/rome/RssAndAtOMUtilitiEsROMEV0.5AndAboveTutorialsAndArticles/RssAndAtOMUtilitiEsROMEV0.5TutorialUsingROMEToCreateAndWriteASyndicationFeed.html`*).*

Subscribing to Events by Polling

Inside the new polling component, the logic you are about to add consists of a few generic steps. These steps are likely to be similar in any Atom-feed polling application you build.

1. Fetch a batch of events starting from the last event ID that was processed (or the first item if no event has been processed yet).
2. Process each item in the batch according to domain policies.
3. Store the ID of the last event processed.

The first part of implementing the polling consumer for the Discovery bounded context is shown in Listing 13-25. This contains only the high-level logic. You can add all this code to your project inside a single class called `BeganFollowingPollingFeedConsumer` in the root of the project.

LISTING 13-25: High-Level Polling Logic for the BeganFollowingFeedConsumer

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace Discovery.Followers.BeganFollowingConsumer
{
    public class BeganFollowingPollingFeedConsumer
    {
        const string EntryPointUrl = "http://localhost:4100/accountmanagement";

        private static string LastEventIdProcessed;

        public static void Main(String[] args)
        {
            // probably not a production quality implementation
            while(true)
            {
                FetchAndProcessNextBatchOfEvents();
                Thread.Sleep(1000);
            }
```

(*continued*)

LISTING 13-25: *(continued)*

```
        }

        private static void FetchAndProcessNextBatchOfEvents()
        {
            var atomFeed = FetchFeed();
            var ups = GetUnprocessedEvents(atomFeed.Items.ToList());

            if (ups.Any())
                ProcessEvents(ups);
            else
                Console.WriteLine("No new events to  process");
        }
```

Listing 13-25 shows the first part of the feed consumer. It illustrates how a batch of events will be retrieved from the feed and processed. You can see polling is set to a maximum of once per second with the call to `Thread.Sleep()`.

Focus now shifts to the lower-level details of actually fetching the feed. This is shown in Listing 13-26; it's an example of a REST API client following links in hypermedia from an entry point to a target resource. You need to add this code directly below the code you added from Listing 13-25. You also need to add `ServiceStack.Text` to the project by running the following command:

```
Install-Package ServiceStack.Text -Project Discovery.Reccommendations.Followers
```

LISTING 13-26: A REST API Client Following Hypermedia Links to the Atom Feed

```
private static SyndicationFeed FetchFeed()
{
    using(var wc = new WebClient())
    {
        // get the feed URI from the entry point resource
        var rawEp = wc.DownloadString(EntryPointUrl);
        var hal = JsonSerializer.DeserializeFromString<HALResource>(rawEp);
        var feedUrl = hal._links["beganfollowing"].href;

        // parse a strongly typed syndication feed
        var rawFeed = wc.DownloadString(feedUrl);
        var feedXmlReader = XDocument.Parse(rawFeed).CreateReader();
        return SyndicationFeed.Load(feedXmlReader);
    };
}
```

The code in Listing 13-26 also depends on the classes in Listing 13-27 and the following `using` statements, which need to be added in the same file:

```
using ServiceStack.Text;
using System.Xml.Linq;
```

LISTING 13-27: HAL Client Models

```
public class HALResource
{
    public Dictionary<string, Link> _links { get; set; }
}

public class Link
{
    public string href { get; set; }
}
```

After fetching the feed, individual events can then be processed. A demonstrative implementation of this for the `BeganFollowingPollingFeedConsumer` is shown in Listing 13-28.

LISTING 13-28: Processing Each Event and Marking It as Processed

```
private static List<SyndicationItem> GetUnprocessedEvents(
    List<SyndicationItem> events)
{
    var lastProcessed = events.SingleOrDefault(s => s.Id == LastEventIdProcessed);
    var indexOfLastProcessedEvent = events.IndexOf(lastProcessed);

    return events.Skip(indexOfLastProcessedEvent + 1).ToList();
}

private static void ProcessEvents(List<SyndicationItem> events)
{
    foreach (var ev in events)
    {
        var evnt = ParseEvent(ev.Content);
        Console.WriteLine(
            "Processing event: " + evnt.AccountId + " - " + evnt.FollowerId
        );
        // Your domain rules here
        LastEventIdProcessed = ev.Id;
    }
}
```

Listing 13-28 demonstrates the generic high-level logic you would likely see in a feed consumer. First you fetch a batch of events from the feed, then you select the ones that have not yet been processed. In some cases where feeds are paged or archived, you may need to make additional requests, again using hypermedia, to locate the last event processed. After locating the events that are unprocessed, you then process each one according to your domain rules and update the ID of the last processed event.

You may be wondering how errors are handled during the processing of events. With REST-based integration, there is no out-of-the-box support for poison messages or transitive messages. This is covered in a touch more detail toward the end of the chapter.

To complete the example, you need to implement the remaining piece of lower-level logic, which parses events from the feed. This is shown in Listings 13-29 and 13-30. You also require a final pair of `using` statements:

```
using System.IO;
using System.Xml;
```

LISTING 13-29: Parsing Events from the Feed

```
private static BeganFollowing ParseEvent(SyndicationContent content)
{
    // reference to servicestack.text
    var jsonString = ParseFeedContent(content);
    var bf = JsonSerializer.DeserializeFromString<BeganFollowing>(jsonString);
    return bf;
}

private static string ParseFeedContent(SyndicationContent syndicationContent)
{
    using(var sw = new StringWriter())
    using(var xw = XmlWriter.Create(sw))
    {
        syndicationContent.WriteTo(xw, "BF", "BF");
        xw.Flush();

        return XDocument.Parse(sw.ToString()).Root.Value;
    }
}
```

LISTING 13-30: Model of Domain Event in the Consumer

```
public class BeganFollowing
{
    public string AccountId { get; set; }
    public string FollowerId { get; set; }
}
```

That wraps up the example for this chapter. Hopefully you've understood enough theory and seen enough examples to feel confident about considering REST as an option for bounded context integration on your projects.

All that remains for this example is to test that everything works. You can do that by POSTing new followers, as shown previously. Keep an eye on the console window that automatically pops up. You should see output similar to Figure 13-21. You can also access the Atom feed directly in the browser again to check the new events that appear on it.

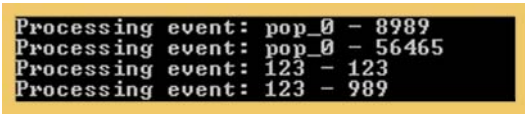


FIGURE 13-21: Feed consumer processing events

> **NOTE** *If there's anything further you want to know or anything that didn't make full sense, please feel free to post your thoughts at* `http://p2p.wrox.com/`.

Maintaining REST Applications

As with a messaging or any other system, you have to support the application after it has been initially deployed. This may involve versioning APIs as they evolve, monitoring how the system is performing, or capturing metrics that are used to inform business decisions.

Versioning

Small improvements to APIs can easily be achieved without breaking any existing clients. The key is to make sure changes are backward compatible. If you had an application that produced the Shipping Status resource:

```
{
    "totalLegs": 5,
    "legsCompleted": 3,
    "currentVesselId": "sst399",
    "nextVesselId": "u223a"
}
```

and wanted to add a new piece of information to it, you need only add the extra piece of information at the bottom like this:

```
{
    "totalLegs": 5,
    "legsCompleted": 3,
    "currentVesselId": "sst399",
    "nextVesselId": "u223a",
    "eta": "2014-09-01"
}
```

This is a backward-compatible change and is desirable because clients coupled to the old format do not break.

API overhauls are a more contentious topic. These occur when you want to make big or breaking changes to an API. You may want to remove resources, move information between resources, or completely change formats. The two most common versioning options are to include the version in the URI or in an HTTP header. Versioning a URI usually involves a prefix such as `/v2/account management/`. Alternatively, versioning with a header may involve using the HTTP `Version` header like this: `Version: 2`.

Monitoring and Metrics

A big benefit of using HTTP is that there are a lot of off-the-shelf monitoring tools you can simply plug in to your APIs to immediately have a whole host of metrics. New relic (`http://newrelic.com/`) is a popular choice but it is not free. Instead, or in combination, you may want to capture custom metrics.

In such cases, tools like StatsD (`https://github.com/etsy/statsd/`) and the C# StatsD client (`https://github.com/goncalopereira/statsd-csharp-client`) are popular options.

Drawbacks with REST for Bounded Context Integration

You've probably started to form your own opinions now, but REST is definitely an option for teams that want to build scalable, fault-tolerant systems without being coupled to messaging frameworks. Before you decide that REST is the right choice, though, it's important to discern a few of its drawbacks.

A number of REST's drawbacks when compared to messaging systems involve more development work up front. It can be a bigger initial effort to build scalable, fault-tolerant systems with REST. Most of the additional development work is to compensate for features that come out of the box with messaging solutions. But as you read through the list of drawbacks, keep in mind that over the lifetime of the project, the drawbacks may turn into advantages. You'll have fewer frameworks to manage, and you will be closer to the metal when it comes to understanding how your distributed system actually communicates.

> **NOTE** *Please keep in mind that the drawbacks in this section are specifically about using REST for asynchronous, event-driven applications. These drawbacks do not necessarily apply to the use of REST in other contexts.*

Less Fault Tolerance Out of the Box

Event-driven REST improves fault tolerance compared to RPC but lacks a little compared to messaging solutions. In the previous chapter, intent was captured to place an order and immediately stored. Any failures delivering the `PlaceOrder` command would simply result in the message being retried. That's not true for the REST system you built in this chapter. If the Event Store was unavailable during an attempt to store a Began Following event, there is no automatic recovery when the Event Store comes back online.

One option to improve fault tolerance is to add store-and-forward mechanisms yourself. This could involve adding queues in locations where fault tolerance is important to the business. Alternatively, you could try a high-availability approach by adding more instances of an application behind a load balancer or to a cluster. The Event Store supports clustering, so that's definitely a viable option for the example in this chapter.

To summarize, you have to work a bit harder to gain some of the fault tolerance benefits that messaging frameworks provide by default.

Eventual Consistency

Share-nothing, loosely coupled systems that communicate asynchronously are always going to have a high susceptibility to eventual consistency. Event-driven REST as recommended in this chapter definitely falls into that category. For example, when the Account Management bounded context exposes Began Following events, it has already stored them locally. But consumers who poll the feed

don't immediately get updated until they have polled the feed and processed the new events. So, depending on which API clients hit, they may or may not see information based on recent events.

Dealing with eventual consistency when integrating with REST relies on the same fundamental concepts as in a messaging system. You need to forego big transactions in favor of smaller ones. Also, you need to roll forward into new states. Finally, consider retrying messages a number of times in the hope that eventually they will succeed.

THE SALIENT POINTS

- HTTP's popularity makes it a serious candidate for integrating bounded contexts.
- Among its many benefits, HTTP leaves you completely free of any technology couplings, allowing you to mix and match technologies as you prefer.
- Using HTTP means you may be able to use the same set of APIs internally and externally.
- You can use HTTP in a number of ways; you can use it for RPC or event-driven REST.
- RPC can be a good choice for simple solutions, whereas event-driven REST can lead to better fault tolerance and scalability.
- With RPC over HTTP, you can use feature-rich but verbose SOAP or lightweight XML or JSON.
- REST is fundamentally about resources, hypermedia, and statelessness based on how the web works.
- REST takes full advantage of HTTP's conventions and capabilities.
- Domain events can be used as the messages in event-driven REST systems.
- Asynchronous polling of Atom feeds that contain lists of events provides the basis for event-driven REST applications.
- You can still use SOA's principles with REST to build loosely coupled systems and loosely coupled teams.
- HTTP requests and responses are the contract between bounded contexts. Try to avoid breaking changes, and aim for backward compatibility to avoid disrupting other teams.
- Whichever form of HTTP you use, take every sensible opportunity to make domain concepts explicit.

PART III

Tactical Patterns: Creating Effective Domain Models

- **CHAPTER 14:** Introducing the Domain Modeling Building Blocks
- **CHAPTER 15:** Value Objects
- **CHAPTER 16:** Entities
- **CHAPTER 17:** Domain Services
- **CHAPTER 18:** Domain Events
- **CHAPTER 19:** Aggregates
- **CHAPTER 20:** Factories
- **CHAPTER 21:** Repositories
- **CHAPTER 22:** Event Sourcing

14

Introducing the Domain Modeling Building Blocks

WHAT'S IN THIS CHAPTER?

- The role of tactical patterns in creating an effective object-oriented domain model
- Introducing value objects, entities, domain services, and modules to model your domain and behaviors
- An overview of the lifecycle patterns: aggregate, factory, and repository
- The emerging patterns of event sourcing and domain events

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 14 download and individually named according to the names throughout the chapter.

Mapping the implementation model back to the analysis model and ensuring they are bound to one another is hard. To guide developers and clarify designs, Evans has built upon the domain model pattern that was first catalogued in Martin Fowler's book *Patterns of Enterprise Application Architecture*. He introduces a pattern language containing a number of building block patterns to enable the creation of effective domain models. The patterns, built around best-practice, object-oriented techniques, are sometimes referred to as the tactical patterns of Domain-Driven Design (DDD). Many of the patterns themselves are not new, but Evans was the first to group them in this manner as an aid for developers to create effective domain models. This chapter gives a high-level introduction to the tactical building block patterns of

Domain-Driven Design. Each pattern has its own chapter in this part of the book where it is covered in more detail.

Although this chapter and the rest of this part detail techniques for creating domain models, the implementation tactics for building domain models should remain flexible and open to innovation. Evans's original text favored an object-oriented approach, but don't overlook the different modeling paradigms, as discussed in Chapter 5, "Domain Model Implementation Patterns." The domain events pattern, which you will read about later in this chapter, was not included in the original building blocks, something that Evans recently said he regretted. In addition, functional programming and event sourcing (covered in Chapter 18, "Domain Events") are becoming popular ways to express domain models.

The patterns used to create domain models and tie implementation to analysis have continually evolved since Evans's original text. The semantics of how you create domain models can and will change, what is important is to represent concepts in code using the language of the domain—the Ubiquitous Language.

TACTICAL PATTERNS

The role of the tactical patterns in DDD is to manage complexity and ensure clarity of behavior within the domain model. You use the patterns to capture and convey meaning, relationships, and logic within the domain. The patterns are built around solid object-oriented principles, and many are catalogued in widely regarded design books, namely *Patterns of Enterprise Application Architecture*, by Martin Fowler, and *Design Patterns: Elements of Reusable Object-Oriented Software*, by Ralph Johnson, John Vlissides, Richard Helm, and Erich Gamma.

Each building block pattern is designed to have a single responsibility; it could be to represent a concept in the domain like an entity or a value object, or it could be to ensure that the concepts of the domain are kept uncluttered from lifecycle concerns like the factory or repository objects. In a way, you can view the building blocks as a ubiquitous language (UL) for developers to use as a framework for constructing rich and useful domain models.

You can use numerous building block patterns, shown in Figure 14-1, in the creation of a domain model. Note that the application services pattern is a client of the domain model and is therefore not covered in this part of the book. Application services are covered in Chapter 25, "Commands: Application Service Patterns for Processing Business Use Cases."

PATTERNS TO MODEL YOUR DOMAIN

The following patterns represent the policies and logic within the problem domain. They express relationships between objects, model rules, and bind the detail of the analysis model to the code implementation model. These are the patterns that express the elements of your model in code.

Entities

An entity represents a concept in your domain that is defined by its identity rather than its attributes. Although an entity's identity remains fixed throughout its lifecycle, its attributes may change. An entity is responsible for defining what it means to be the same; in code this is often achieved by overriding the equality operations of a class.

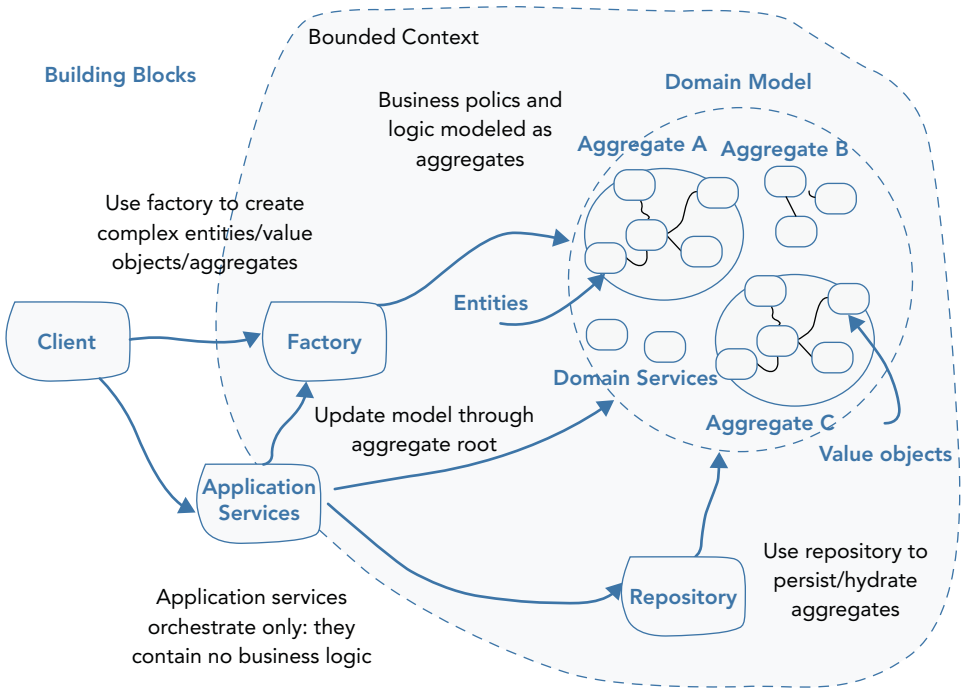


FIGURE 14-1: Tactical patterns—domain model building blocks.

An example of an entity is a product; its unique identity won't change once it is set but its description, price, etc., can be altered many times. Entities are mutable as the attributes can change.

Figure 14-2 shows the main concepts of an entity.

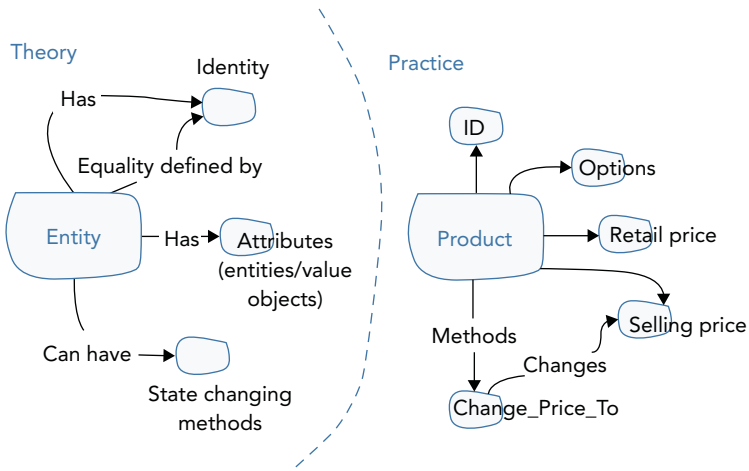


FIGURE 14-2: An entity.

In Listing 14-1, you see a product modeled as an entity.

LISTING 14-1: A Product Entity

```
public class Product : Entity<Guid>
{
    private readonly IList<Option> _options;
    private Price _selling_price;
    private Price _retail_price;

    public Product(Guid id, Price sellingPrice, Price retailPrice)
        : base(id)
    {
        _selling_price = sellingPrice;
        if (!selling_price_matches(retailPrice))
            throw new PricesNotInTheSameCurrencyException(
                "Selling and retail price must be in the same currency");

        _retail_price = retailPrice;
        _options = new List<Option>();
    }

    public void change_price_to(Price new_price)
    {
        if (!selling_price_matches(new_price))
            throw new PricesNotInTheSameCurrencyException(
                "You cannot change the price of this
                  product to a different currency");
        _selling_price = new_price;
    }

    public Price savings()
    {
        Price savings = _retail_price.minus(_selling_price);

        if (savings.is_greater_than_zero())
            return savings;
        else
            return new Price(0m, _selling_price.currency);
    }

    private bool selling_price_matches(Price retail_price)
    {
        return _selling_price.Equals(retail_price);
    }

    public void add(Option option)
    {
        if (!this.contains(option))
            _options.Add(option);
        else
            throw new ProductOptionAddedNotUniqueException(
                string.Format(
```

```
                "This product already has the option {0}",
                option.ToString()));
    }

    public bool contains(Option option)
    {
        return _options.Contains(option);
    }
}
```

Listing 14-1 shows that the identifier of a product is set on construction and there are no methods to change it. The `Product` entity delegates all work to the `Money` and `Option` value objects—the attributes/characteristics of the `Order`. The product entity encapsulates data and exposes behavior; the data of the class is hidden.

You will also have noticed that the entity has a generic base class that takes the type used for identification. Within the base class are the overridden equality methods, similar to what you saw in the `Money` value object. However, this time when comparing objects, you compare the type and the ID.

For completeness, Listing 14-2 shows the implementation of the `Entity` base class.

LISTING 14-2: Entity Base Class

```
public abstract class Entity<TId> : IEquatable<Entity<TId>>
{
    private readonly TId id;

    protected Entity(TId id)
    {
        if (object.Equals(id, default(TId)))
        {
            throw new ArgumentException(
                "The ID cannot be the default value.", "id");
        }

        this.id = id;
    }

    public TId Id
    {
        get { return this.id; }
    }

    public override bool Equals(object obj)
    {
        var entity = obj as Entity<TId>;
        if (entity != null)
        {
            return this.Equals(entity);
        }
```

continues

LISTING 14-2 *(continued)*

```
            return base.Equals(obj);
        }

        public override int GetHashCode()
        {
            return this.Id.GetHashCode();
        }

        public bool Equals(Entity<TId> other)
        {
            if (other == null)
            {
                return false;
            }
            return this.Id.Equals(other.Id);
        }
    }
```

Listing 14-2 shows that by inheriting from this base class, you keep the logic that determines equality between entities within the entity itself. The abstract base class keeps all the noise of identity and equality checking out of the implementation class so that it can focus on business logic.

Value Objects

Value objects represent the elements or concepts of your domain that are known only by their characteristics; they are used as descriptors for elements in your model; they do not require a unique identity. Because value objects have no conceptual identity within the model, they are defined by their attributes; their attributes determine their identity. Value objects don't need identity because they are always associated with another object and are therefore understood within a particular context. For instance, you may have an order entity that uses value objects to represent the order shipping address, items, courier information, and so on. Not one of these characteristics needs identity itself because it only has meaning within the context of being attached to an order. An order address that is not attached to an order does not have meaning. Value objects are comparable based on their attributes, and like entities are responsible for any equality checks.

Because they are defined by their attributes, value objects are treated as immutable; that is, once constructed, they can never alter their state. A good example of a value object is money. It doesn't matter that you can't distinguish between the same five one pound coins or one dollar bills in your pocket. You don't care about the currency's identity—only about its value and what it represents. If somebody swapped a five dollar bill for one you have in your wallet, it would not change the fact that you still have five dollars. Of course, in real life, money can have a unique identifier in the form of a serial number, but the domain model does not reflect real life. Instead, it is an abstraction of it built to fulfill the needs of use cases within the problem domain. Figure 14-3 shows the main concepts of a value object.

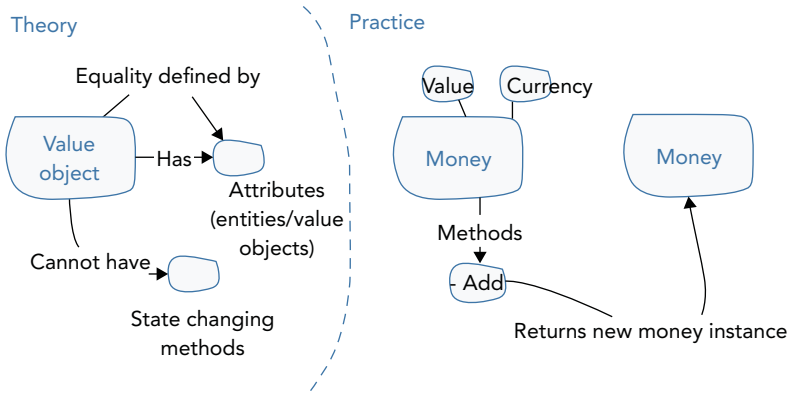


FIGURE 14-3: A value object.

Listing 14-3 shows money modeled as a value object.

LISTING 14-3: A Money Value Object

```
public class Money
{
    protected readonly decimal _value;
    private readonly Currency _currency;

    public Money()
        : this(0m, new SterlingCurrency())
    {
    }

    public Money(decimal value, Currency currency)
    {
        _value = value;
        _currency = currency;
    }

    public Money add(Money to_add)
    {
        if (this._currency.Equals(to_add._currency))
            return new Money(_value + to_add._value, _currency);
        else
            throw new NonMatchingCurrencyException(
                 "You cannot add money with different currencies.");
    }

    public Money minus(Money to_discount)
    {
        if (this._currency.Equals(to_discount._currency))
```

continues

LISTING 14-3 *(continued)*

```
            return new Money(_value - to_discount._value, _currency);
        else
            throw new NonMatchingCurrencyException(
                 "You cannot remove money with different currencies.");
    }

    public override string ToString()
    {
        return string.Format("{0}",
            _currency.format_for_displaying(_value));
    }

    // Equality overrides

    public static bool operator ==(Money money, Money money_to_compare)
    {
        if ((object)money == null)
        {
            return (object)money_to_compare == null;
        }

        return money._value == money_to_compare._value &&
                money._currency == money_to_compare._currency;
    }

    public static bool operator !=(Money money, Money money_to_compare)
    {
        return (money._value != money_to_compare._value ||
                money._currency != money_to_compare._currency);
    }

    public bool Equals(Money obj)
    {
        if (obj == null) return false;
        return obj._value == _value && obj._currency.equals(_currency);
    }

    public override bool Equals(System.Object obj)
    {
        if (obj == null) return false;
        if (obj.GetType() != this.GetType()) return false;

        return ((Money)obj)._value == _value &&
                            obj._currency.equals(_currency);
    }

    public override int GetHashCode()
    {
        return (_value.GetHashCode() + _currency.GetHashCode())
                            .GetHashCode();
    }
}
```

As you can see from Listing 14-3 the equality methods have been overridden, meaning that the `Money` object is compared only by its attributes; in this instance, the attributes are the value and the currency. The money class is immutable. Once it has been created, it cannot change state. The `add` method returns a new instance of a `Money` object. This is known as closure of operations because you are not altering the state of the original money object. Another value object is used to capture the concept of currency. You delegate to the currency object when comparing equality. Again, the equality methods have been overridden.

Domain Services

Domain services encapsulate domain logic and concepts that are not naturally modeled as value objects or entities in your model. Domain services have no identity or state; their responsibility is to orchestrate business logic using entities and value objects. A good example of a domain service is a shipping cost calculator as shown in Listing 14-4. This service is a business function that, given a set of consignments (value objects) and a collection of weight bandings, can calculate the cost of shipping. This functionality does not sit comfortably on a domain object, so it is better represented as a domain service.

LISTING 14-4: A Domain Service Shipping Cost Calculator

```
public class ShippingCostCalculator
{
    private IEnumerable<WeightBand> _weightBand;
    private readonly WeightInKg _boxWeightInKg;

    public ShippingCostCalculator(IEnumerable<WeightBand> weightBand,
                                  WeightInKg boxWeightInKg)
    {
        _weightBand = weightBand;
        _boxWeightInKg = boxWeightInKg;
    }

    public Money CalculateCostToShip(IEnumerable<Consignment> consignments)
    {
        var weight = GetTotalWeight(consignments);

        // Reverse sort list
        _weightBand = _weightBand.OrderBy(x =>
                x.ForConsignmentsUpToThisWeightInKg.Value);

        // Get first match
        var weightBand = _weightBand.FirstOrDefault(x =>
                x.IsWithinBand(weight));

        return weightBand.Price;
    }

    private WeightInKg GetTotalWeight(
                    IEnumerable<Consignment> consignments)
    {
```

continues

LISTING 14-4 *(continued)*

```
            var totalWeight = new WeightInKg(0m);

            // Calculate the weight of the items
            foreach (Consignment consignment in consignments)
                totalWeight = totalWeight.Add(consignment.ConsignmentWeight());

            // Add a box weight
            totalWeight = totalWeight.Add(_boxWeightInKg);

            return totalWeight;
        }
    }
```

The `ShippingCostCalculator` encapsulates the domain logic that calculates the shipping cost of consignment based on the weight of a collection of consignments, along with the weight of a box. By organizing this logic under a specific domain service and naming it, you can explicitly talk to domain experts about a particular piece of domain logic such as a policy or a process in a concise manner. The `ShippingCostCalculator` may have been an implicit concept held by the business, but by naming it, you have ensured it is now explicit and should be added to the UL and the analysis model.

Modules

Modules in C# are implemented as namespaces or projects. You use them to organize and encapsulate related concepts (entities and value objects) so you can simplify your understanding of larger domain models. Modules are used to decompose the domain model. Don't confuse them with subdomains that decompose the domain and bounded contexts that delimit the applicability of a domain model. As shown in Figure 14-4, module names are lifted straight from the UL and enable distinct parts of the domain model to be understood in isolation. Modules enable developers to quickly read and understand a domain model in code before digging deep into class files. They also act as a responsibility boundary, clearly defining parts of the domain model and ensuring that relationships between domain objects are kept to a minimum. Apply modules to promote low coupling and high cohesion within your domain model. Try limiting the contents of a module to a cohesive set.

LIFECYCLE PATTERNS

The following patterns deal with the creation and persistence of the objects that represent the structure of the domain.

Aggregates

Entities and value objects collaborate to form complex relationships that meet invariants within the domain model. When dealing with large interconnected associations of objects, it is often difficult to ensure consistency and concurrency when performing actions against domain objects. Figure 14-5 shows a large object graph. Trying to treat this collection of objects as one conceptual whole is difficult and could result in performance problems for an application. For example we would not want to block a customer updating an address on her account just because an earlier order's status is being changed at the same time. These two things are unrelated and need not share a consistency or concurrency boundary.

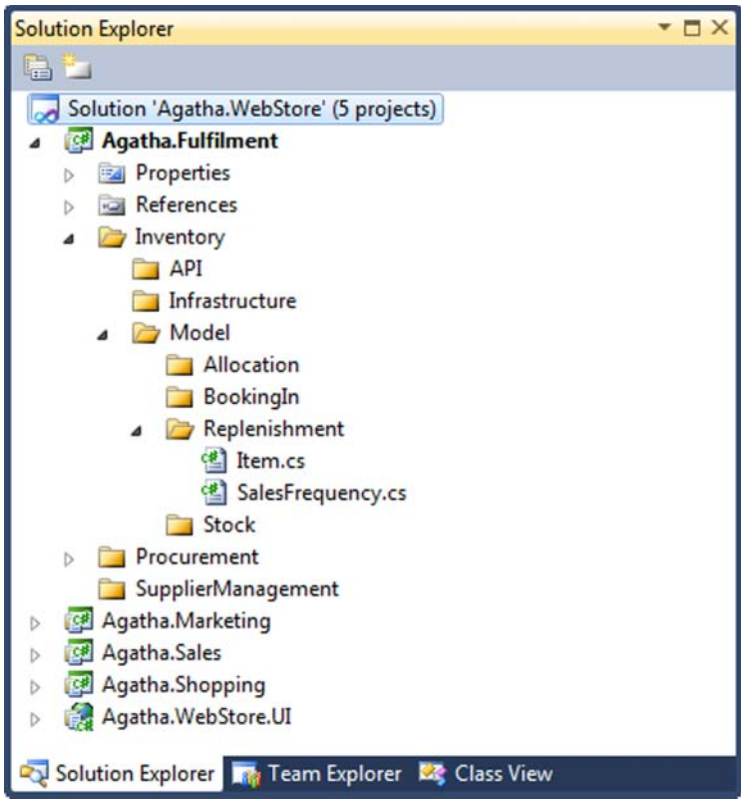


FIGURE 14-4: Modules used to organize domain concepts within a domain model.

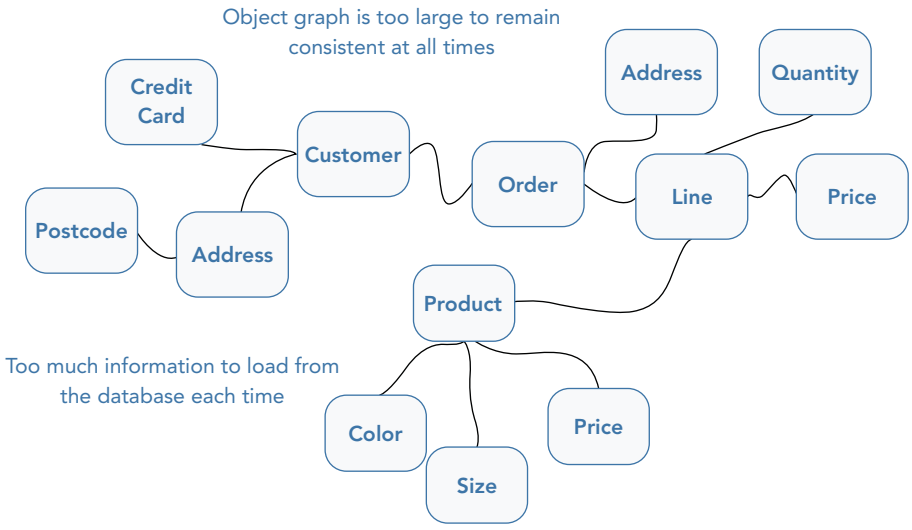


FIGURE 14-5: A large object graph.

Domain-Driven Design has the Aggregate pattern to ensure consistency and to define transactional concurrency boundaries for object graphs. Large models are split by invariants and grouped into aggregates of entities and value objects that are treated as a conceptual whole. As shown in Figure 14-6, you can distill the model into aggregates.

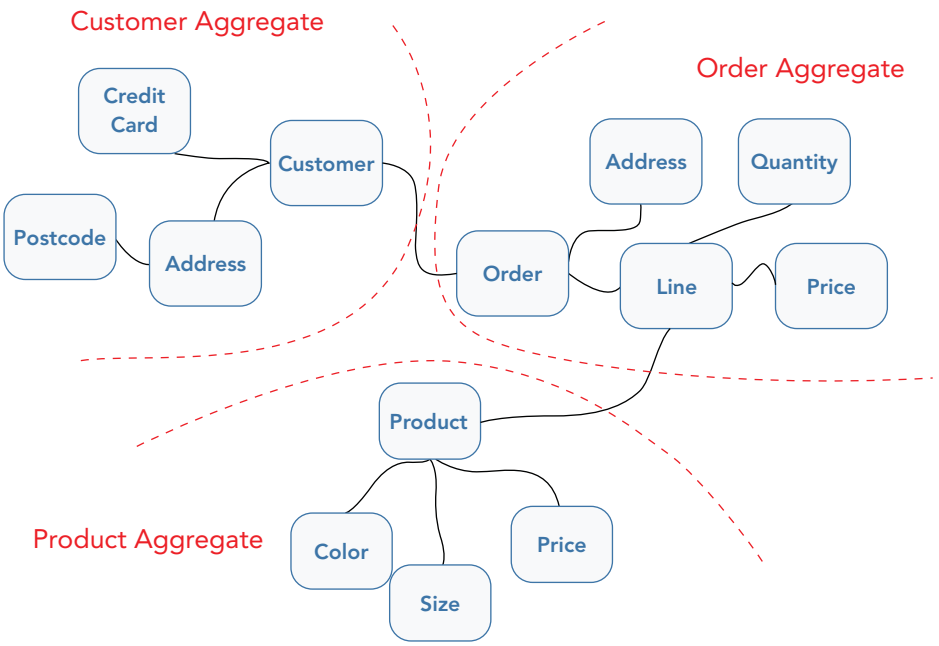


FIGURE 14-6: A large object graph split into aggregates.

Relationships between aggregate roots should be implemented by keeping a reference to the ID of another aggregate root and not a reference to the object itself, as shown in Figure 14-7. This principle helps to keep a boundary between aggregates and avoids the need to load large object graphs that are not required.

The aggregate groupings in Figure 14-6 and 14-7 at first glance look like a reasonable way to split the object graph; however, defining aggregate groups based purely on related concepts only goes some way to improve consistency and concurrency challenges. Take the customer aggregate, if an address is amended at the same time that some personal details of a customer are changed we could introduce blocking issues. These two concepts although related to a customer do not require an invariant. Therefore, we can split these into two separate aggregates and use this same logic for a credit card, then group them all under a customer module, as can be seen in Figure 14-8.

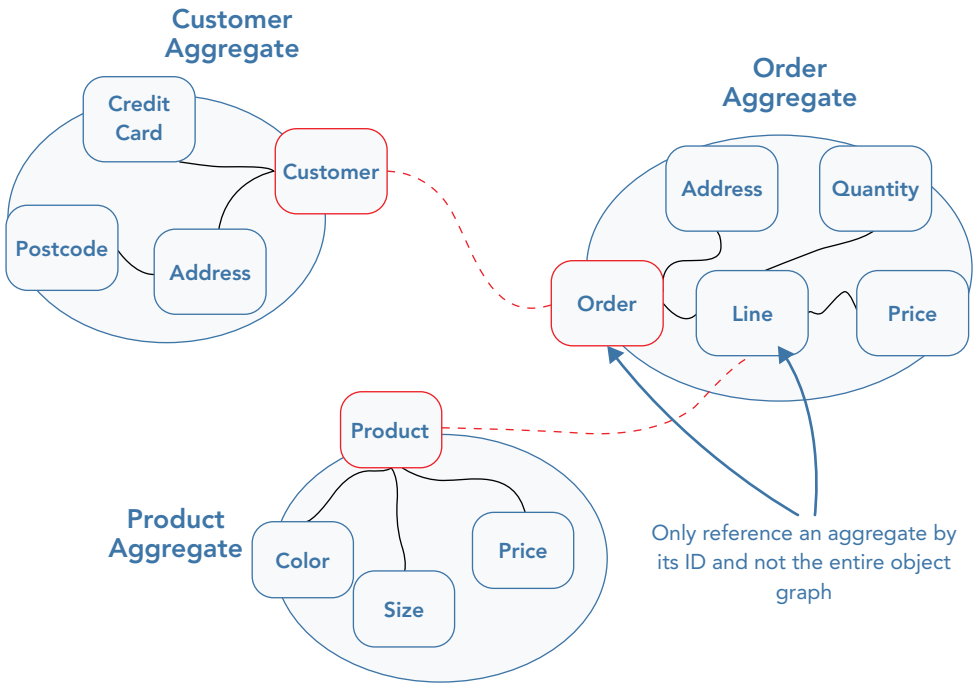


FIGURE 14-7: An aggregate root acts as the entry point to the aggregate.

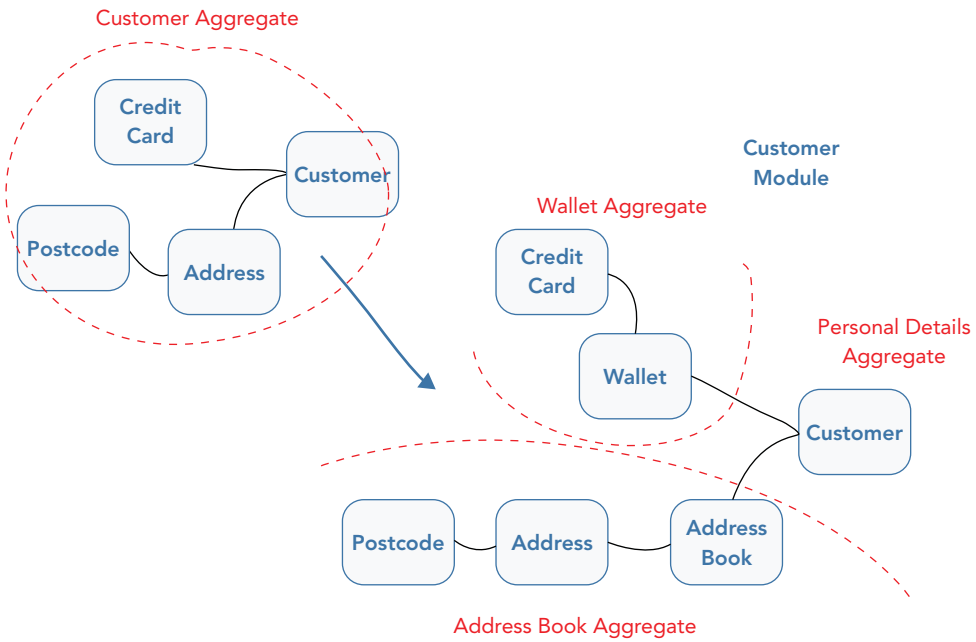


FIGURE 14-8: Aggregates should be based around invariants.

WHAT IS AN INVARIANT?

Invariants are rules that enforce consistency in the domain model. Whenever there is a change to an entity or aggregate, the business rules still apply. An example of an invariant is a rule stating that a customer must always have a complete address. To ensure that you adhere to this invariant, do not give the user the opportunity to independently edit lines of the address, enabling it to become invalid. Instead, treat the address perhaps as a value object to ensure that a change results in the invariant remaining valid.

An aggregate root, shown in Figure 14-9, acts as the entry point into the aggregate. No other entity or value object outside of the aggregate can hold a reference to an object within the aggregate. Objects outside the aggregate can only reference the aggregate root of another aggregate. Any changes to objects in the aggregate need to come through the root. The root encapsulates the data of the aggregate and only exposes behaviors to change it.

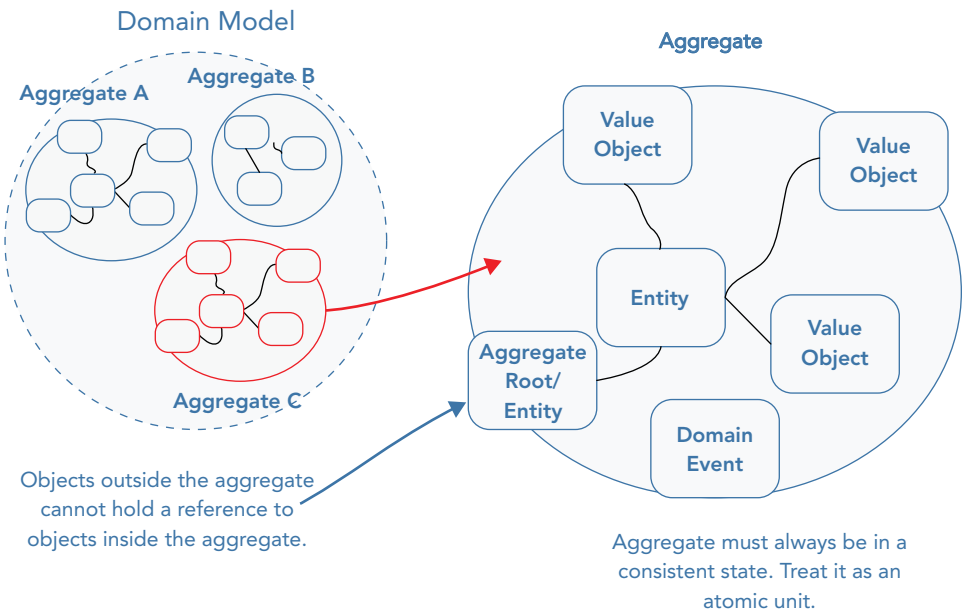


FIGURE 14-9: An aggregate root acts as the entry point to the aggregate.

Factories

If the creation of an entity or a value object is sufficiently complex, you should delegate the construction to a factory. A factory ensures that all invariants are met before the domain object

is created. If a domain object is simple and has no special rules for valid construction, favor a constructor method over a factory object. You can also use factories when re-creating domain objects from persistent storage.

In Listing 14-5, you see that a `Customer` entity has a factory method to enable an address to be created with a valid customer ID.

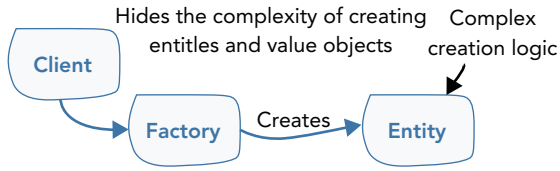


FIGURE 14-10: A factory.

LISTING 14-5: A Factory Method on an Entity

```
public class Customer : Entity<Guid>
{
    public Customer(Guid id)
        : base(id)
    {
    }

    // ...

    public Address create(Address delivery_address)
    {
        return new Address(Guid.NewGuid(), this.Id);
    }
}
```

Repositories

A domain model needs a method for persisting and hydrating an aggregate. Because an aggregate is treated as an atomic unit, you should not be able to persist changes to an aggregate without persisting the entire aggregate. A repository, as shown in Figure 14-11, is a pattern that abstracts the underlying persistence store from the model allowing you to create a model without thinking about infrastructure concerns. The repository is the mechanism that you should use to retrieve and persist aggregates. For view rendering, a repository is not required, and querying against a data store is the most efficient method for reporting needs. A repository is an infrastructure concern, so it is not always necessary to abstract away the

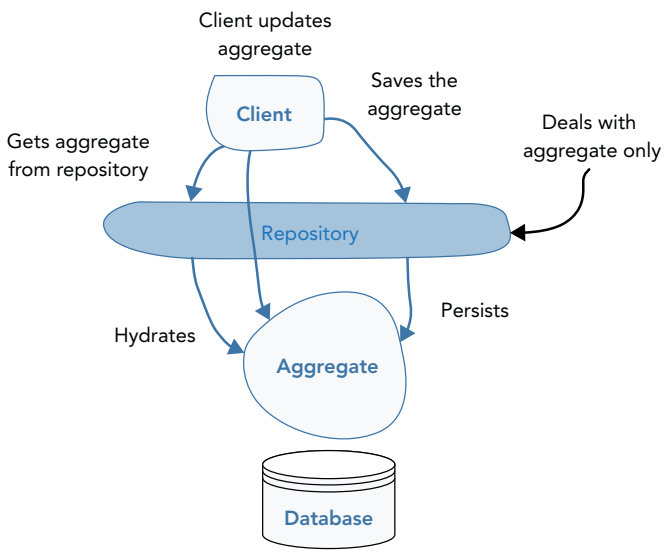


FIGURE 14-11: A repository.

underlying framework doing all the hard work. It can be more worthwhile to lean on ORM (Object Relational Mapper) frameworks to act as a repository; examples include NHibernate, RavenDB, and Entity Framework. Many developers get hung up on this pattern. Think of it simply as a method of persistence and rehydration. Treat it like infrastructure, and don't get hung up on abstracting it away.

EMERGING PATTERNS

Since Eric Evans' original text, two patterns have emerged that are useful for creating domain models. Namely the domain events pattern, covered in Chapter 18, and the event sourcing pattern, covered in Chapter 22 ("Event Sourcing").

Domain Events

Domain events signify something that has happened in the problem domain that the business cares about. You can use events to record changes to a model in an audit fashion, or you can use them as a form of communication across aggregates. Often an operation on a single aggregate root can result in side effects that are outside the aggregate root boundary. Other aggregates within the model can listen for events and act accordingly.

For example, consider the basket within an e-commerce site, as shown in Figure 14-12. Every time a customer places an item in a basket, it's important to update the recommended products that are displayed on the site. In this scenario, after a domain event is raised with details of the basket, a customer's recommendations are modified. The event is subscribed to by the recommendation's bounded context. Without using a domain event, you need to explicitly couple the basket bounded context to the recommendation context. Domain events give a more natural flow of communication and focus on the when.

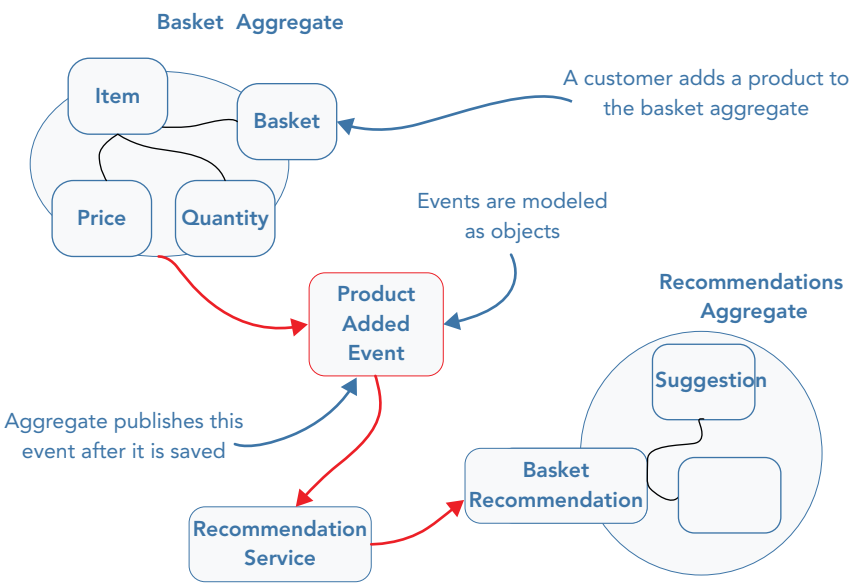


FIGURE 14-12: A domain event.

In Listing 14-6, you see a domain event being published from a basket.

LISTING 14-6: Domain Events being raised on a basket entity

```
namespace Domain.Shopping
{
    public class Basket
    {
        private BasketItems _items;
        private Guid _id;

        // ....

        public void add(Product product)
        {
            _items.Add(BasketItemFactory.CreateItemFor(product, this));

            DomainEvents.raise(new ProductAddedToBasket(this._id,
                                    _items.ids_of_items_in_basket()));
        }
    }

    public class ProductAddedToBasket
    {
        public Guid basket_id { get; private set; }
        public IEnumerable<Guid> items_in_basket { get; private set; }

        public ProductAddedToBasket(Guid basket_id,
                                    IEnumerable<Guid> items_in_basket)
        {
            basket_id = basket_id;
            items_in_basket = items_in_basket;
        }
    }

    // .....
}
```

In Listing 14-7, you see a recommendation service handling the event raised by the basket.

LISTING 14-7: A Domain Event Handler

```
namespace Domain.Recommendations
{
    public class RecommendationsEventHandlers
    {
        // ......

        public void Handle(Shopping.ProductAddedToBasket product_added)
        {
            _recommendation_service.update_suggestions_for(
```

continues

LISTING 14-7 *(continued)*

```
                product_added.basket_id, product_added.items_in_basket);
            }
        }
    }
```

Don't worry too much about the syntax of the code at this stage; it is covered in Chapter 18. Domain events play a crucial role in the code.

Event Sourcing

A popular alternative to traditional snapshot-only persistence is event sourcing. Instead of storing the state of an entity in a database, you store the series of events that lead up to the state. Storing all of the events increases the analytical capabilities of a business. Instead of just asking what the current state of an entity is, a business can ask what the state was at any time in the past, as shown in Figure 14-13.

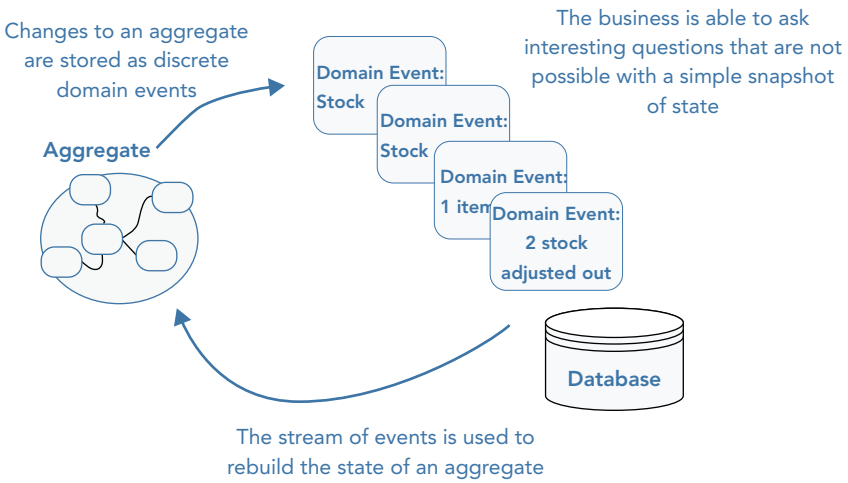


FIGURE 14-13: Storing events, not snapshots

Being able to query the state of your domain model at any time in the past provides a competitive business advantage because you can correlate events that have occurred in the real world with changes to the state of your domain model. Online travel agents may want to investigate why the number of bookings had a massive dip in a certain month. With event sourcing, they can rebuild the state of their catalog and re-run the searches that their users made to understand why those users did not find a vacation they wanted.

As a DDD practitioner, not only is your responsibility to suggest the possibility of using event sourcing to the business, but you also have to learn new ways of modeling your domain to support event sourcing. In particular, you need to move away from dumping an entity's state into a database using an ORM. Instead, you need to find a way of modeling, capturing, and persisting domain

events. You may even need to add event store functionality to an existing database. All of these concerns are covered in detail in Chapter 22.

THE SALIENT POINTS

- The tactical patterns of DDD are Evans's building blocks based on Martin Fowler's patterns for use when creating an Object-Oriented Domain model.
- The building blocks are guides to creating effective domain models, but they are only guides. The way you implement domain models can vary greatly, so don't get too hung up on the building block patterns.
- Entities
  - Are defined by their identity.
  - Identity remains constant throughout its lifetime.
  - Are responsible for equality checks.
- Value Objects
  - Describe the properties and characteristics within the problem domain.
  - Have no identity.
  - Are immutable, meaning that they cannot be changed. Instead properties modeled as value objects must be replaced.
- Domain Services
  - Contain domain logic that can't naturally be placed in an entity or value object, whereas application services orchestrate the execution of domain logic but don't actually implement it.
  - Have no internal state, so you can call it repeatedly with the same input, and it always gives the same output.
- Modules
  - Are used to decompose, organize, and increase the readability of the domain model.
  - Namespaces, an implementation of modules, can be applied to reduce coupling and increase cohesion within the domain model.
  - Enable readers to quickly understand the design of a model.
  - Help to define clear boundaries between domain objects.
  - Encapsulate concepts that can be understood independently of each other. They operate on a higher level of abstraction than aggregates and entities.
- Aggregates
  - Decompose large object graphs into small clusters of domain objects to reduce the complexity of the technical implementation of the domain model.

- Represent domain concepts, not just generic collections of domain objects.
- Are based around domain invariants.
- Are a consistency boundary to ensure the domain model is kept in a reliable state.
- Ensure transactional concurrency boundaries are set at the right level of granularity to ensure a usable application by avoiding blocking at the database level.

- Factories
  - Separate use from construction.
  - Encapsulate complex entity and value object construction.
- Repositories
  - Expose the interface of an in-memory collection of aggregate roots.
  - Should not be used for reporting.
  - Provide the retrieval and persistence needs of aggregate roots.
  - Decouple the domain layer from database strategies and infrastructure code.
- Domain Events
  - Are significant occurrences in the real-world problem domain that business users care about; they are part of the ubiquitous language (UL).
  - Are an emerging design pattern that makes domain events more explicit in code.
  - Are akin to publish-subscribe, where events are raised and event handlers handle them.
- Event Sourcing
  - Replaces traditional snapshot-only storage with a full history of events that produce the current state.
  - Allows powerful querying capabilities that revolve around time, known as temporal queries.

15

Value Objects

WHAT'S IN THIS CHAPTER?

- An introduction to the value object DDD modeling construct
- A discussion of what value objects are and when to use them
- Patterns that have emerged for working with value objects
- Options available for persisting value objects with NoSQL and SQL databases

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 15 download and individually named according to the names throughout the chapter.

A domain model contains entities, which are akin to characters in a movie. And just like characters in a movie, it's often their attributes that make entities interesting or useful. You might note that an attribute of James Bond is that he has a high level of charisma. Equally, you might find a `BankAccount` entity interesting if it has a large `Balance` attribute. Modeling these important descriptive attributes is the role of a Domain-Driven Design (DDD) construct known as the *value object*.

Value objects have no identity. They are purely for describing domain-relevant attributes of entities, usually in the form of some quantity. Having no identity to deal with often makes working with value objects relatively pain free and enjoyable. In particular, being immutable and combinable are two characteristics that support their ease of use. You will learn about these characteristics, along with other pertinent ones, in this chapter. You will also see some common value object modeling patterns that promote usability and expressiveness.

Working with value objects can still be challenging at times because there are a few considerations that require deeper thinking: notably, persistence, validation, and primitive avoidance. This chapter covers these challenges, too, to ensure that you know everything you need to be able to create value objects of your own.

WHEN TO USE A VALUE OBJECT

Value objects are an entity's state, describing something about the entity or the things it owns. A ship may have a maximum cargo capacity, a grocery may have a stock level, and a financial report may have quarterly turnovers. Each of these relationships would likely be modeled as an entity-value object relationship. For each example, note how the value object represents a particular concept that has a measurement, magnitude, or value—hence, value object. Two key rationales make value objects an important technical construct in DDD.

Representing a Descriptive, Identity-Less Concept

If you look purposefully at value objects, like `Money` and others shown throughout this chapter, you will see a logical commonality. It would not make sense for them to have an identity. Only entities may have an identity. A basic example of this is a bank account; you would almost certainly need to look up a `BankAccount` entity by its ID, but would you look up its balance by its ID? In most domains, you wouldn't because the balance has no meaning or importance in isolation.

Listing 15-1 shows a `BankAccount` entity whose balance is represented by a `Money` object. After the previous discussion, hopefully it's clear why this modeling decision is sensible.

LISTING 15-1: Using a Value Object to Represent a Concept That Has No Identity

```
public class BankAccount
{
    public BankAccount(Guid id, Money startingBalance)
    {
        this.Id = id;
        this.Balance = startingBalance;
    }

    public Guid Id { get; private set; }

    public Money Balance { get; private set; }

    ...
}

// value object
public class Money
{
    public Money(int amount, Currency currency)
    {
        this.Amount = amount;
        this.Currency = currency;
```

```
    }

    private int Amount { get; set; }

    private Currency Currency { get; set; }

    ...
}
```

When a concept lacks an apparent identity, it's a big clue that it should be a value object in your model. Later in the chapter, you will gain a better understanding of this when further examples of value objects are shown. In particular, understanding their defining characteristics is key to being successful with value objects.

Enhancing Explicitness

DDD is all about explicitly communicating important business rules and domain logic. Conversely, primitive types like integers and strings in isolation aren't great at this. Although it is possible to represent descriptive concepts with primitive types, most DDD practitioners strongly discourage it. Instead, primitives should be wrapped up into cohesive value objects that explicitly represent the concepts they are modeling.

Listing 15-2 shows an object representing the current winning bid in an online auction. To represent the price of the bid, in money, an integer is used.

LISTING 15-2: Object That Relies on Unclear Primitives

```
public class WinningBid
{
    ...

    public int Price { get; private set; }

    ...
}
```

Representing the amount of the winning bid as an integer in Listing 15-2 is a sub-optimal design choice for two major reasons. One reason is that the integer does not express what a price is in this domain—it doesn't restrict inputs to the allowed range of values, and it doesn't express the unit of measurement or currency. This is a massive source of ambiguity that hides important details of the domain.

By modelling the winning bid amount as an integer, there is also a big risk that related domain concepts will be scattered throughout the domain rather than being cohesively co-located, because you cannot add behavior to primitive classes (nor would it make sense to).

In Listing 15-3, the `Price` value object shows that in the online auction domain, the rules for incrementing the winning bid for a price are significant. You can see that the benefit of the `Price` value object is to cohesively group all the related behaviors of a price: `BidIncrement()` and `CanBeExceededBy()`. This helps to express and enforce the rules of the price concept in this domain. Whenever there is a discussion involving prices, developers and domain experts need only look at this one single class to understand what a price is and what rules apply to it.

LISTING 15-3: Using Value Objects to Be Explicit

```
public class Price
{
    public Price(Money amount)
    {
        if (amount == null)
            throw new ArgumentNullException("Amount cannot be null");

        Amount = amount;
    }

    public Money Amount { get; private set; }

    public Money BidIncrement()
    {
        if (Amount.IsGreaterThanOrEqualTo(new Money(0.01m))
            && Amount.IsLessThanOrEqualTo(new Money(0.99m)))
                return Amount.add(new Money(0.05m));

        if (Amount.IsGreaterThanOrEqualTo(new Money(1.00m))
            && Amount.IsLessThanOrEqualTo(new Money(4.99m)))
                return Amount.add(new Money(0.20m));

        if (Amount.IsGreaterThanOrEqualTo(new Money(5.00m))
            && Amount.IsLessThanOrEqualTo(new Money(14.99m)))
                return Amount.add(new Money(0.50m));

        return Amount.add(new Money(1.00m));
    }

    public bool CanBeExceededBy(Money offer)
    {
        return offer.IsGreaterThanOrEqualTo(BidIncrement());
    }

}
```

As Listing 15-3 shows, by wrapping primitives with domain concepts, the type system does the mapping from data to domain concept for you, and the lack of clarity goes away. Consequently, your conversations with domain experts can be clearer and domain concepts will be more apparent to anyone reading the code. In addition, all functionality related to a single concept is modelled cohesively. In this instance, the rules for incrementing a price and determining if a price can be exceeded by another price are the cohesive grouping of behaviors related to the price concept that should and do live together.

> **NOTE** *You will see in Chapter 16, "Entities" that pushing behavior into value objects keeps entities focused. In the winning bid scenario, the behaviors related to price concept may have been forced into the* `WinningBid` *class and bloated it with additional responsibilities.*

Listing 15-3 also illustrates how value objects should often be fine-grained. The `Price` value object itself pushes the related concern of representing money into the `Money` value object, using an instance of `Money` to represent the amount of the winning bid. You will the see `Money` value object later in this chapter.

DEFINING CHARACTERISTICS

Being mostly self-contained is what makes value objects fundamentally easier to work with. Like proponents of functional programming, DDD practitioners are fond of value objects because they are immutable, side effect free, and easily testable. As you will see in the following examples, they do have a few other defining characteristics that are important to be aware of.

Identity-Less

As you learn about value objects, the single most important detail to remember is that they have no identity. Value objects are important because they tell you something about another object. How tall is a `Person` entity? How many years have there been no claims on an `InsurancePolicy` entity? What is the weight of a `Fruit` entity? Hopefully you can discern from these examples why having an identity would make no sense.

Because they have no identity and describe other concepts in the domain, normally you will uncover entities first, and from there realize the types of value objects that are relevant to them. Generally, value objects need the context of an entity to be relevant.

Comparing value objects is a crucial operation, though, even though they have no identity. This is enabled through attribute- or value-based equality.

> **NOTE** *Technically, value objects may have IDs using some database persistence strategies. This is purely a technical detail of persisting them and does not equate to having an identity in the domain. A range of options for persisting value objects is shown at the end of this chapter.*

Attribute-Based Equality

Entities are considered equal if they have the same ID. Conversely, value objects are considered equal if they have the same value.

If two `Currency` value objects represent the same amount of money, they are considered equal, regardless of whether each variable points to the same object or a different one. Likewise, if two `Temperature` value objects represent 30 degrees Celsius, the same applies. Listing 15-4 illustrates a `Meters` value object used in the domain model of a personal fitness system that tracks how much distance has been covered. Listing 15-5 then shows test cases demonstrating its attribute-based equality.

LISTING 15-4: Value Object with Attribute-Based Equality

```
public class Meters
{
    public Meters(decimal distanceInMeters)
    {
        if (distanceInMeters < (decimal)0.0)
            throw new DistancesInMetersCannotBeNegative();

        this.DistanceInMeters = distanceInMeters;
    }

    protected decimal DistanceInMeters { get; private set; }

    public Yards ToYards()
    {
        return new Yards(DistanceInMeters * (decimal)1.0936133);
    }

    public Kilometers ToKilometers()
    {
        return new Kilometers(DistanceInMeters / 1000);
    }

    public Meters Add(Meters meters)
    {
        return new Meters(
             this.DistanceInMeters + meters.DistanceInMeters
        );
    }

    public bool IsLongerThan(Meters meters)
    {
        return this.DistanceInMeters > meters.DistanceInMeters;
    }

    public override bool Equals(object obj)
    {
        var m = obj as Meters;
        if (m == null) return false;

        return ToTwoDecimalPlaces(m.DistanceInMeters)
            == ToTwoDecimalPlaces(DistanceInMeters);
    }

    private decimal ToTwoDecimalPlaces(decimal distanceInMeters)
    {
        return Math.Round(
            distanceInMeters, 2, MidpointRounding.AwayFromZero
        );
    }
}
```

LISTING 15-5: Test Cases Demonstrating Attribute-Based Equality

```
[TestMethod]
public void Same_distances_are_equal_even_if_different_references()
{
    var oneMeter = new Meters((decimal)1);
    var oneMeterX = new Meters((decimal)1);
    Assert.AreEqual(oneMeter, oneMeterX);

    var fiftyPoint25 = new Meters((decimal)50.25);
    var fiftyPoint25X = new Meters((decimal)50.25);
    Assert.AreEqual(fiftyPoint25, fiftyPoint25X);
}
```

Listing 15-4 shows how the `Meters` value object overrides `Equals()` to implement attribute-based equality. By default, C# (and other languages like Java) will consider two objects to be equal if they point to the same object reference or pointer. With a value object, you don't care if the two references point to the same object, you care about whether they represent the same domain-relevant value. In Listing 15-4, you can see the implementation of `Equals()` fulfils this obligation by returning true, indicating the two objects are equal, if they represent the same distance in meters to two decimal places (precision will vary based on domain rules and context).

Listing 15-4 shows only the minimum amount of code necessary to demonstrate attribute-based equality. To get full equality support in C#, you actually need to override a few other methods that the compiler and run time require for various scenarios. Implementing all these methods in all your value objects can get a bit tiresome, especially when it is unnecessary. As shown in Listing 15-6, you can employ a base class to prevent unnecessary boredom and repetition.

LISTING 15-6: ValueObject Base Class That Encapsulates Equality and Identity Boilerplate

```
public abstract class ValueObject<T> where T : ValueObject<T>
{
    protected abstract IEnumerable<object> GetAttributesToIncludeInEqualityCheck();

    public override bool Equals(object other)
    {
        return Equals(other as T);
    }

    public bool Equals(T other)
    {
        if (other == null)
        {
            return false;
        }
        return GetAttributesToIncludeInEqualityCheck()
               .SequenceEqual(other.GetAttributesToIncludeInEqualityCheck());
    }

    public static bool operator ==(ValueObject<T> left, ValueObject<T> right)
```

(*continued*)

LISTING 15-6 *(continued)*

```
    {
        return Equals(left, right);
    }

    public static bool operator !=(ValueObject<T> left, ValueObject<T> right)
    {
        return !(left == right);
    }

    public override int GetHashCode()
    {
        int hash = 17;
        foreach (var obj in this.GetAttributesToIncludeInEqualityCheck())
            hash = hash * 31 + (obj == null ? 0 : obj.GetHashCode());

        return hash;
    }
}
```

To ensure that your classes support all the native comparison operations in C#, you need to implement `Equals()`, `GetHashCode()`, and the operator overloads `==` and `!=`. Using a base class like the one shown in Listing 15-6 makes your life easy by doing all this for you. All that's left for you to do is implement `GetAttributesToIncludeInEqualityCheck()` in each sub class, as shown in Listing 15-7. If you are prepared to add another layer of complexity, you can investigate reflection-based alternatives.

LISTING 15-7: Alternative Implementation of Value Object Using Base Class

```
public class Meters : ValueObject<Meters>
{
    ...

    protected override IEnumerable<object>
GetAttributesToIncludeInEqualityCheck()
    {
        return new List<Object> { DistanceInMeters };
    }
}
```

Listing 15-7 shows an alternative implementation of `Meters` with the same semantics and behavior. In addition it also supports `==` and `!=`. This is a satisfying improvement over having to repeatedly apply the boilerplate that the `ValueObject` base class now manages.

> **NOTE** *Some programming languages have inherent support for value object-like classes. In Scala, for example, case classes have built-in support for attribute-based equality. It's definitely worth investigating what is possible with your chosen language.*

Behavior-Rich

As much as possible your value objects should expose expressive domain-oriented behavior and encapsulate state. This was demonstrated with Listing 15-4 where the `Meters` value object exposed the behaviors: `ToFeet()`, `ToKilometers()`, `Add()`, and `IsLongerThan()`. Importantly, note also how it encapsulated its `DistanceInMeters` state. As a general rule, all primitive values should be private or protected by default. Only when you have a very good reason should you break encapsulation and make them public. But first consider adding a method to the value object that provides the required functionality.

As you will see repeated throughout this book. Focusing on behavior to create behavior-rich domain models is one of the most crucial aspects of creating and evolving a domain model, so that domain concepts are explicit. This applies equally to value objects and other types of domain objects that are introduced later in the book.

Cohesive

As a descriptive concept, usually describing something with a quantity, value objects often cohesively encapsulate the value of measurement and the unit of measurement. You've seen this already with the `Height` value object, which encapsulates the size and the unit of measurement. Similarly, you also saw this with the `Money` value object, which encapsulates both the amount and the currency.

It's not always the case that being cohesive means encapsulating a value and unit of measurement, though. It could be any number of fields. For instance, a `Color` value object may have a `Red`, a `Green`, and a `Blue` property.

Immutable

Once created, a value object can never be changed. Instead, any attempts to change its values should result in the creation of an entirely new instance with the desired values. This is because immutability is usually easier to reason about, with fewer dangerous side effects.

A classic example of immutability, and generally a good role model for value objects, is .NET's `DateTime` class. The unit tests in Listing 15-8 exemplify how calling any operations that appear to mutate the object's state, including `AddMonths()` and `AddYears()`, actually return a completely new `DateTime` object.

LISTING 15-8: DateTime Is a Good Example of Immutability

```
[TestClass]
public class DateTime_immutability_specs
{
    [TestMethod]
    public void AddMonths_creates_new_immutable_DateTime()
    {
        var jan1st = new DateTime(2014, 01, 01);
        var feb1st = jan1st.AddMonths(1);

        // first object remains unchanged
```

(*continued*)

LISTING 15-8 *(continued)*

```
        Assert.AreEqual(new DateTime(2014, 01, 01), jan1st);

        // second object is a new immutable instance
        Assert.AreEqual(new DateTime(2014, 02, 01), feb1st);
    }

    [TestMethod]
    public void AddYears_creates_new_immutable_DateTime()
    {
        var jan2014 = new DateTime(2014, 01, 01);
        var jan2015 = jan2014.AddYears(1);
        var jan2016 = jan2015.AddYears(1);

        // first object remains unchanged
        Assert.AreEqual(new DateTime(2014, 01, 01), jan2014);

        // second object remains unchanged
        Assert.AreEqual(new DateTime(2015, 01, 01), jan2015);

        Assert.AreEqual(new DateTime(2016, 01, 01), jan2016);
    }
}
```

Implementing immutability yourself is easy. First, you decide how you are going to expose state. One option is to have `readonly` instance variables, as shown in Listing 15-9. The other approach is to have properties that are set in the constructor and never altered, as previous listings demonstrated. Both approaches are common, and it's usually just a matter of personal preference or project conventions. Sometimes, though, frameworks may force you into one choice or the other.

In addition to the state itself, you expose descriptive method names that express the creation of the new value in domain-oriented terms. For `DateTime`, those methods include the previously discussed `AddMonths()` and `AddYears()`; for the `Money` value object in Listing 15-9, the methods are `Add()` and `Subtract()`.

LISTING 15-9: Making Value Objects Immutable

```
public class Money : ValueObject<Money>
{
    protected readonly decimal Value;

    public Money() : this(0m)
    {
    }

    public Money(decimal value)
    {
        Value = value;
    }

    public Money Add(Money money)
```

```
    {
        return new Money(Value + money.Value);
    }

    public Money Subtract(Money money)
    {
        return new Money(Value - money.Value);
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new List<Object>() { Value };
    }
}
```

The `Money` value object in Listing 15-9 has two methods that sound like they mutate its state: `Add()` and `Subtract()`. Looking at the implementation tells a different story, though, because both methods return a new instance of `Money` with the updated value, while the original instance is left completely unchanged. Being immutable also supports combinability, as the next example illustrates.

Combinable

Values are often represented numerically, so in a lot of cases, they can be combined to create a new value. As you saw in the previous example, `Money` can be added to `Money` to create a new amount. Combinability like this is a defining characteristic of value objects in general. So when you are with domain experts and they talk about combining two instances of a certain concept, this is a clear sign that you may need to model the concept as a value object.

> **NOTE** *Combinable in this context means the values of two value objects can be combined. Remember, though, that value objects are immutable, and the result will be a new value object with a desired value. The original objects remain unchanged.*

Through representing a value or quantity, in most cases value objects can be combined using operations like addition, subtraction, and multiplication, as the previous immutability example demonstrated. For enhanced expressiveness, you can override these native operations, on a per-object basis, in many programming languages. Listing 15-10 shows how the `Money` value object can be updated to support the plus `+` and minus `-` operators in C#.

LISTING 15-10: Native Combinability of Value Objects

```
public class Money : ValueObject<Money>
{
    protected readonly decimal Value;

    public Money() : this(0m)
    {
```

(*continued*)

LISTING 15-10 *(continued)*

```
    }

    public Money(decimal value)
    {
        Value = value;
    }

    public Money Add(Money money)
    {
        return new Money(Value + money.Value);
    }

    public Money Subtract(Money money)
    {
        return new Money(Value - money.Value);
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new List<Object>() { Value };
    }

    public static Money operator + (Money left, Money right)
    {
        return new Money(left.Value + right.Value);
    }

    public static Money operator - (Money left, Money right)
    {
        return new Money(left.Value - right.Value);
    }
}
```

Overriding the + and - operators is light work in C#, as Listing 15-10 shows. You can see unit tests in Listing 15-11 that demonstrate using these operators to combine immutable `Money` objects into new instances.

LISTING 15-11: Combining Immutable Value Objects into New Instances

```
[TestClass]
public class Combining_money_tests
{
    [TestMethod]
    public void Money_supports_native_addition_syntax()
    {
        var m = new Money(200);
        var m2 = new Money(300);

        var combined = m + m2;

        Assert.AreEqual(new Money(500), combined);
```

```
    }

    [TestMethod]
    public void Money_supports_native_subtraction_syntax()
    {
        var m = new Money(50);
        var m2 = new Money(49);

        var combined = m - m2;

        Assert.AreEqual(new Money(1), combined);
    }
}
```

Self-Validating

Value objects should never be in an invalid state. They themselves are solely responsible for ensuring this requirement. In practice, this means that when you create an instance of a value object, the constructor should throw an exception if the arguments are not in accordance with domain rules. As an example, if you are modeling money with a `Money` value object in an e-commerce application, there may be two important domain rules:

- All money is accurate to two decimal places.
- All money must be a positive value.

These two rules apply to all instances of `Money` in the domain and should never be violated. Listing 15-12 shows a `Money` value object that enforces these constraints in its constructor, ensuring it can never be created in an invalid state.

LISTING 15-12: Self-Validating Value Object

```
public class Money : ValueObject<Money>
{
    protected readonly decimal Value;

    public Money() : this(0m)
    {
    }

    public Money(decimal value)
    {
        Validate(value);

        Value = value;
    }

    private void Validate(decimal value)
    {
        if (value % 0.01m != 0)
            throw new MoreThanTwoDecimalPlacesInMoneyValueException();

        if (value < 0)
```

(*continued*)

LISTING 15-12 *(continued)*

```
            throw new MoneyCannotBeANegativeValueException();
    }

    public Money Add(Money money)
    {
        return new Money(Value + money.Value);
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new List<Object>() { Value };
    }
}
```

The first thing that happens in `Money`'s constructor is the call to `Validate()`, which enforces the important self-validating characteristic of value objects. Implementation-wise, you don't have to go to a lot of effort. As Listing 15-12 shows, in many cases, all you need to check is that the state of the value object is within the allowable range. If it's not, you just throw a descriptive exception and abort construction as soon as possible.

Other coding patterns do exist for enforcing validation, though, depending on your preferences. Each pattern's flexibility and expressiveness can vary based on your context, so it's worthwhile knowing about them. The first alternative pattern is to validate inside a factory method, as shown in Listing 15-13.

LISTING 15-13: Validating Value Objects with a Factory Method

```
public class Money : ValueObject<Money>
{
    // ..

    public static Money Create(decimal amount)
    {
        if (amount % 0.01m != 0)
            throw new MoreThanTwoDecimalPlacesInMoneyValueException();

        if (amount < 0)
            throw new MoneyCannotBeANegativeValueException();

        return new Money(amount);
    }
}
```

Notice in Listing 15-13 how `Create()` is `static`. This is the factory method that is used as an alternative to validating in `Money`'s constructor. You may want to use this pattern when value objects can be created in different states based on the context. For example, in some scenarios, it may be fine to allow negative money, but not in others.

Using factory methods does mean that you can bypass validation altogether and create an instance with the `new` keyword. So use it with caution. If you have a situation in which different contexts

require different validation rules, you should definitely check out the introduction to micro or tiny types later in this chapter.

Another pattern for validating value objects is to use code contracts, which trade off additional technical complexity with greater expressiveness and fluency. You can use code contracts to supplement either of the previous patterns mentioned. Listing 15-14 illustrates using code contracts with constructor validation.

LISTING 15-14: Self-Validating with Code Contracts in the Constructor

```
public class Name : ValueObject<Name>
{
    public readonly string firstName;
    public readonly string surname;

    public Name(string firstName, string surname)
    {
        Check.that(firstName.is_not_empty()).on_constraint_failure(() =>
        {
            throw new ApplicationException("You must specify a first name.");
        });

        Check.that(surname.is_not_empty()).on_constraint_failure(() =>
        {
            throw new ApplicationException("You must specify a surname.");
        });

        this.firstName = firstName;
        this.surname = surname;
    }

    // ..
}
```

If you do prefer code contracts, you can minimize the additional complexity by creating reusable contracts and helper objects. Listing 15-15 shows a few basic code contract utilities that support the example in Listing 15-14. You are free to build a library of these helpers yourself or consider existing code contracts libraries such as the official Microsoft offering (`http://msdn.microsoft.com/en-us/library/dd264808(v=vs.110).aspx`).

LISTING 15-15: Reusable Code Contract Helpers

```
public static class StringExtensions
{
    public static bool is_not_empty(this String string_to_check)
    {
        return !String.IsNullOrEmpty(string_to_check);
    }
}

public class CheckConstraint
```

(*continued*)

LISTING 15-15 *(continued)*

```
{
    private readonly bool _assertion;

    public CheckConstraint(bool assertion)
    {
        _assertion = assertion;
    }

    public void on_constraint_failure(Action onFailure)
    {
        if (!_assertion) onFailure();
    }
}

public sealed class Check
{
    public static CheckConstraint that(bool assertion)
    {
        return new CheckConstraint(assertion);
    }
}
```

Testable

Immutability, cohesion, and combinability are three qualities of value objects that make them easy to test in expressive domain-oriented language. Immutability precludes the need to use mocks or verify side effects, cohesion allows single concepts to be fully tested in isolation, and combinability allows you to express the relationships between different values.

Throughout this section, you've already seen a number of examples of unit tests. But they were shown in the context of other characteristics, so you may not have taken the time to appreciate how low friction they are. Listing 15-6 is another short example showing the testability of value objects, demonstrating how error conditions can easily be tested without mocking.

LISTING 15-16: Value Objects Are Easy to Test

```
[TestClass]
public class Name_validation_tests
{
    [TestMethod]
    public void First_names_cannot_be_empty()
    {
        try
        {
            var name = new Name("", "Torvalds");
        }
        catch (ApplicationException e)
        {
            Assert.AreEqual("You must specify a first name.", e.Message);
```

```
            return;
        }

        Assert.Fail("No ApplicationException was thrown");
    }

    [TestMethod]
    public void Surnames_cannot_be_empty()
    {
        try
        {
            var name = new Name("Linus", "");
        }
        catch (ApplicationException e)
        {
            Assert.AreEqual("You must specify a surname.", e.Message);
            return;
        }

        Assert.Fail("No ApplicationException was thrown");
    }
}
```

By observing the tests in Listing 15-16, it is easy to discern the lower levels of friction when compared to testing other objects like application services. This is mainly due to the lack of side effects and mutability. You'll often hear proponents of functional programming lauding these characteristics. In essence, value objects themselves are a functional concept (or at least very close).

> **NOTE** *In the context of functional programming, the fancy term for describing side-effect-free methods or functions is* referential transparency. *For many programmers, learning about referential transparency and functional programming is worthwhile. Languages like F#, Scala, and Haskell are continuing to gain in popularity. For a good introduction, the Haskell wiki has an accessible article on the essence and motivations of functional programming (*`http://www.haskell.org/haskellwiki/Functional_programming`*).*

COMMON MODELING PATTERNS

DDD practitioners have built up a small collection of patterns over the years that improve the experience of working with value objects. Mostly, the benefits are aimed at improving expressiveness and clarity, but some have other slight benefits, including maintainability. This section presents three basic patterns so that you can immediately start to use them and start to think about patterns of your own, too.

Static Factory Methods

Using static factory methods is a popular technique for wrapping the complexities of object construction behind a simpler, more expressive interface. An excellent example of this is .NET's `TimeSpan` class with its `FromDays()`, `FromHours()`, and `FromMilliseconds()` static factory

methods. These alternatives are more expressive and less ambiguous than the five-integer-parameter constructor, as Listing 15-17 illustrates.

LISTING 15-17: Static Factory Methods Can Be More Expressive and Can Hide Complexity

```
[TestMethod]
public void TimeSpan_factory_methods()
{
    var sixDays = TimeSpan.FromDays(6);
    var threeHours = TimeSpan.FromHours(3);
    var twoMillis = TimeSpan.FromMilliseconds(2);

    var sixDaysx = new TimeSpan(6, 0, 0, 0, 0);
    var threeHoursx = new TimeSpan(0, 3, 0, 0, 0);
    var twoMillisx = new TimeSpan(0, 0, 0, 0, 2);

    Assert.AreEqual(sixDays, sixDaysx);
    Assert.AreEqual(threeHours, threeHoursx);
    Assert.AreEqual(twoMillis, twoMillisx);
}
```

Static factory methods are a stylistic choice that you are free to choose and ignore as you wish. Listing 15-18 shows the change that is made to the previously shown `Height` value object by having a static factory method for each unit of currency. In this case, the code is arguably more expressive, easier for clients to call, and more maintainable because clients of the code no longer need to couple themselves to the `MeasurementUnit` enum.

LISTING 15-18: Static Factory Methods Can Be More Maintainable and Expressive

```
public class Height
{
    public Height(int size, MeasurmentUnit unit)
    {
        this.Size = size;
        this.Unit = unit;
    }

    public int Size { get; private set; }

    public MeasurmentUnit Unit { get; private set; }

    // ..

    public static Height FromFeet(int feet)
    {
        return new Height(feet, MeasurmentUnit.Feet);
    }

    public static Height FromMetres(int metres)
    {
        return new Height(metres, MeasurmentUnit.Metres);
    }
}
```

Micro Types (Also Known as Tiny Types)

Avoiding primitives can help you be more explicit about the intent of your code by reducing error-causing ambiguity. This was already discussed previously in this chapter. A pattern called micro types takes this principle even further by wrapping already-expressive types with even more expressive types. To clarify, with micro types, the types being wrapped do not have to be primitives; they can already be explicit concepts that wrap primitives. This is useful, arguably, because it adds contextual clarity that can reduce errors.

An example of micro types is presented initially with the `OvertimeCalculator` domain service shown in Listing 15-19. Note the two parameters of type `HoursWorked` and `ContractedHours`.

LISTING 15-19: Domain Service That Accepts Only Micro Types

```
public class OvertimeCalculator
{
    public OvertimeHours Calculate(HoursWorked worked, ContractedHours contracted)
    {
        var overtimeHours = worked.Hours - contracted.Hours;
        return new OvertimeHours(overtimeHours);
    }
}
```

Both the `HoursWorked` and `ContractedHours` types used in Listing 15-19 are micro types that wrap an `Hours` value object. `OvertimeHours` is the return micro type, which is also just a contextual wrapper for `Hours`. All four class definitions are shown in Listing 15-20.

LISTING 15-20: Micro Type That Wraps an Already-Expressive Value Object

```
// main underlying value object
public class Hours : ValueObject<Hours>
{
    public readonly int Amount;

    public Hours(int amount)
    {
        this.Amount = amount;
    }

    public static Hours operator - (Hours left, Hours right)
    {
        return new Hours(left.Amount - right.Amount);
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { Amount };
    }
```

(*continued*)

LISTING 15-20 *(continued)*

```
}


// micro types

public class HoursWorked : ValueObject<HoursWorked>
{
    public readonly Hours Hours;

    public HoursWorked(Hours hours)
    {
        this.Hours = hours;
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { Hours };
    }
}

public class ContractedHours : ValueObject<ContractedHours>
{
    public readonly Hours Hours;

    public ContractedHours(Hours hours)
    {
        this.Hours = hours;
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { Hours };
    }
}

public class OvertimeHours : ValueObject<OvertimeHours>
{
    public readonly Hours Hours;

    public OvertimeHours(Hours hours)
    {
        this.Hours = hours;
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { Hours };
    }
}
```

As you can see from Listings 15-20 and 15-21, `HoursWorked` and `ContractedHours` add no additional behavior or state. It would be just as easy for `OvertimeCalculator.Calculate()` to accept two `Hours`

instances. But doing so emphasizes making the parameter names explicit and relying on callers to supply the two `Hours` objects in the correct order. Using micro types, you make the type system work harder for you so that it both increases the explicitness of your code and prevents human error.

LISTING 15-21: Unit Test Demonstrating Contextual Application of Micro Types

```
[TestClass]
public class Micro_types_example_tests
{
    [TestMethod]
    public void Calculates_overtime_hours_as_hours_additional_to_contracted()
    {
        var hoursWorked = new Hours(40);
        var contractedHours = new Hours(35);

        // wrap with micro types for contextual explicitness
        var hoursWorkedx = new HoursWorked(hoursWorked);
        var contractedHoursx = new ContractedHours(contractedHours);

        var fiveHours = new Hours(5);
        var fiveHoursOvertime = new OvertimeHours(fiveHours);

        var calc = new OvertimeCalculator();
        var result = calc.Calculate(hoursWorkedx, contractedHoursx);

        Assert.AreEqual(fiveHoursOvertime, result);
    }
}
```

Using micro types is far from an industry best practice. In fact, it's quite divisive. Some claim micro types are a precursor to clearer, more composable code, but for others, micro types are too many layers of annoying indirection. It's up to you to decide if you want to use the micro types pattern.

Collection Aversion

Some DDD practitioners feel that you should never have a collection of value objects. The rationale for this is that primitive collections do not properly express domain concepts. It is also argued that having a collection of value objects often means that you need to pick out specific items using some form of identity, which is clearly a violation of value objects having no identity. Again, though, this is not a universally applied practice within the community, but it is one that you should have a good justification for eschewing.

An example that highlights the lack of clarity through exposing a value object collection is the `Customer` entity presented in Listing 15-22, which has multiple `PhoneNumber` value objects.

LISTING 15-22: Entity with Collection of Value Objects

```
public class Customer
{
    public Customer(Guid id)
```

(*continued*)

LISTING 15-22 *(continued)*

```
        {
            this.Id = id;
        }

        public Guid Id { get; private set; }

        public IEnumerable<PhoneNumber> PhoneNumbers { get; set; }

        // ..
    }

    public class PhoneNumber : ValueObject<PhoneNumber>
    {
        public readonly string Number;

        public PhoneNumber(string number)
        {
            this.Number = number;
        }

        protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
        {
            return new object[] { Number };
        }

        // ..
    }
```

There are multiple `PhoneNumber`s that are part of the `PhoneNumbers` collection. Why? Could they be home and cell phone? Perhaps there's an emergency contact or work number in there, too. It's just not clear by using a collection. But in this domain, `PhoneNumber`s are structured according to business requirements. Each customer must supply a home, mobile, and work number.

A more honest way of modeling the domain is shown in Listing 15-23, which clearly uses structure to model the important domain concepts of a home, mobile, and work phone number. It also makes it easy to change any of these numbers without requiring an ID lookup or a nasty hack.

LISTING 15-23: Entity with an Expressive Value Object That Replaces a Collection

```
public class Customer
{
    public Customer(Guid id)
    {
        this.Id = id;
    }

    public Guid Id { get; private set; }

    public PhoneBook PhoneNumbers { get; set; }

    // ..
```

```
}

public class PhoneBook : ValueObject<PhoneBook>
{
    public readonly PhoneNumber HomeNumber;
    public readonly PhoneNumber MobileNumber;
    public readonly PhoneNumber WorkNumber;

    public PhoneBook(PhoneNumber homeNum, PhoneNumber mobileNum,
                     PhoneNumber workNum)
    {
        this.HomeNumber = homeNum;
        this.MobileNumber = mobileNum;
        this.WorkNumber = workNum;
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { HomeNumber, MobileNumber, WorkNumber };
    }
}

public class PhoneNumber : ValueObject<PhoneNumber>
{
    public readonly string Number;

    public PhoneNumber(string number)
    {
        this.Number = number;
    }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { Number };
    }

    // ..
}
```

In Listing 15-23, the modified `Customer` entity now has its `PhoneNumbers` represented by a `PhoneBook` value object. Each type of phone number—home, work, and mobile—can now be accessed without requiring any type of ID look-up. This is important because, as has been repeatedly mentioned, value objects do not have an identity.

Most importantly with Listing 15-23 compared to Listing 15-22 is that the intent and domain concepts are much clearer. `Customer` entities have a home, work, and mobile number. This important domain structure is encoded into types and enforced by the type system.

PERSISTENCE

Arguably, the trickiest aspect of dealing with value objects is persisting them. With document-oriented data stores like RavenDB and EventStore, this is a lesser problem; with these technologies, it's usually feasible to store the value object and the entity in the same document. With SQL databases,

however, there is a strong tradition of normalization, which leads to more variability concerning implementation. Therefore, the remainder of this chapter mostly focuses on SQL-based use cases.

> **NOTE** *You can test, modify, and inspect all the examples in this section by downloading this chapter's sample code.*

NoSQL

Many NoSQL databases use data denormalization, which highly contrasts with the strong convention of normalizing SQL databases. NoSQL can be beneficial to DDD because entire entities—sometimes entire aggregates—can be modeled as single documents. Problems associated with joining tables, normalizing data, and lazy loading via ORMs just don't exist with document-oriented modeling. In the context of value objects, this simply means that they are stored with the entity. For example, the `Customer` entity and `Name` value object shown in Listing 15-24 can be stored as the single, denormalized JavaScript Object Notation (JSON) document shown in Listing 15-25.

LISTING 15-24: Entity with Value Objects That Needs to Be Persisted

```
public class Customer
{
    // ..

    public Guid Id { get; protected set; }

    public Name Name { get; protected set; }

    // ..
}

public class Name : ValueObject<Name>
{
    // ..

    public string FirstName { get; protected set; }
    public string Surname { get; protected set; }
    // ..

}
```

LISTING 15-25: Denormalized Document-Oriented JSON Storage Format of Entity with Embedded Value Object

```
{
    "Customer": {
        "Id": ..,
        "Name": {
```

```
            "FirstName": ..,
            "Surname": ..
        },
        // ..
    }
}
```

A number of document-oriented NoSQL databases persist documents as JSON. Listing 15-25 shows how a `Customer` and its `Name` value object can be persisted in such a database as a single JSON document. Document databases like RavenDB usually apply this convention by default, meaning that you just have to create an appropriately structured object model.

Although embedding value objects in documents is a common convention, it is still an optional one. You can store value objects as separate documents for performance reasons, if necessary.

SQL

Persisting value objects in a SQL database comes with choices. You can follow standard SQL conventions and normalize your value objects in their own tables, or you can denormalize them in-situ akin to the document-oriented approach shown in Listing 15-25. This section shows an example of each of those two broad approaches. It's worth pointing out that these examples are just a guide. There are many variations possible with each approach, so you are free to customize and experiment based on your own needs.

Flat Denormalization

A common pattern for persisting value objects is to just store their value directly using a custom representation. This is a great choice when you don't want to have extra tables and extra statements in your queries to join them.

`DateTime` is a good template for persisting value objects directly because they are stored in a SQL database as a textual representation. There is no separate `DateTime` table that stores each value used in the application. In fact, `DateTime` is such a common value that it has its own database type. You can model your own value objects in this way, too, even without having an explicit database type. To achieve this, though, you need to create your own storage format, and you may need to teach your frameworks and ORMs how to use it.

Creating a Persistence Format

To be able to repopulate value objects when they are loaded from persistence, you are required to choose a format that uniquely identifies each possible value. A value object can then be persisted in this format when it is saved to the database and parsed from this format when it is loaded from the database.

Overriding `ToString()` is one possible pattern for generating a value object's persistence format. Because this format uniquely describes a value object, it can also be good for debugging. Listing 15-26 shows an updated `Name` value object that overrides `ToString()` to return a unique description of the value it represents, with the intention of persisting the object in that format. Listing 15-27 then shows some unit tests demonstrating how it works.

LISTING 15-26: Overriding ToString() to Return a Unique Description in a Persistence-Ready Format

```
public class Name : ValueObject<Name>
{
    public Name(string firstName, string surname)
    {
        this.FirstName = firstName;
        this.Surname = surname;
    }

    public string FirstName { get; protected set; }

    public string Surname { get; protected set; }

    protected override IEnumerable<object> GetAttributesToIncludeInEqualityCheck()
    {
        return new object[] { FirstName, Surname };
    }

    public override string ToString()
    {
        return String.Format(
            "firstName:{0};;surname:{1}", FirstName, Surname
        );
    }
}
```

LISTING 15-27: Example Value Object Representations

```
[TestMethod]
public void Each_value_has_a_unique_representation()
{
    var sallySmith = new Name("Sally", "Smith");
    Assert.AreEqual("firstName:Sally;;surname:Smith", sallySmith.ToString());

    var billyJean = new Name("Billy", "Jean");
    Assert.AreEqual("firstName:Billy;;surname:Jean", billyJean.ToString());
}
```

Creating a persistence format is chiefly about creating a unique representation for each value. This is the minimum requirement for this approach of persistence to work. In Listing 15-26, `ToString()` is overridden to produce a human-readable representation. In your applications, you can also favor human-readable, but sometimes you may want to prefer formats that have a smaller footprint or even another quality. When choosing a storage format, it's also worth keeping in mind that at some point you may need to filter SQL queries based on the values.

Another optional aspect of the implementation shown in Listing 15-26 is actually overriding `ToString()`. This was done for the dual benefit of persistence and debugging enhancements. But if you wanted a separate `ToString()` and persistence format, you could either overload `ToString()` or create another method, perhaps `ToPersistenceFormat()`. The most important detail is ensuring

that whichever method generates the value to be persisted is called at the time of persistence by your handcrafted SQL queries or your choice of ORM.

Persisting Values on Save

This example shows you how to persist a value object using an ORM (the most complex case). If you are persisting with handcrafted SQL queries, it should be easy to pass the custom representation of your value object into the `INSERT` or `UPDATE` query.

To persist your value object, the most important step is instructing your ORM. In NHibernate, you can do this by creating a customer `IUserType`, as Listing 15-28 shows. Most ORM frameworks are likely to have a similar feature.

LISTING 15-28: Manually Persisting Value Objects Using Framework Hooks

```
public class NameValueObjectPersister : IUserType
{

    // ..

    public void NullSafeSet(IDbCommand cmd, object value, int index)
    {
        // Value parameter is the Name instance to be persisted
        var parameter = (IDataParameter)cmd.Parameters[index];
        if (value == null)
        {
            parameter.Value = DBNull.Value;
        }
        else
        {
            // Name.ToString() produces persistence-format representation
            parameter.Value = value.ToString();
        }
    }

    // ....
}
```

NHibernate's `IUserType` allows developers to manually take control of persisting specific types of objects. In Listing 15-28, `NameValueObjectPersister` implements the `IUserType` interface to control how `Name` value objects are persisted. As you can see in Listing 15-28, `IUserType.NullSafeSet()` is where your custom logic is inserted to generate the value that is stored in the database.

> **NOTE** *NHibernate and some other ORMs have native support for value objects, saving you from having to devise your own custom format. For NHibernate, its Component concept is what you need, as demonstrated on the NHibernate blog (*`http://nhforge.org/blogs/nhibernate/archive/2008/09/17/value-objects.aspx`*). The examples in this section avoided this approach to demonstrate a more generic end-to-end solution that could be used with any technology. However, it's definitely worthwhile investigating the native support for value objects provided by your ORM because it can potentially save you a lot of time.*

> **NOTE** *To follow along with the NHibernate-based examples in the remainder of this chapter, you need to add NHibernate as a dependency to your project. The easiest way to do that is to add the following NuGet packages to your project: NHibernate and FluentNHibernate.*
>
> *For a beginner's guide to NHibernate, the official documentation is one of the best resources (*`http://nhforge.org/wikis/howtonh/your-first-nhibernate-based-application.aspx`*). Equally, the FluentNHibernate documentation is one of the best resources for FluentNHibernate (*`https://github.com/jagregory/fluent-nhibernate/wiki/Getting-started`*).*

Parsing Values on Load

Having told your ORM to persist your value object in a custom format, or having manually inserted the value yourself, the remaining task is to handle loading of the object by parsing the representation back into a value object. There are a few possible approaches to implementing this. First, you could have a constructor that takes a string representation of the value object. Alternatively, you could have an intermediate component that parses the string and constructs an instance in the normal way. This example uses the latter approach to avoid any persistence or framework-related concerns creeping into the value object.

As with persisting, the key detail is finding the framework hook that allows you to inject your custom parsing logic. With NHibernate, that hook is another method on the `IUserType` that was previously introduced. You can see how to parse a `Name` value object from its persistence representation in Listing 15-29.

LISTING 15-29: Manually Parsing Value Objects from Their Persistence Representation Using Framework Hooks

```
// NHibernate customization for custom mapping logic
public class NameValueObjectPersister : IUserType
{
    // ..

    public object NullSafeGet(IDataReader rs, string[] names, object owner)
    {
        object storageRepresenation = NHibernateUtil.String.NullSafeGet(
            rs, names[0]
        );
        if (storageRepresenation == null)
        {
            return null;
        }

        // storage representation format: firstName:{X};;surName:{Y}
        var parts = storageRepresenation.ToString().Split(
            new[] { ";;" }, StringSplitOptions.None
        );
```

```
            var firstName = parts[0].Split(':')[1];
            var surName = parts[1].Split(':')[1];

            return new Name(firstName, surName);
        }

        // ..
    }
```

`IUserType.NullSafeGet()` is the low-level NHibernate hook that allows you to manually reconstruct an object from its persisted state. In Listing 15-29, the implementation of `NullSafeGet()` extracts the `FirstName` and `Surname` from the plain-text representation of the `Name` value object that is being loaded and uses them to create a new `Name` instance. As mentioned, whichever ORM you are using, there is likely to be an equivalent hook that gives you low-level control for storing value objects in a custom format and parsing them back again.

You can see the full implementation of `NameValueObjectPersister` with this chapter's sample code, including unit tests that fully configure NHibernate before storing and loading sample data. Figure 15-1 shows an example of what is stored in the database after running one of the tests.

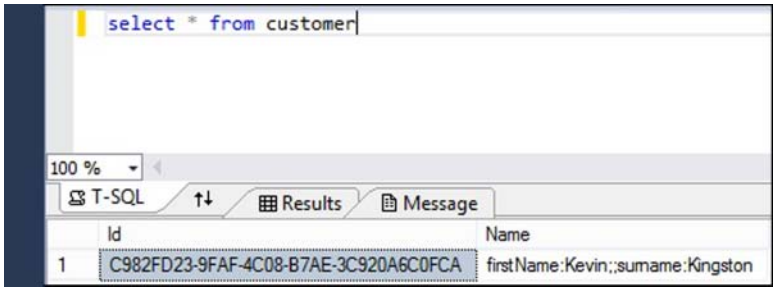


FIGURE 15-1: Value object stored in custom format.

Normalizing into Separate Tables

Denormalization is the de facto persistence strategy for value objects, but normalization is a strong SQL tradition. Sometimes company standards may even enforce the latter. Normalization can still be the best choice in some situations, though. Performance and efficiency reasons would be common examples—especially if the value object has a large representation that you do not want to be loaded every time the owning entity is loaded.

In this example, you see how to map an entity-value object relationship, where each type has its own table, and where a foreign key joins the two. As in the last example, NHibernate will be used to demonstrate this scenario, but the general pattern can likely be applied with whatever technology you are using.

For NHibernate to store the `Name` value object in a new table, the first adjustment is to add a protected zero-argument constructor, as Listing 15-30 illustrates.

LISTING 15-30: Modifying Value Object to Support Framework Conventions

```
public class Name : ValueObject<Name>
{
    protected Name()
    {
        // Required by NHibernate
    }

    // ..

}
```

Surprisingly, there is little additional work you need to do. When using Fluent NHibernate, you need only configure a `Join()` mapping relationship, as shown in Listing 15-31.

LISTING 15-31: Instructing ORM to Persist Value Object in a Separate Table

```
// Fluent NHibernate mapping class
public class CustomerNormalizedMap : ClassMap<Customer>
{
    public CustomerNormalizedMap()
    {
        Id(x => x.Id);

        // Create a separate table for the value object
        Join("CustomerName", join =>
        {
            join.KeyColumn("Id");
            join.Component(x => x.Name, c =>
            {
                c.Map(x => x.FirstName);
                c.Map(x => x.Surname);
            });
        });
    }
}
```

By using a `Join()` mapping instruction, the code in Listing 15-31 is telling Fluent NHibernate to create a separate `CustomerName` database table that stores `Name` value objects. The `KeyColumn()` instruction tells NHibernate to use the customer's ID as a foreign key in the `CustomerName` table. So when loading a `Name` occurs at run time, a `Customer` entity can find its `Name` by supplying its own ID. Whichever technology you are using, there is likely to be some convention, configuration, or hook that allows you to achieve something similar.

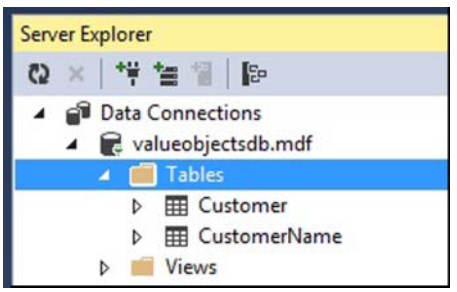


FIGURE 15-2: Separate table for Customer and Name.

Figure 15-2 illustrates the tables that NHibernate creates when you run the preceding configuration. Additionally,

you can see in Figure 15-3 how persisted data is stored—notably, the `Customer`'s ID is being used as the foreign key.

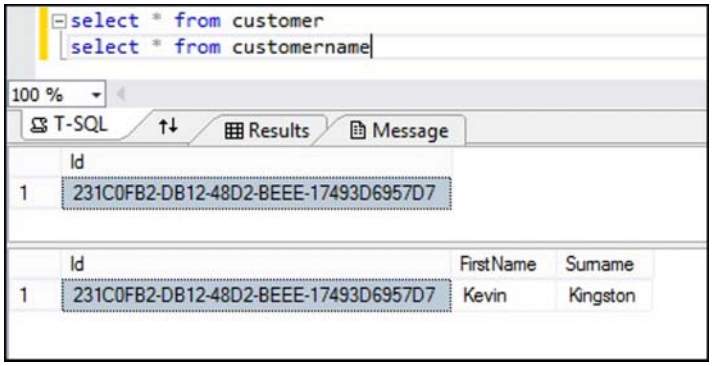


FIGURE 15-3: Customer foreign key used to join Customer and Name.

THE SALIENT POINTS

- Value objects are DDD modeling constructs that represent descriptive quantities like magnitudes and measurements.
- Due to having no identity, you do not have to lumber value objects with the complexities associated with entities.
- You are encouraged to wrap primitive types with integers and strings so that domain concepts are articulated well in the code.
- Examples of value objects include `Money`, `Currency`, `Name`, `Height`, and `Color`. However, it's important to remember that value objects in one domain might be entities in another and vice versa.
- Value objects are immutable; their values cannot change.
- Value objects are cohesive; they can wrap multiple attributes to fully encapsulate a single concept.
- Value objects can be combined to create new values without altering the original.
- Value objects are self-validating; they should never be in an invalid state.
- A number of modeling patterns exist for working with value objects, but you can also create your own.
- You can persist the value of value objects directly in a denormalized form. This is the most common and easiest case to implement with document-oriented databases and is still a popular approach with SQL.
- You can also persist value objects as their own documents or tables. This is still a common option for SQL databases, but it's often an optimization when applied to document databases.

16

Entities

WHAT'S IN THIS CHAPTER?

- An introduction to the DDD concept of entities
- Uncovering entities in a problem domain
- Differentiating entities and value objects
- Examples of entity implementation fundamentals
- Suggestions to help produce behavior-rich domain models by creating expressive entities
- Examples of optional design principles and patterns that can enhance the expressiveness and maintainability of entities

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 16 download and individually named according to the names throughout the chapter.

As you work with domain experts, they often refer to concepts that have an inherent identity in the problem domain. For instance, they may talk about a specific customer or a specific sporting event. These concepts are known as *entities*, and unlike value objects, it is usually not acceptable for two entities that have similar values to be considered equal.

Finding entities in your domain and modeling them explicitly is important for conceptual and technical reasons. If you understand that a concept is an entity, you can start to probe domain experts for related details, such as its life cycle. Technically, you also want to understand which concepts are entities because distinct trade-offs and considerations apply to their design and implementation, as you will see later in this chapter.

Understanding that entities are primarily concepts with a unique identity is unquestionably the main takeaway from this chapter. Although you will see implementation guidance and modeling patterns based on years of refinement in the Domain-Driven Design (DDD) community, it's important to be aware that many of these details can change over time with new insights and the emergence of new technologies. So as long as you understand the conceptual role of entities, and you are aware of general implementation considerations, you will always be in a good position to at least get your domain model in the right shape.

UNDERSTANDING ENTITIES

This section offers a theoretical look into what entities are and how they differ from other types of domain objects.

Domain Concepts with Identity and Continuity

A good opportunity to find entities in the domain you are modeling is to pay attention to how domain experts speak. As an example, imagine you are in discussions with a holiday and travel domain expert. The domain expert may explain to you that travelers choose a hotel they like and then book a holiday there. The domain expert also informs you that it is not acceptable for the traveler to be given a booking at a different hotel—even if it has the same name or has other values that are the same. In this situation, the domain expert is implicitly telling you that hotels are an entity, because their identity and uniqueness are important. Without them, it would not be possible to distinguish one hotel from other hotels, and travelers would not get the booking they expected.

Another piece of information to try to extract from domain experts is the type of identity an entity has. In many cases, identity can be an important domain concept in itself and can be explicitly modeled to improve the expressiveness of your domain model. In the case of a hotel, it might actually have a real-world ID that enables many disparate travel agents and applications to share data about it—like availability and user reviews. On the other hand, a hotel may have no real-world unique identifier, so an application would need to generate an arbitrary one. If it's not clear, then it's in your best interests to ask the domain expert(s) if an entity has a unique identity in the problem domain.

Aside from identity, but closely related, is another tell-tale sign that you have an entity: continuity. Domain experts may give you clues like, "The order is accepted," "The order is confirmed after payment," and "The order is fulfilled by the courier." These clues indicate that this "thing" has a life cycle in the domain. And usually, to have a life cycle in the domain, an identity is needed to allow the "thing" to be found and updated at various stages of its life cycle.

Uncovering entities can occur at any time; it's not necessarily the case that you sit down with domain experts and first identify all the entities in a system up front. Some DDD practitioners start by identifying the events that occur in the domain. And from there they collaborate with domain experts to understand which entities are involved with each event. In general, you should always be on the lookout.

Context-Dependent

Admittedly, it can be challenging to distinguish entities from other types of domain objects, like value objects. As mentioned previously, entities are fundamentally about identity—focusing on the "who" rather than the "what." But it can still be difficult to ascertain whether a concept is an entity or a value object, especially because entities and value objects are context dependent; a concept that is definitely an entity in one domain could unequivocally be a value object in another domain.

A common DDD example of context dependence is money. In a banking application, a customer might put $100 in her bank account. When she withdraws her $100 at some point in the future, she may receive different bank notes or coins than the ones she deposited. This difference is irrelevant, though, because the identity of the money is not important; the customer only cares about the value of the money. So in this domain, money is without a doubt a value object. But in another domain, perhaps involving the manufacture or traceability of money, the identity of individual notes or coins may actually be an important domain concept. So each piece of money would be an entity with a unique identifier.

IMPLEMENTING ENTITIES

Most entities have similar characteristics, so there are some fundamental design and implementation considerations that you should be aware of, including assigning identity, applying validation, and delegating behavior. In this section, you will see guidance and examples for each of the fundamental design considerations so that once you have uncovered entities in your domain, you can begin implementing them in your model.

Assigning Identifiers

Sometimes an entity's identity is determined by a natural domain identity, whereas other times there may be no preexisting natural identifier. With the former, it's your responsibility to collaborate with domain experts to reveal the natural identity, whereas for the latter, you need to generate an arbitrary identity in your application, potentially with help from your datastore.

Natural Keys

When trying to ascertain what an entity's identity should be, you should first consider whether the entity already has a unique identifier in the problem domain. These are called *natural keys*. Here are a few examples:

- Social security numbers (SSNs)
- Country name
- Payroll number
- National ID number
- ISBN (for books)

You do need to ensure that a natural key is never going to change. If it does, you may have lots of references in your system pointing to the old identity. At best, it's a hassle to update all of them. At

worst, you may forget to update some references or make mistakes that lead to significant business-level problems.

Once you have identified a natural key, and you're confident it definitely *is* a natural key, it's usually straightforward to assign it to an entity. Most of the time, you just need to add a constructor parameter, as shown in Listing 16-1.

LISTING 16-1: Entity Taking Natural Key ID as Constructor Parameter

```
public class Book
{
    public Book(ISBN isbn)
    {
        this.ISBN = isbn;
        this.Id = isbn.Number;
    }

    public string Id { get; private set; }

    public ISBN ISBN { get; private set; }
}
```

You can see in Listing 16-1 that the `Book` entity has an `ISBN` constructor parameter that is set as the identity of the newly created instance. You pass the natural key in, and the entity then assumes that identity.

One issue to be cognizant of when using natural keys is ensuring that your object-relational mapper (ORM) or data access technology is configured to allow them. Some frameworks may override the ID you have given an entity unless you explicitly tell the framework that you are managing the ID yourself. This issue also applies to arbitrarily generated keys, which are covered next.

Arbitrarily Generated IDs

When there is no unique identifier in the problem domain, you need to decide what kind of ID you are going to use and how you will generate it. Common formats include incremental numbers, globally unique identifiers (GUIDs) (aka universally unique identifiers, UUIDs), and strings.

Incremental Numeric Counters

Numbers usually have the smallest footprint, but they inherently pose the challenge of maintaining a global counter of the last assigned ID. Conversely, GUIDs are good because they don't have this problem. Instead, you just generate a GUID which is automatically guaranteed to be unique. However, it does then take up more storage space when it's persisted. The additional storage is relatively insignificant, though, so for many applications GUIDs are the default approach. Strings tend to be used for custom ID formats such as hashes, amalgamations of multiple attributes, or even timestamp-based approaches.

Listing 16-2 demonstrates using incremental numbers, whereby a static variable keeps track of the last assigned ID and a factory method ensures that each new entity gets the next sequential number for its ID.

LISTING 16-2: Maintaining a Global Counter

```
public static class RandomEntityFactory
{
    private static long lastId = 0;

    public static RandomEntity CreateEntity()
    {
        return new RandomEntity(++lastId);
    }
}

public class RandomEntity
{
    public RandomEntity(long Id)
    {
        this.Id = Id;
    }

    public long Id { get; private set; }
}
```

If an application crashes, then a static variable like `lastId` shown in Listing 16-2 loses its value, meaning that old IDs are likely to be reused when the application restarts. To remedy this, you likely need to persist the counter. But doing so is suboptimal for many use cases because of the performance overhead of reading and updating combined with the additional complexity. Some implementations even require fragile locking techniques in distributed or load-balanced environments. It comes down to your best judgment on a case-by-case basis, but if generating IDs is a complex and fragile part of your system, then perhaps GUIDs are an easy way to make life much easier for your team.

GUIDs/UUIDs

Using GUIDs can be a massive simplification over maintaining a global counter. In many cases, the performance, complexity, and synchronization problems completely go away, as Listing 16-3 shows.

LISTING 16-3: GUID Creation Simplifies ID Management

```
public static class VehicleFactory
{
    public static Vehicle CreateVehicle()
    {
        var id = Guid.NewGuid();
        return new Vehicle(id);
    }
}

public class Vehicle
{
    public Vehicle(Guid id)
    {
        this.Id = id;
```

continues

LISTING 16-3 *(continued)*

```
        }

        public Guid Id { get; private set; }

        ...
    }
```

GUIDs are guaranteed to be unique, so no matter where the call to `GUID.NewGuid()`, it always produces a unique identifier. Therefore, `VehicleFactory.CreateVehicle()` always creates a `Vehicle` with a unique ID—even if the same method was invoked at the exact time on multiple servers in a load-balanced environment. This demonstrates why you should often favor GUIDs if you're generating your own IDs.

Using GUIDs can be especially useful when you have logic in the browser that needs to create an entity and needs to post back to multiple back-end application programming interfaces (APIs). Without an ID, there's no way for the back-end services to know you are posting information about the same entity to each of them. You can solve this problem by creating a GUID on the client using JavaScript. Figure 16-1 illustrates this process.

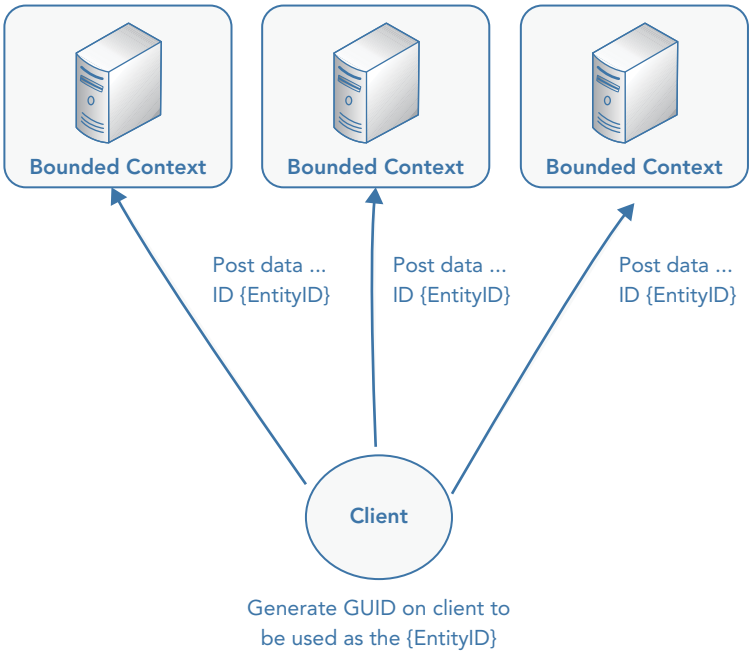


FIGURE 16-1: Creating a client-side GUID and posting to multiple back-end services.

There was a long discussion on the "DDD/CQRS" (Command Query Responsibility Segregation) mailing list about client-side ID generation. It's definitely worth a read if you are considering this approach or want to learn more about it (`https://groups.google.com/forum/#!msg/dddcqrs/xYfmh2WwHKk/XW7eauXcKkcJ`).

Strings

Using strings is an opportunity for you to create your own custom ID format. There are many possible strategies that you can use to create formats. One example is including some of the entity's state in the ID for diagnostic benefits. Listing 16-4 shows a simplified example of this, in which a `HolidayBooking` entity's ID is a string containing the ID of the traveler who booked it, the start and end dates of the booking, and a timestamp of when the booking was confirmed.

LISTING 16-4: Using a Custom ID Format Based on Entity Values and Timestamps

```
public class HolidayBooking
{
    public HolidayBooking(int travelerId, DateTime firstNight,
                          DateTime lastNight, DateTime booked)
    {
        this.TravelerId = travelerId;
        this.FirstNight = firstNight;
        this.LastNight = lastNight;
        this.Booked = booked;
        this.Id = GenerateId(travelerId, firstNight, lastNight, booked);
    }

    public string Id { get; private set; }

    public int TravelerId { get; private set; }

    public DateTime FirstNight { get; private set; }

    public DateTime LastNight { get; private set; }

    public DateTime Booked { get; private set; }

    private string GenerateId(int travelerId, DateTime firstNight,
                              DateTime lastNight, DateTime booked)
    {
        return string.Format(
            "{0}-{1}-{2}-{3}",
            travelerId, ToIdFormat(firstNight),
            ToIdFormat(lastNight), ToIdFormat(booked)
         );
    }

    private string ToIdFormat(DateTime date)
    {
        return date.ToString("yyyy/MM/dd");
    }

    ...
}
```

`GenerateId()` in Listing 16-4 is called in the constructor of the `HotelBooking` entity to generate a unique ID using a custom format. This is a trivial example, but it is still similar to implementations used in real applications.

Similar to natural keys, if you want to include an entity's values in its ID for diagnostic purposes, the values probably should not change. So it is worth being extra careful and having a good understanding of domain characteristics when creating a custom ID format.

Datastore-Generated IDs

It's often easy enough and safe enough to delegate ID generation to your datastore. Most databases, ranging from SQL databases like MS SQL Server to document databases like RavenDB, natively support ID generation.

Creating datastore-generated IDs often follows the similar pattern of passing your newly created entity into your chosen data access library. Upon successful completion of the next transaction, your entity then has its ID set. Listing 16-5 contains a test case demonstrating this behavior using NHibernate backed by SQL Server.

LISTING 16-5: Delegating ID Generation to a Datastore via an ORM

```
[TestClass]
public class DatastoreIdGenerationExample
{
    // see this chapter's sample code for implementation of CreateSession()
    ISession session = CreateSession();

    [TestMethod]
    public void Id_is_set_by_datastore_via_ORM()
    {
        var entity1 = new IdTestEntity();
        var entity2 = new IdTestEntity();

        // initially no ID
        Assert.AreEqual(0, entity1.Id);
        Assert.AreEqual(0, entity2.Id);

        NHibernateTransaction(session =>
        {
            session.Save(entity1);
            session.Save(entity2);
        });

        // ID will have been set via NHibernate
        Assert.AreEqual(1, entity1.Id);
        Assert.AreEqual(2, entity2.Id);
    }

    private void NHibernateTransaction(Action<ISession> action)
    {
        using (var transaction = session.BeginTransaction())
        {
            action(session);
            transaction.Commit();
        };
```

```
        }

        ...
    }
```

In Listing 16-5, two `IdTestEntity` instances are created without an ID. You can see that the initial two assertions verify this. Then, inside a NHibernate transaction, both entities are saved (which equates to being persisted once the NHibernate transaction is committed). The final two assertions verify that the IDs were indeed set as expected. You can see all the code, including the NHibernate configuration and setup, in this chapter's sample code.

> **NOTE** *The example in Listing 16-5 uses NHibernate version 4.0.0.4000 and Fluent NHibernate version 1.4.0.0. Newer or older versions may have different APIs.*

Pushing Behavior into Value Objects and Domain Services

Keeping entities focused on the responsibility of identity is important because it prevents them from becoming bloated—an easy trap to fall into when they pull together many related behaviors. Achieving this focus requires delegating related behavior to value objects and domain services. You actually saw examples of this in the previous chapter when value objects were combinable, comparable, and self-validating (thus keeping logic out of the entities that use them). Equally, you'll see in Chapter 17, "Domain Services," that there are often cases where stateless domain operations that at first appear to belong to entities can actually be encapsulated as domain services.

To demonstrate the benefits of pushing behavior from entities into value objects, Listing 16-6 shows an updated version of the `HolidayBooking` entity previously introduced in Listing 16-4. This version implements key domain policies using the Ubiquitous Language (UL). First, it ensures that the first night of a holiday precedes the last night. Second, it ensures that the booking meets the minimum duration of three nights. It offloads both of these requirements to the `Stay` value object.

LISTING 16-6: Entity Pushing Behavior into a Value Object

```
public class HolidayBooking
{
    public HolidayBooking(int travelerId, Stay stay, DateTime booked)
    {
        this.TravelerId = travelerId;
        this.Stay = stay;
        this.Booked = booked;
        this.Id = GenerateId(
            travelerId, stay.FirstNight, stay.LastNight, booked
        );
```

continues

LISTING 16-6 *(continued)*

```
    }

    public string Id { get; private set; }

    public int TravelerId { get; private set; }

    public Stay Stay { get; private set; }

    public DateTime Booked { get; private set; }

    private string GenerateId(int travelerId, DateTime firstNight,
                              DateTime lastNight, DateTime booked)
    {
        return string.Format(
            "{0}-{1}-{2}-{3}",
            travelerId, ToIdFormat(firstNight), ToIdFormat(lastNight),
            ToIdFormat(booked)
        );
    }

    private string ToIdFormat(DateTime date)
    {
        return date.ToString("yyyy/MM/dd");
    }

}

public class Stay
{
    public Stay(DateTime firstNight, DateTime lastNight)
    {
        if (firstNight > lastNight)
            throw new FirstNightOfStayCannotBeAfterLastNight();

        if (DoesNotMeetMinimumStayDuration(firstNight, LastNight))
            throw new StayDoesNotMeetMinimumDuration();

        this.FirstNight = firstNight;
        this.LastNight = lastNight;
    }

    public DateTime FirstNight { get; private set; }

    public DateTime LastNight { get; private set; }

    private bool DoesNotMeetMinimumStayDuration(DateTime firstNight,
                                                DateTime lastNight)
    {
        return (lastNight - firstNight) < TimeSpan.FromDays(3);
    }
}
```

Listing 16-4 demonstrated the initial `HolidayBooking` entity that was focused solely on identity. It would have easily been possible to implement the new behavior directly inside that class, instead of in the `Stay` value object shown in Listing 16-6. However, putting this logic directly inside the entity would reduce expressiveness by mixing in logic related to identity with logic related to stays (the period of time for which the holiday booking applies to).

You might think that `Stay` is quite a small class and it would be fine for the logic to go directly inside the `HolidayBooking`. But imagine other aspects of a booking, such as extras like transfers or flights. If you put all of those responsibilities inside the `HolidayBooking` entity it would obscure and intermingle domain concepts. Any time you add behavior to an entity, it's worth your while to consider whether you can push it into a value object for enhanced domain clarity.

> **NOTE** *Generally you should avoid bloating any classes with more than one responsibility because they are harder to maintain, harder to test, and become a part of the code base you'll hate working with. Adhering to the Single Responsibility Principle (SRP) is excellent advice for general software development and especially for entities. For example, when you push behavior into a value object, it is easier to change and test just that single responsibility without affecting other responsibilities.*

One aspect of your entities to be conscious of when pushing behavior into value objects is the depth of the object graph. In Listing 16-6, the `Stay` property is public and so are its `FirstNight` and `LastNight` properties. That makes it possible for consumers of the entity to call `HotelBooking.Stay.FirstNight`. Three levels deep is reasonable in this case, but you should be careful about how much of your object graph to expose, because clients will couple themselves to it. In this scenario, if you wanted to refactor the `FirstNight` and `LastNight` properties, it may be difficult because other parts of the system are tightly-coupled to them. On a case-by-case basis you'll have to trade-off the depth at which your object graph is exposed against how far up the object graph you bring public behavior.

> **NOTE** *The discussion around whether to allow clients of the* `Stay` *value object access to its* `FirstNight` *property relates closely to the object-oriented programming (OOP) principle The Law of Demeter (LoD). For a balanced discussion on the LoD, Phil Haack, formerly of Microsoft, has published a pragmatic blog post that echoes the thoughts of many (*`http://haacked.com/archive/2009/07/14/law-of-demeter-dot-counting.aspx/`*).*

Validating and Enforcing Invariants

In addition to identity, a primary implementation requirement for entities is ensuring they are self-validating and always valid. This is similar to the self-validating nature of value objects, although it's usually much more context dependent due to entities having a life cycle. As an example, a `FlightBooking` entity may be allowed to have its `DepartureDate` modified while it is awaiting

confirmation by the airline. Once confirmed, however, the validation rules then preclude changes to the `DepartureDate` in line with business policy. Listing 16-7 shows an implementation of the `FlightBooking` entity demonstrating these context-dependent validation rules.

LISTING 16-7: Context-Dependent and Constructor Entity Validation

```
public class FlightBooking
{
    private bool confirmed = false;

    public FlightBooking(Guid id, DateTime departureDate, Guid customerId)
    {
        if (id == null)
            throw new IdMissing();

        if (departureDate == null)
            throw new DepartureDateMissing();

        if (customerId == null)
            throw new CustomerIdMissing();

        this.Id = id;
        this.DepartureDate = departureDate;
        this.CustomerId = customerId;
    }

    public Guid Id { get; private set; }

    public DateTime DepartureDate { get; private set; }

    public Guid CustomerId { get; private set; }

    public void Reschedule(DateTime newDeparture)
    {
        if (confirmed) throw new RescheduleRejected();

        this.DepartureDate = newDeparture;
    }

    public void Confirm()
    {
        this.confirmed = true;
    }
}
```

When `Reschedule()` is invoked with an updated departure date, the `FlightBooking` entity shown in Listing 16-7 throws a `RescheduleRejected` exception. But it doesn't do that if the booking has not yet been confirmed. So this validation is context dependent because it only applies in certain scenarios. You may feel that this contradicts the statement that "entities are always valid." However, "always valid" really means "always contextually valid" in the case of entities.

You can also see in Listing 16-7 that each constructor argument is validated. This is another important guideline for ensuring that entities are always valid. It will also help to prevent problems

spreading through your domain and putting your domain into inconsistent states. This example uses specific exception types for implementing constructor validation, but it's up to you how you want to implement validation. In general it's best to be expressive.

> **WARNING** *Validating constructor arguments is omitted from some of the code samples in this chapter to aid the clarity of examples. However, in a real application, you should have a very good reason for not validating constructor parameters due to the damage that can be caused.*

Entities are also responsible for enforcing a more fundamental form of validation: invariants. Invariants are facts about an entity. They mandate that the values of certain attributes must fall within a certain range to be an accurate representation of the entity being modeled.

In the context of hotels, to find invariants, you might ask yourself, "What makes a hotel a hotel?" or "What does it mean for a hotel to be a hotel?" In a travel and holidays domain, a hotel represents a building with rooms that travelers can book as the base for their holiday. If a building doesn't have rooms, there's no way it can be a hotel. Listing 16-8 shows how the invariant of "being a hotel necessities having rooms" can be enforced.

LISTING 16-8: Enforcing Invariants of an Entity

```
public class Hotel
{
    public Hotel(Guid id, HotelAvailability initialAvailability,
                 HotelRoomSummary rooms)
    {
        ...

        EnforceInvariants(rooms);
        this.Id = id;
        this.Availability = initialAvailability;
        this.Rooms = rooms;
    }

    private void EnforceInvariants(HotelRoomSummary rooms)
    {
        if (rooms.NumberOfSingleRooms < 1 &&
            rooms.NumberOfDoubleRooms < 1 &&
            rooms.NumberOfFamilyRooms < 1)
            throw new HotelsMustHaveRooms();
    }

    public Guid Id { get; private set; }

    public HotelAvailability Availability { get; private set; }

    public HotelRoomSummary Rooms { get; private set; }

    ...
}
```

Enforcing invariants doesn't have to be complicated, although ideally it should be explicit. This is exemplified in Listing 16-8 with `EnforceInvariants()`. Before construction of the `Hotel` begins, this method is fired and validates that the constructor arguments satisfy the fundamental requirements of a hotel. In this case, that means ensuring that the `Hotel` has at least one room. Upon failure of this condition, an explicit `HotelsMustHaveRooms` exception is thrown, leaving no doubts about the invariant and no way to bypass it.

As mentioned, validation and invariants are similar in nature and appearance. You can see evidence of this by comparing Listing 16-7 and Listing 16-8. However, the subtle, but significant, difference is that the invariant in Listing 16-8 always applies regardless of context. Subtle differences like this are important, because when you are learning about domains, you need to be able to distinguish contextual validation rules from invariants to accurately understand and model your problem domain.

You will see later in this chapter that it is possible to push validation behavior out of entities using specifications. You will also see that when any entity has a number of states, you might want to consider modeling them explicitly if you are tempted to use the state pattern.

Focusing on Behavior, Not Data

A common opinion that many DDD practitioners share is that entities should be behavior oriented. This means that an entity's interface should expose expressive methods that communicate domain behaviors instead of exposing state. More generally, this is closely related to the OOP principle of "Tell Don't Ask."

Focusing on an entity's behavior when using DDD is important because it makes your domain model more expressive. Also, by encapsulating an entity's state, that state can only be operated on by the instance that encapsulates it. This means that any behavior that needs to modify the state has to belong to the entity. This is desirable because it prevents you from putting logic that belongs to an entity in the wrong place.

To implement behavior-focused entities you need to be wary of exposing getters and extremely sensitive to exposing setters (making them public). Exposing setters means there is an increased risk that an entity's state will be updated by other parts of the system in a way that offers no explanation of why it is being updated—hiding the domain concept or the reason for the update, as the next example demonstrates.

If you were modeling a problem domain that involved tracking the results of soccer matches, you might be tempted to implement a `SoccerCupMatch` entity similar to Listing 16-9, in which all the entity's state is exposed. However, you should think carefully about implementing a state-oriented design like this.

LISTING 16-9: An Entity That Exposes get and set Access to Its State

```
public class SoccerCupMatch
{
    public SoccerCupMatch(Guid id, Scores team1Scores, Scores team2Scores)
    {
        if (id == null)
```

```
            throw new ArgumentNullException(
                "Soccer cup match ID cannot be null"
            );

        if (team1Scores == null)
            throw new ArgumentNullException(
                "Team 1 scores cannot be null"
            );

        if (team2Scores == null)
            throw new ArgumentNullException(
                "Team 2 scores cannot be null"
            );

        this.ID = id;
        this.Team1Scores = team1Scores;
        this.Team2Scores = team2Scores;
    }

    public Guid ID { get; private set; }

    public Scores Team1Scores { get; set; }

    public Scores Team2Scores { get; set; }
}
```

If you expose an entity's state for public access and modification, as the `SoccerCupMatch` in Listing 16-9 does, you leave open the possibility that behavior belonging to the entity can be located elsewhere in the domain that is less explicit. For instance, soccer has an interesting rule called the away goals rule. Two teams play each other twice in a cup match, once at the home stadium for each team. If the overall score is a draw then the team that scored the most goals at their opponent's stadium (away goals) is the winner. For example, if the first match was 1–0 to team A at team A's stadium, and the second match was 2–1 to team B at team B's stadium, the overall score would be 2–2. However, team A scored 1 away goal and team B scored 0 away goals. So team A would be the winner.

If you look at the revised `SoccerCupMatch` entity in Listing 16-9, you can see that it only exposes its state—the score. Clients of this entity may see the score as 2–2 and erroneously behave like the match was a draw or that team B won. Potentially, there could be multiple clients of the entity that all implement logic for checking the winner of a match in different ways. Such inconsistencies would lead to strange and incorrect behavior in the domain model. Instead, the `SoccerCupMatch` entity should encapsulate its state and expose the behavior for calculating a winner as the updated version in Listing 16-10 does.

LISTING 16-10: A Behavior-Rich Entity That Encapsulates Its State

```
public class SoccerCupMatch
{
    public SoccerCupMatch(Guid id, Scores team1Scores, Scores team2Scores)
    {
        ...
```

continues

LISTING 16-10 *(continued)*

```
    }

    public Guid ID { get; private set; }

    private Scores Team1Scores { get; set; }

    private Scores Team2Scores { get; set; }

    public Scores WinningTeamScores
    {
        get
        {
            if (Team1Scores.TotalScore > Team2Scores.TotalScore)
                return Team1Scores;

            if (Team2Scores.TotalScore > Team1Scores.TotalScore)
                return Team2Scores;

            var awayGoalsWinner = FindWinnerUsingAwayGoalsRule();
            if (awayGoalsWinner == null)
                return FindWinnerOfPenaltyShootout();
            else
                return awayGoalsWinner;
        }
    }

    private Scores FindWinnerUsingAwayGoalsRule()
    {
        if (Team1Scores.AwayLegGoals > Team2Scores.AwayLegGoals)
            return Team1Scores;

        if (Team2Scores.AwayLegGoals > Team1Scores.AwayLegGoals)
            return Team2Scores;

        // The scores were exactly the same so no away goals winners
        return null;
    }

    private Scores FindWinnerOfPenaltyShootout()
    {
        if (Team1Scores.ShootoutScore > Team2Scores.ShootoutScore)
            return Team1Scores;

        if (Team2Scores.ShootoutScore > Team1Scores.ShootoutScore)
            return Team2Scores;

        throw new ThereWasNoPenaltyShootout();
    }
}
```

Even if you didn't know what soccer was, by looking at the `SoccerCupMatch` implementation in Listing 16-10, you would be able to infer the meaning of the away goals rule. Also as important,

by making the state private and only exposing behavior, clients of `SoccerCupMatch` can only communicate with instances of `SoccerCupMatch` using domain terminology as opposed to just mutating state. This is a big step toward creating behavior-rich, expressive domain models.

Even with the best of intentions, sometimes you are faced with big compromises in which you appear to have no choice to expose getters and setters. Later in this chapter, you see how the memento pattern can alleviate the need to expose getters and setters.

> **NOTE** *For a full explanation of the away goals rule, you can consult Wikipedia:* `http://en.wikipedia.org/wiki/Away_goals_rule`.

Avoiding the "Model the Real-World" Fallacy

Even though capturing precise problem domain behavior is a fundamental practice of DDD, an unfortunate consequence is that people new to DDD tend to model too much behavior. They innocently buy into the fallacy that DDD is about modeling the real world. Subsequently, they try to model many real-world behaviors of an entity that are not actually relevant to the software application being built. This is a fairly common occurrence, and a completely understandable one that arises out of a team's best intentions. Unfortunately, it is problematic because it increases the number of concepts and complexity in a domain model. It also leads to confusion if there are parts of a codebase that appear to never be used.

Once you understand that your domain model is not intended to be a full-scale model of the actual problem domain, you probably won't make this mistake, or if you do, you will quickly realize and correct yourself. But there is a similar problem that's sometimes harder to detect; sometimes entities are shaped based on UI requirements. Often, the best solution is to add any translations or UI-related logic onto a view model or data transfer object (DTO) to avoid polluting an entity's interface and responsibilities.

> **NOTE** *Chapter 25, "Commands: Application Service Patterns for Processing Business Use Cases," also contains guidance for enabling encapsulated state to be passed up through the service layer to UIs. In particular, the visitor pattern demonstrations are highly relevant. The memento pattern shown later in this chapter is also highly relevant.*

It's a familiar sight for technical concepts to creep into an entity. Common examples include dependency injection attributes, validation attributes, and ORM functionality like lazy loading. Again, to many, these are considered pollutants that decrease the expressiveness of a domain model. It's often best to avoid them as much as possible. Admittedly, sometimes you really will find yourself in a tough situation in which you have to pragmatically balance out the desire for a clean domain model with the need to get things done in the simplest way possible. In such situations, you may have to grit your teeth and accept the pain of polluting your domain model, but don't give in without a fight.

Designing for Distribution

In recent years distributed systems have become the new normal. Subsequently, new design choices have arisen for domain models and, in particular, entities. Overwhelmingly, however, most DDD practitioners strongly suggest not distributing entities. In essence, this means that an entity should be confined to a single class in a single domain model inside a single bounded context. This guidance makes more sense if you consider the typical `Customer` entity shown in Listing 16-11.

LISTING 16-11: An Entity That Logically Spans Multiple Bounded Contexts

```
public class Customer
{
    public Customer(Guid id, AddressBook addresses, Orders orderHistory,
        PaymentDetails paymentDetails, LoyaltySummary loyalty)
    {
        ...
    }

    public Guid Id { get; private set;}

    public AddressBook Addresses { get; private set; }

    public Orders OrderHistory { get; private set; }

    public PaymentDetails PaymentDetails { get; private set; }

    public LoyaltySummary Loyalty { get; private set; }
}
```

Addresses, personal details, order history, payment details, and loyalty are all present on the `Customer` entity in Listing 16-11. In a monolithic application, there may be some reason to justify this design. But in a distributed system, addresses, orders, payment details, and loyalty information may reside in different bounded contexts. So to load this entity from persistence, queries against multiple databases, in different bounded contexts, might be necessary. As you saw in Chapters 11 through 13, tight coupling between distributed components is highly susceptible to scalability, resiliency, and other problems arising from the fact that there is a network involved.

With distribution in mind, the `Customer` entity in Listing 16-11 actually represents a number of different concepts that live in different bounded contexts. Listing 16-12 illustrates how the loyalty, orders, and payments aspects of the `Customer` entity can be remodeled with distribution and bounded context partitioning in mind.

LISTING 16-12: Distribution/Bounded Context-Aware Entities

```
namespace MarketingBoundedContext
{
    public class Loyalty
    {
```

```
            ...

            public Guid CustomerId { get; private set; }

            public LoyaltySummary Loyalty { get; private set; }

            ...
        }
    }

    namespace AccountsBoundedContext
    {
        public class OrderHistory
        {
            ...

            public Guid CustomerId { get; private set; }

            public Orders Orders { get; private set; }

            ...
        }
    }

    namespace BillingBoundedContext
    {
        public class PaymentDetails
        {
            ...

            public Guid CustomerId { get; private set; }

            public CardDetails Default { get; private set; }

            public CardDetails Alternate { get; private set; }

            ...
        }
    }
```

Each of the entities in Listing 16-12 contains a chunk of functionality taken from the bloated `Customer` entity previously introduced in Listing 16-11. All these entities have a `CustomerId` that enables the information from them to be combined, even though they reside in different bounded contexts. This solution is likely to prove significantly less problematic in a distributed system. And, as a bonus, the inclusion of bounded contexts better aligns with the problem domain.

Even before DDD was being applied to distributed systems, the notion of decomposing systems into bounded contexts and apportioning responsibilities, like those of the `Customer` entity in Listing 16-11, has always been a strong recommendation. Therefore, the guidance presented in this section around distribution actually applies to any DDD implementation that has, or can benefit from, multiple bounded contexts.

COMMON ENTITY MODELING PRINCIPLES AND PATTERNS

In this section you are introduced to a few principles and patterns that can improve the expressiveness and maintainability of your entities. These aren't the only patterns, though. In fact, the aim of this section is to show you that there is room for creativity when implementing entities, assuming that the fundamentals in the previous section have been considered first.

Implementing Validation and Invariants with Specifications

Specifications are small, single-purpose classes, similar to policies. The benefits of using specifications for validation and invariants include enhanced expressiveness through encapsulating a single concept in its own class and the increased testability of immutable, side-effect-free logic. Using specifications is also an example of pushing behavior out of entities and into other types of objects, as recommended earlier in the chapter.

Listing 16-13 contains an extract of an updated version of the `FlightBooking` entity from earlier in the chapter. This version uses a specification to help apply its rescheduling business rule.

LISTING 16-13: Using Basic Specifications for Validation

```
public class FlightBooking
{
    private NoDepartureReschedulingAfterBookingConfirmation spec =
          new NoDepartureReschedulingAfterBookingConfirmation();

    ...

    public void Reschedule(DateTime newDeparture)
    {
        if (!spec.IsSatisfiedBy(this)) throw new RescheduleRejected();

        this.DepartureDate = newDeparture;
    }

    ...
}
```

Basically, the logic for assessing whether the new departure date can be accepted has been offloaded into a separate class called `NoDepartureReschedulingAfterBookingConfirmation`, which perfectly describes the business rule. This class is shown in Listing 16-14 along with the generic `ISpecification` interface it implements.

LISTING 16-14: Specification Implementation and Generic Interface

```
public interface ISpecification<T>
{
    bool IsSatisfiedBy(T Entity);
}

public class NoDepartureReschedulingAfterBookingConfirmation :
```

```
    ISpecification<FlightBooking>
{
    public bool IsSatisfiedBy(FlightBooking booking)
    {
        return !booking.Confirmed;
    }
}
```

A specification using this variation of the pattern has a single method, `IsSatisfiedBy()`, as shown in Listing 16-14. `IsSatisfiedBy()` should return `false` if the business rule being modeled cannot be applied to the passed-in object.

You may be wondering if the overhead of creating separate specifications and using a generic interface is actually worth it. It's certainly debatable—both having separate specifications and bothering to have the interface. There's another key aspect of this pattern, though, that benefits from the `ISpecification` abstraction and might convince you of its usefulness: the composition of specifications, as demonstrated in Listing 16-15.

LISTING 16-15: Composing Specifications

```
new OrSpecification<FlightBooking>(
    new FrequentFlyersCanRescheduleAfterBookingConfirmation(),
    new NoDepartureReschedulingAfterBookingConfirmation()
);
```

Some airlines provide frequent flyers with tasty little perks such as short-notice rescheduling. If you wanted to update the "no rescheduling after confirmation policy" to not apply to select customers, you could model it using an `OrSpecification`, as shown in Listing 16-14. An `OrSpecification` first tries one specification, and if that fails it tries the other one. With this in mind, Listing 16-15 should now make sense; the `OrSpecification` first tries the `FrequentFlyersCanRescheduleAfterBookingConfirmation` specification, and only if that fails does it revert to the `NoDepartureReschedulingAfterBookingConfirmation` specification, in line with the business rules of this domain.

You can see the implementation of `OrSpecification` in Listing 16-16.

LISTING 16-16: Generic Specifications for Composing Other Specifications

```
public class OrSpecification<T> : ISpecification<T>
{
    private ISpecification<T> first;
    private ISpecification<T> second;

    public OrSpecification(ISpecification<T> first, ISpecification<T> second)
    {
        this.first = first;
        this.second = second;
    }

    public bool IsSatisfiedBy(T entity)
```

continues

LISTING 16-16 *(continued)*

```
    {
        return first.IsSatisfiedBy(entity) || second.IsSatisfiedBy(entity);
     }
}
```

Building on the foundation of `ISpecification`, you can also compose specifications in other ways. For example, you can create an `AndSpecification` that requires both specifications to evaluate to `true`. In addition, you can create a base class or apply the builder pattern for fluently composing specifications, as suggested in Listing 16-17.

LISTING 16-17: Fluent Composition of Specifications

```
var ffSpec = new FrequentFlyersCanRescheduleAfterBookingConfirmation();
var ndSpec = new NoDepartureReschedulingAfterBookingConfirmation();

var spec = ffSpec.Or(ndSpec).Or( ... );
```

Avoid the State Pattern; Use Explicit Modeling

Many domains have entities that naturally exhibit different life cycle stages or states. In each state, usually only a subset of the entity's behavior is applicable. For example, an online takeaway order may have the states "in kitchen queue," "being prepared," "being cooked," and "out for delivery." Obviously, when an order is out for delivery, it cannot be put in the oven, or when the order is in the oven, it cannot start being prepared because that has already happened.

Because of its life cycle, there is sometimes a temptation to model entities using the state pattern. However, some DDD practitioners strongly discourage liberal use of the state pattern for entities. Instead, the alternative is to just have explicit classes for each state.

To implement the online takeaway order scenario just described as an entity using the state pattern, Listing 16-18 shows the `OnlineTakeawayOrder` entity along with the interface for each state and an implementation of one example state.

LISTING 16-18: Implementing an Entity Using the State Pattern

```
public class OnlineTakeawayOrder
{
    // state that behavior is delegated to
    private IOnlineTakeawayOrderState state;

    public OnlineTakeawayOrder(Guid id, Address address)
    {
        ...

        this.Id = id;
```

```
        this.Address = address;
        this.state = new InKitchenQueue(this);
    }

    public Guid Id { get; private set; }

    public Address Address { get; private set; }

    public void Cook()
    {
        state = state.Cook();
    }

    public void TakeOutOfOven()
    {
        state = state.TakeOutOfOven();
    }

    public void Package()
    {
        state = state.Package();
    }

    public void Deliver()
    {
        state = state.Deliver();
    }
}

// each state implements this
public interface IOnlineTakeawayOrderState
{
    IOnlineTakeawayOrderState Cook();

    IOnlineTakeawayOrderState TakeOutOfOven();

    IOnlineTakeawayOrderState Package();

    IOnlineTakeawayOrderState Deliver();
}

public class InKitchenQueue : IOnlineTakeawayOrderState
{
    private OnlineTakeawayOrder order;

    public InKitchenQueue(OnlineTakeawayOrder order)
    {
        this.order = order;
    }

    public IOnlineTakeawayOrderState Cook()
    {
        // handle this scenario accordingly
```

continues

LISTING 16-18 *(continued)*

```
            ...

            return new InOven(order);
        }

        // all below methods are not applicable in this state
        public IOnlineTakeawayOrderState TakeOutOfOven()
        {
            throw new ActionNotPermittedInThisState();
        }

        public IOnlineTakeawayOrderState Package()
        {
            throw new ActionNotPermittedInThisState();
        }

        public IOnlineTakeawayOrderState Deliver()
        {
            throw new ActionNotPermittedInThisState();
        }
    }
```

Listing 16-18 accentuates the biggest criticisms of the state pattern: it can result in massive amounts of boilerplate code and unimplemented methods. These problems are made clear by `InKitchenQueue`, which is one of five possible states in this simplified scenario. Each implements only one method purposefully. For the methods that aren't valid in a particular state, the state implementation throws an `ActionNotPermittedInState` exception. Although this example is extreme in that each state implementation handles only a single method, many implementations of the state pattern really are far noisier and more verbose than this.

More relevant to DDD is the fact that the state pattern is less explicit. By allowing methods that should not be called, the state pattern does not explicitly model the rules of the domain. If you cannot package an order that is in the oven, some DDD practitioners argue that the type system should reinforce this explicitly. You reach this ideal by having a separate entity for each state, with only the applicable operations for the state modeled as part of the entity's interface. Listing 16-19 shows the result of applying this philosophy to the online takeaway scenario.

LISTING 16-19: Replacing the State Pattern with Explicit State Modeling

```
// these entities collectively replace the OnlineTakeawayOrder entity
public class InKitchenOnlineTakeawayOrder
{
    public InKitchenOnlineTakeawayOrder(Guid id, Address address)
    {
        ...

        this.Id = id;
        this.Address = address;
```

```
    }

    public Guid Id { get; private set; }

    public Address Address { get; private set; }

    // only contains methods it actually implements
    // returns new state so that clients have to be aware of it
    public InOvenOnlineTakeawayOrder Cook()
    {
        ...

        return new InOvenOnlineTakeawayOrder(this.Id, this.Address);
    }
}

public class InOvenOnlineTakeawayOrder
{
    public InOvenOnlineTakeawayOrder(Guid id, Address address)
    {
        ...

        this.Id = id;
        this.Address = address;
    }

    public Guid Id { get; private set; }

    public Address Address { get; private set; }

    public CookedOnlineTakeawayOrder TakeOutOfOven()
    {
        ...

        return new CookedOnlineTakeawayOrder(this.Id, this.Address);
    }
}

// the other three states are available in this chapter's sample code
```

Listing 16-19 shows two new entities that make the `IOnlineTakeawayOrderState` and its implementations completely redundant. You can see that the noise has been cut down and the boilerplate is completely gone. However, the real benefit is that the rules of the domain are now much clearer in the code; no longer is there a single `OnlineTakeawayOrder` entity that is passed around the domain model allowing invalid methods to be called on it. Instead, the domain model must explicitly specify which state it needs to operate on, resulting in more expressive domain rules that the type system enforces.

Avoiding Getters and Setters with the Memento Pattern

If you buy into the guidance presented in this chapter that avoiding getters and setters helps to create behavior-rich domain models, you are actually quite likely to run into a common problem: getting

data out of your domain models. You may want to present information on a UI or send e-mails to customers, but if the information you need is hidden away inside entities, something has to give. You may feel compelled to expose a getter, and sometimes that may be the best choice. Intriguingly, though, the memento pattern does give you another option.

With the memento pattern, you create a snapshot of your entity's state. The snapshot is bundled out the backdoor of your entity, allowing you to harness its data in other parts of your application, like the UI. But you still maintain some of the benefits of encapsulation, because the structure of the entity's state is still private. In essence, mementos represent a stable interface that other parts of the application can become coupled to, while the innards of your entity can still be refactored with little friction.

One part of an e-commerce domain model that might benefit from the memento pattern is a shopping basket, as Listing 16-20 illustrates.

LISTING 16-20: Maintaining Encapsulation with the Memento Pattern

```
public class Basket
{
    // there are no public setters or getters
    private int Id {get;set}
    private int Cost {get; set;}
    private List<Item> Items {get; set;}

    public void Add(Item item)
    {
        ...
    }

    public void Add(Coupon coupon)
    {
        ...
    }

    // okay for outside world to be coupled to BasketSnapshot
    public BasketSnapshot GetSnapshot()
    {
        return new BasketSnapshot(this.Id, this.Cost, this.Items.Count(),.......)
    }

    ...
}
```

`Basket.GetSnapshot()` in Listing 16-20 provides a snapshot of the `Basket` entity's state—that is, a memento. By exposing its data in this way, its internal state—`Id`, `Cost`, and `Items`—remains encapsulated from the rest of the application.

Favor Hidden-Side-Effect-Free Functions

Side effects can make code harder to reason about and harder to test, and they can often be the source of bugs. In a broad programming context, avoiding side effecting functions as much as possible is generally considered good advice. You even saw in the previous chapter how being side-effect-free and immutable were two of the main strengths of value objects. But if avoiding side effects is good advice, avoiding hidden side effects is a fundamental expectation.

It is likely that many of an entity's operations need to perform side effects. In particular, many methods on an entity likely need to update its encapsulated state. But this pattern of thinking can lead to the unnecessary habit of making all of an entity's methods side effecting, and worse—home to hidden side effects. Accordingly, you should be on the lookout for opportunities to replace side effecting methods, like `Dice.Value()` shown in Listing 16-21, with side effect-free implementations instead.

LISTING 16-21: Entity Behavior with Hidden Side Effect

```
public class Dice
{
    private Random r = new Random();

    public Dice(Guid id)
    {
        ...

        this.Id = id;
    }

    public Guid Id { get; private set; }

    // Bad: looks like a query, but changes every time
    public int Value()
    {
        return r.Next(1, 7);
    }

    ...
}
```

When you focus on encapsulating state and creating behavior-rich domain models, it can be easy to write code like the `Dice` entity in Listing 16-21 that houses unnecessary side effects. Clients of `Dice` are likely to assume that calling `Value()` will get the value of the dice the last time it was rolled. This is because `Value()` sounds like a query or property. As you can see though, calling `Value()` has the hidden side effect of changing the value. So a client of `Dice` may call `Value()` twice expecting to see the same value, without realizing it doesn't behave that way.

A more sensible implementation is shown in Listing 16-22. In this example Dice has a `Roll()` method. `Roll()` sounds like a command, alerting clients that this method has side effects. In addition there is a void return type making this even clearer. You can also see that `Value` is now a property. It sounds like a read and faithfully all it does is read the current value of the dice. No hidden side effects.

LISTING 16-22: Side-Effect-Free Entity Behavior

```
public class Dice
{
    private Random r = new Random();

    public Dice(Guid id)
    {
```

continues

LISTING 16-22 *(continued)*

```
        ...

        this.Id = id;
    }

    public Guid Id { get; private set; }

    // Good: does not change each time called
    public int Value { get; private set; }

    // Good: sounds like a command - side effect expected
    public void Roll()
    {
        Value = r.Next(1, 7);
    }

    ...
}
```

THE SALIENT POINTS

- Entities are domain concepts that have a unique identity in the problem domain.
- Having a life cycle is also a distinguishing characteristic of entities.
- Discussions with domain experts are a common way of uncovering entities in a domain, but you have to pay attention to their wording.
- Value objects are the inverse of entities; they have no identity, and their equality is based on representing the same value.
- Entities and value objects are context dependent; an entity in one domain might be a value object in another.
- Choosing an entity's ID is a fundamental implementation concern.
- Natural keys from the problem domain, application generated, and datastore generated are all techniques for creating entity IDs.
- Entities should always be valid for the given context.
- Invariants are fundamental truths about an entity, so they should always be enforced.
- An entity's behavior is dictated by the needs of the application being built. It is not about modeling every behavior in the real world.
- Be careful of modeling physical concepts as single entities. The typical `Customer` entity can often be logically split across multiple bounded contexts into numerous entities.
- Be wary of using the state pattern for modeling entity life cycles; often it can be a messy way to hide domain concepts.
- Consider patterns like the memento pattern that allow you to maintain behavior-rich domain models without exposing the structure of your entities.

17

Domain Services

WHAT'S IN THIS CHAPTER?

- An introduction to domain services
- A disambiguation from other types of service
- Advice about when to consider using domain services
- Examples of domain services for a variety of domains
- A discussion, with several examples and techniques, on how to use domain services from within the application layer and the domain model

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 17 download and individually named according to the names throughout the chapter.

When building domain models, you sometimes come across concepts or behavior that do not logically sit comfortably within an entity or aggregate in the system. This is an implicit sign that you may need to create a domain service.

Confusingly, *service* is an overloaded term. Fortunately, though, domain services are easily distinguished based on two defining characteristics: they represent domain concepts, and they are stateless. You are most likely to use domain services to orchestrate entities and encapsulate business policies rather than carry out infrastructural plumbing; leave that to application services.

In this chapter, you learn how to build domain services that cater to different scenarios. These scenarios include exposing domain services as contracts that are implemented outside the domain model, and building pure domain services that contain important business rules

and reside entirely within the domain model. This chapter also shows some of the ways that your domain services can be consumed from both the service layer and the domain model.

Before moving on to the concrete examples, this chapter elaborates on the motives for using domain services and provides further clarification of precisely what they are.

UNDERSTANDING DOMAIN SERVICES

This book contains information about Service Oriented Architecture (SOA), application services, and domain services. None of these concepts are closely related, yet there is still lots of confusion among developers and on mailing lists about the differences between them. Conceptually, domain services represent domain concepts; they are behaviors that exist in the problem domain, they occur in conversations with domain experts, and they are certainly part of the ubiquitous language (UL). If you can remember those important characteristics, you will find domain services to be a useful tool.

A few technical characteristics of domain services are important to be aware of so that you can model them in your code effectively. You will learn about them shortly.

When to Use a Domain Service

You've spoken to domain experts about a domain concept that involves multiple entities, but you're unsure about which entity "owns" the behavior. It doesn't appear to belong to any of them, and it seems awkward when you try to force-fit it onto either of them. This pattern of thinking is a strong indicator of the need for a domain service.

Encapsulating Business Policies and Processes

A primary concern for domain services is carrying out some behavior involving entities or value objects. This can be demonstrated using two `Competitor` entities and an `OnlineDeathmatch` entity, all taken from the domain model of an online gaming system. They are shown in Listing 17-1 and Listing 17-2, respectively.

LISTING 17-1: Competitor Entity

```
public class Competitor
{
    ...

    public Guid Id {get; protected set;}

    public string GamerTag { get; protected set; }

    public Score Score { get; set; }

    public Ranking WorldRanking { get; set; }
}
```

LISTING 17-2: OnlineDeathmatch Entity that Contains Logic for Calculating Scores

```
public class OnlineDeathmatch : IGame
{
    private Competitor player1;
    private Competitor player2;

    public OnlineDeathmatch(Competitor player1, Competitor player2, Guid id)
    {
        ...

        this.player1 = player1;
        this.player2 = player2;
        this.Id = id;
    }

    public Guid Id { get; protected set; }

    public void CommenceBattle()
    {
        ...

        // battle completes

        // the following would actually be based on game result
        var winner = player1;
        var loser = player2;

        UpdateScoresAndRewards(winner, loser);
    }

    private void UpdateScoresAndRewards(Competitor winner, Competitor loser)
    {
        // real system uses rankings, history, bonus points, in-game actions
        // seasonal promotions, marketing campaigns
        winner.Score = winner.Score.Add(new Score(200));
        loser.Score = loser.Score.Subtract(new Score(200));
    }
}
```

On completion of an `OnlineDeathMatch`, each player has his score updated based on how he performed in the game. One option for locating this logic is to place the score calculation and reward logic within the `OnlineDeathmatch`, as shown in Listing 17-3.

One big problem with `CalculateNewPlayerScores()` being part of the `OnlineDeathMatch` is that the rules are flexible; the business may want to award double points or hand out special prizes at certain times. What the business is saying in cases like this is that score and reward calculation are fundamentally important concepts within themselves. The same special promotions are often applied to different types of games: `TeamBattle`, `CaptureTheFlag`, `GangCrusade`, and so on.

This scenario is hinting at the need for domain services that encapsulate the single responsibility of a specific score or reward policy. Listing 17-4 shows the `IGamingScorePolicy` and

`IGamingRewardPolicy` domain service interfaces that you can inject and apply into an `OnlineDeathmatch` or any other type of domain object that needs them.

LISTING 17-3: Interfaces for Domain Services

```
// domain service interfaces - part of UL
public interface IGamingScorePolicy
{
    void Apply(IGame game);
}

public interface IGamingRewardPolicy
{
    void Apply(IGame game);
}
```

LISTING 17-4: Utilizing Domain Services that Orchestrate Entities

```
public class OnlineDeathmatch : IGame
{
    private Competitor player1;
    private Competitor player2;

    // domain services
    private IEnumerable<IGamingScorePolicy> scorers;
    private IEnumerable<IGamingRewardPolicy> rewarders;

    public OnlineDeathmatch(Competitor player1, Competitor player2,
                  Guid id, IEnumerable<IGamingScorePolicy> scorers,
                  IEnumerable<IGamingRewardPolicy> rewarders)
    {
        this.player1 = player1;
        this.player2 = player2;
        this.Id = id;
        this.scorers = scorers;
        this.rewarders = rewarders;
    }

    public Guid Id { get; protected set; }

    public void CommenceBattle()
    {
        ...

        // battle completes

        // the following would actually be based on game result
        this.Winners.Add(player1);
        this.Losers.Add(player2);

        UpdateScoresAndRewards();
```

```
    }

    private void UpdateScoresAndRewards()
    {
        foreach (var scorer in scorers)
        {
            scorer.Apply(this);
        }

        foreach (var rewarder in rewarders)
        {
            rewarder.Apply(this);
        }
    }
}
```

Using the approach in Listing 17-4, you can switch in and out specific score and loyalty calculations according to the latest business promotions. Equally as importantly, you can now make these concepts explicit in the domain. For example, when a business is running its "free 12-month subscription for high scores" promotion, you can model this explicitly in the domain as a `Free12MonthSubscriptionForHighScoresRewardPolicy` domain service, as shown in Listing 17-5.

LISTING 17-5: Concrete Implementation of a Pure Domain Service

```
// domain service implementation; part of the UL
public class Free12MonthSubscriptionForHighScoresReward : IGamingRewardPolicy
{
    private IScoreFinder repository;

    public Free12MonthSubscriptionForHighScoresReward(IScoreFinder repository)
    {
        this.repository = repository;
    }

    public void Apply(IGame game)
    {
        var top100Threshold = repository.FindTopScore(game, 100);
        if (game.Winners.Single().Score > top100Threshold)
        {
            // trigger reward process
        }
    }
}
```

Listing 17-5 shows the `Free12MonthSubscriptionForHighScoresRewardPolicy` domain service that explicitly defines an important domain concept that exists within the UL. This domain service contains important business rules, lives inside the domain model, and hence is pure. However, some domain services are not implemented within the domain model; instead, their implementation lives in the service layer.

Representing Contracts

The other broad use case for domain services is as a contract—where the concept itself is important to the domain, but the implementation relies on infrastructure that cannot be used in the domain model. An example of this is `ShippingRouteFinder`, shown in Listing 17-6. You can see that it makes a web request to the routing API so that it can find the available routes for the given journey endpoints.

LISTING 17-6: Implementation of Domain Service that Resides in the Service Layer and Not the Domain Model

```
// implementation of domain service - this would live in the service layer
public class ShippingRouteFinder : IShippingRouteFinder
{
    public Route FindFastestRoute(Location departing, Location destination,
                                  DateTime departureDate)
    {
        // this method makes an HTTP call; best to keep out of the domain
        var response = QueryRoutingApi(departing, destination, departureDate);

        var route = ParseRoute(response);

        return route;
    }

    // ...
}

// interface for domain service - this would live in the domain model
// this is the "contract"
public interface IShippingRouteFinder
{
    Route FindFastestRoute(Location departing, Location destination,
                           DateTime departureDate);
}
```

Importantly, the logic of making HTTP requests to a web server cannot live in the domain because it is an infrastructural concern; therefore, the implementation of the `ShippingRouteFinder` domain service in Listing 17-6 cannot live in the domain model. But the concept is part of the UL, so the interface must.

You can use domain services as contracts for a variety of scenarios, including:

- Entity identification
- Exchange rate lookup
- Tax lookup
- Real-time notifications

Anatomy of a Domain Service

Domain services have three fundamental technical characteristics: they represent behavior, and thus have no identity; they are stateless; and they often orchestrate multiple entities or domain objects. In addition to the examples in the previous section, the `RomanceOMeter` in Listing 17-7 highlights these characteristics. The `RomanceOMeter` is inspired by a similar concept in the domain model of an online dating website where it is used to assess how compatible two love seekers are.

LISTING 17-7: A Quintessential Stateless, Behavior-Only, Entity-Orchestrating Domain Service

```
// domain service - part of UL
public class RomanceOMeter
{
    // stateless - collaborators are allowed, though

    // behavior-only
    public CompatibilityRating AssessCompatibility(LoveSeeker seeker1,
                                                   LoveSeeker seeker2)
    {
        var rating = new CompatibilityRating(0);

        // orchestrate entities:
        // compare dating history, blood type, lifestyle etc
        if (seeker1.BloodType == seeker2.BloodType)
        {
            rating = rating + new CompatibilityRating(250);
        }

        ...

        // return another domain object (value object in this case)
        return rating;
    }
}
```

There's no ID or identification-related state on the `RomanceOMeter` in Listing 17-7. It is pure behavior, containing only `AssessCompatibility()` that performs the romance calculation. As you can also see, it keeps no state and orchestrates multiple `LoveSeeker` entities; the calculation is based entirely on the inputs. In doing so, it satisfies the stateless requirement and exemplifies how domain services orchestrate other domain objects.

Avoiding Anemic Domain Models

After accepting that not all domain logic needs to live on entities directly and that domain services are a useful concept, you need to be careful not to push too much logic into domain services, which

can lead to inaccurate, confusing, and anemic domain models with low conceptual cohesion. Clearly, that would counter some of DDD's major benefits. However, as long as you pay enough attention, it's unlikely to become a problem.

Finding the right balance between too few and too many domain services is easier with experience. But care and logical thinking help you get the majority of decisions correct. For instance, one pattern of thinking that should evoke thoughts of using a domain service occurs when adding new behavior to an entity is an awkward fit and just doesn't feel right, and it doesn't align with what domain experts are saying. You can then discuss the issue further with those domain experts and listen to how they describe the concept. Do they always refer to an entity when discussing the concept, or do they discuss it in isolation? At the other extreme, if you are creating a lot of domain services, maybe your thought process is a little too liberal.

Contrasting with Application Services

A common source of confusion is differentiating application from domain services. Once you understand the conceptual role of both, though, you'll never be confused again. As you've seen in this chapter, domain services represent concepts that exist within the problem domain, and at a minimum, their interface lives in the domain model. Conversely, application services do not represent domain concepts, and they don't contain business rules. Also, they don't even partially live in the domain model—not even their interfaces. As you will see in Chapter 25, "Commands: Application Service Patterns for Processing Business Use Cases," application services live in the service layer and deal will pulling together infrastructural concerns, like transactions, to carry out full business use cases.

A question that appears regularly on DDD forums is: What's the ideal way to handle authentication and authorization in a domain service? The simple answer is that you don't, because that is a responsibility of the service layer. This example typifies the confusion around the two types of service and helps you understand the role of each by knowing why it is a bad question (but completely understandable due to the overloaded term *service*).

You may still be thinking that domain and application services are similar in that they both may have to deal with infrastructural concerns. It's also easy to make a distinction here. Domain services rely on infrastructure that is used to inform domain logic. Conversely, infrastructural concerns in an application service are there to enable the domain model to execute correctly. A domain service makes HTTP calls to a web service or writes something to disk as part of domain logic, but an application service wraps the domain model in a transaction or creates database connections so that the code can run as a single use case.

> **NOTE** *Feel free to visit Wrox's discussion forum at* `http://p2p.wrox.com/` *if you would like to discuss further the difference between application and domain services.*

UTILIZING DOMAIN SERVICES

Having learned about the concept of domain services and seen practical examples, the remaining detail is in understanding how to use them. Some options require little explanation, as is the case with using domain services inside application services. Contentiously, though, domain services often need to be used as a step in domain processes that reside fully in the domain model. This leads to the debate of injecting domain services into entities. As you'll see shortly, this is a solved problem, but no solution is without its trade-offs and detractors.

In the Service Layer

Starting with the easiest case, domain services can be put to use within application services. As part of its role in fulfilling a full business use case, an application service can pull the relevant entities out of a repository and pass them into a domain service, as shown in Listing 17-8.

LISTING 17-8: Pulling Together Domain Services and Entities from Within an Application Service

```
// application service
public class MultiMemberInsurancePremium
{
    private IPolicyRepository policyRepository;
    private IMemberRepository memberRepository;

    // domain service
    private IMultiMemberPremiumCalculator calculator;

    public MultiMemberInsurancePremium(IPolicyRepository policyRepository,
                                       IMemberRepository memberRepository,
                                  IMultiMemberPremiumCalculator calculator)
    {
        this.policyRepository = policyRepository;
        this.memberRepository = memberRepository;
        this.calculator = calculator;
    }

    public Quote GetQuote(int policyId, IEnumerable<int> memberIds)
    {
        var existingMainPolicy = policyRepository.Get(policyId);
        var additionalMembers = memberRepository.Get(memberIds);
        // pass entities into domain service
        var multiMemberQuote = calculator.CalculatePremium(
            existingMainPolicy, additionalMembers
        );

        return multiMemberQuote;
    }
}
```

In Listing 17-8, the `MultiMemberInsurancePremium` application service pieces together the `IMultiMemberPremiumCalculator` with the `Policy` and `Member` entities that are required by its `CalculatePremium()`. This illustrates that, when used in the service layer, domain services and other domain objects like entities can be easily pieced together as needed. This convenience may not always be achievable, though, like in cases where an entity appears to be dependent on a domain service and the domain objects need to be coordinated within the domain model.

In the Domain

Sometimes an entity needs a domain service to carry out its behavior in a way that precludes piecing them together in an application service. A typical example is when a notification needs to occur following an entity executing some task. Listing 17-9 highlights this type of scenario by showing a `RestaurantBooking` entity that triggers a `NotifyBookingConfirmation` when a customer confirms a restaurant booking. The `RestaurantBooking` entity directly depends on the `IRestaurantNotifier` domain service to notify the restaurant of the booking confirmation.

LISTING 17-9: An Entity that Appears to Depend on a Domain Service

```
// entity
public class RestaurantBooking
{
    ...

    public Guid Id { get; protected set; }

    public Restaurant Restaurant { get; protected set; }

    public Customer Customer { get; protected set; }

    public BookingDetails BookingDetails { get; protected set;}

    public bool Confirmed { get; protected set; }

    ...

    // entity behavior that depends on a domain service
    public void ConfirmBooking()
    {
        ...

        // restaurantNotifier is the domain service
        // how can it be in scope, especially when an ORM is involved?
        restaurantNotifier.NotifyBookingConfirmation(
            Restaurant, Customer, Id, BookingDetails
        );

        ...
    }

    ...
}
```

In Listing 17-9, the challenge is to have the `restaurantNotifier`, an instance of `IRestaurantNotifier`, available within the scope of `ConfirmBooking()`. It may seem easy with plain old constructor injection as the obvious choice. It's not always that straightforward, though; often object-relational mappers (ORM)s are used to manage the life cycle of entities, removing the developer's ability to pass in dependencies at object construction. Each of the following techniques aims to alleviate this problem.

Manually Wiring Up

If an entity or other domain object depends on a domain service, you can pass the relevant service into the constructor if you are managing object construction yourself. Listing 17-10 exemplifies the desirable solution of using a factory method to solve this problem.

LISTING 17-10: Manually Passing a Domain Service into an Entity with a Factory Method

```
public static class RestaurantBookingFactory
{
    // initialize object - no ORM to worry about here
    public RestaurantBooking CreateBooking(Restaurant restaurant,
                        Customer customer, BookingDetails details)
    {
        var id = Guid.NewGuid();
        var notifier = new RestaurantNotifier();
        return new RestaurantBooking(restaurant, customer, details, id, notifier);
    }
}
```

Listing 17-10 shows how you can pass a `RestaurantNotifier` domain service into the constructor of the `RestaurantBooking` entity that is manually constructed inside the `CreateBooking()` factory method. Factories are a common pattern that you see in many codebases. So far so good. But this solution becomes problematic in DDD when you don't handle construction of the entity yourself—typically when an ORM loads an entity from persistence.

You can still work around an object's life cycle that an ORM is managing, but the solution is not necessarily pretty. Listing 17-11 shows how your factory methods can add a second stage of object construction that sets the domain service as a property on the entity after an ORM has constructed it. In a similar fashion, you can have an `Init()` method that takes all the dependencies and sets them, like a pseudo-constructor. Either way, you're open to the problem of forgetting that construction of the object occurs in two phases, which can lead to inconsistent states in production systems with lots of ensuing head scratching.

LISTING 17-11: Two-Phase Construction When ORMs Are Involved

```
public class RestaurantBookingRepository
{
    // interface provided by ORM
    private ISession session;

    public RestaurantBookingRepository(ISession session)
```

continues

LISTING 17-11 *(continued)*

```
    {
        this.session = session;
    }

    public RestaurantBooking Get(Guid restaurantBookingId)
    {
        // first phase of construction handled by ORM
        var booking = session.Load<RestaurantBooking>(restaurantBookingId);

        // second phase of construction handled manually
        booking.Init(new RestaurantNotifier());
        // alternatively: booking.Notifier = new RestaurantNotifier();
    }
}
```

You may have thought of other problems associated with two-phase construction, such as remembering to apply the pattern consistently across the codebase and being disciplined to ensure that `Init()` is not called again after the object has been constructed.

If all this manual configuration is undesirable for you, you may prefer to offload some of it to an Inversion of Control (IoC) container by using dependency injection.

Using Dependency Injection

Another approach is to dependency-inject domain services into the entities that need to use them. Once a domain service has been added as a constructor parameter, the remaining task is to wire up the desired implementation using standard dependency injection. Using dependency injection saves you the hassle of manually constructing objects. It's mainly a question of style whether you choose this or manually wiring up your dependencies, though, because the problem of dealing with ORMs may still be present.

With some ORMs, it's possible to have them wire up dependencies that your IoC container manages. Unfortunately, though, with many it is not. However, if you're still determined to use an IoC container and an ORM, the service locator pattern is one potential solution.

Using a Service Locator

Advance warning: the service locator pattern is fairly controversial; you'll learn why shortly. However, you can see in Listing 17-12 how it solves the problem of allowing IoC containers to handle setting the dependencies of entities that are loaded by ORMs, without the need for manual intervention.

LISTING 17-12: Using IoC Containers and ORMs via a Service Locator

```
public class RestaurantBooking
{
    // hard-coded to fetch dependency using IoC container
    private IRestaurantNotifier restaurantNotifier =
```

```
            ServiceLocater.Get<IRestaurantNotifier>();

        public Guid Id { get; protected set; }

        public Restaurant Restaurant { get; protected set; }

        public Customer Customer { get; protected set; }

        ...
    }
```

By taking the hard-coded dependency on the `ServiceLocator` class shown in Listing 17-12, you can load all of an entity's dependencies from an IoC container during object construction after an ORM has initialized it. This removes the need for manual construction steps. But there's a price; now the entity is tightly coupled to the `ServiceLocator`—an infrastructural concern that ideally you don't want polluting the domain model.

The real problems with using a service locator arise from the tight coupling; you have to mock mocks in your tests, for example. Another problem is the obscuring of an object's dependencies, because they're no longer passed into the constructor.

Applying Double Dispatch

If you're happy to forego passing domain services into entities at construction time, you have the option of instead passing them as method arguments using the double dispatch pattern. With double dispatch, a domain service is passed into a method on an entity, and the entity then passes itself into a method on the domain service, as shown in Listing 17-13.

LISTING 17-13: Using Double Dispatch to Supply Domain Services to Entities

```
public void ConfirmBooking(IRestaurantNotifier restaurantNotifier)
{
    ...

    Confirmed = restaurantNotifier.NotifyBookingConfirmation(this);

    ...
}
```

As Listing 17-13 shows, domain services no longer need to be supplied into an entity's constructor. Instead, they can be passed into methods on an entity by an application service. This approach hasn't proven to be massively popular for a few reasons. Some feel that passing dependencies into methods requires callers of the method to supply the dependency—a responsibility that shouldn't belong to them. Others argue that dependencies normally go via the constructor because they should not be part of a method's signature; they're an implementation detail that may be changed, whereas the method signature should not be so volatile.

So double dispatch is not to everyone's liking, but neither are any of the other options presented so far. It's definitely best to explore the idea yourself and assess how it may work on your projects. One

approach that has become popular, though, is using the domain events pattern, with the promise of truly decoupling entities from domain services.

Decoupling with Domain Events

An interesting pattern that completely avoids the need for injecting domain services into entities is domain events. When important actions occur, an entity can raise a domain event that is handled by subscribers who are registered for that event. As you might have guessed, domain services can reside within a subscriber, and therefore, not an entity.

Listing 17-14 highlights the `RestaurantBooking` entity raising a `BookingConfirmedByCustomer` domain event. Listing 17-15 shows an event handler, also known as a subscriber, which handles this type of event by invoking the `IRestaurantNotifier` domain service. Finally, Listing 17-16 shows the small amount of work required to stitch these components together.

LISTING 17-14: An Entity Raising a Domain Event

```
public class RestaurantBooking
{
    public Guid Id { get; protected set; }

    ...

    public void ConfirmBooking()
    {
        ...

        DomainEvents.Raise(new BookingConfirmedByCustomer(this));

        ...
    }

    ...
}
```

LISTING 17-15: A Domain Service Being Applied in an Event Handler

```
// domain event handler
public class NotifyRestaurantOnCustomerBookingConfirmation :
             IHandleEvents<BookingConfirmedByCustomer>
{
    private IRestaurantNotifier restaurantNotifier;

    public NotifyRestaurantOnCustomerBookingConfirmation(
                  IRestaurantNotifier restaurantNotifier)
    {
        this.restaurantNotifier = restaurantNotifier;
    }

    public void Handle(BookingConfirmedByCustomer @event)
```

```
    {
        // now the restaurant does not control its own state as much,
        // but the entity knows nothing about the domain service
        var booking = @event.RestaurantBooking;
        booking.Confirmed =
            restaurantNotifier.NotifyBookingConfirmation(
                booking.Restaurant, booking.Customer,
                booking.Id, booking.BookingDetails
            );
    }
}
```

LISTING 17-16: Registering an Event Handler with a Domain Event

```
var notifier = new RestaurantNotifer();
var handler = new NotifyRestaurantOnCustomerBookingConfirmation(notifier);
DomainEvents.Regsiter<BookingConfirmedByCustomer>(handler);
```

Listings 17-14 to 17-16 show a minimum possible example of the domain events pattern applied to the ongoing restaurant booking example. Chapter 18, "Domain Events," is dedicated to the domain events pattern and provides more in-depth examples that explore the strengths and weaknesses of the pattern. One obvious drawback that you may have discerned from Listings 17-14 to 17-16 is that the logic is now distributed between the `RestaurantBooking` entity and the `NotifyRestaurantOnCustomerBookingConfirmation` event handler, whereas with other patterns this would be a single piece of sequential code.

Should Entities Even Know About Domain Services?

In the previous example, you saw how domain events preclude injecting domain services into entities, whereas each of the other patterns goes in the opposite direction and tries to find a way to make the dependency feasible. The latter is a fairly contentious approach within the DDD community; many practitioners argue that it's just a bad idea. Ultimately, you need to use context, personal preferences, and your experience to decide which option you like the best.

THE SALIENT POINTS

- Sometimes behavior does not belong to an entity or value object yet is still an important domain concept; this is hinting at the need for a domain service.
- Domain services represent domain concepts; they are part of the UL.
- Domain services are often used to orchestrate entities and value objects as part of stateless operations.
- Too many domain services can lead to an anemic domain model that does not align well with the problem domain.

- Too few domain services can lead to logic being incorrectly located on entities or value objects. This causes distinct concepts to be mixed up, which reduces clarity.
- Domain services are also used as contracts, where the interface lives in the domain model, but the implementation does not.
- When an entity depends on a domain service, a variety of options can be used, including dependency injection, double dispatch, and domain events.

18

Domain Events

WHAT'S IN THIS CHAPTER?

- An introduction to the domain events design pattern
- An explanation of domain events' core concepts
- Multiple domain events' implementation patterns
- Testing applications that use domain events

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 18 download and individually named according to the names throughout the chapter.

Domain-Driven Design (DDD) practitioners have found that they can better understand the problem domain by learning about the events that occur within it—not just the entities. These events, known as *domain events*, will be uncovered during knowledge-crunching sessions with domain experts. Uncovering domain events is so valuable that DDD practitioners have innovated knowledge-crunching techniques to make them more event-focused using practices such as event storming (`http://ziobrando.blogspot.co.uk/2013/11/introducing-event-storming.html#.U5YPynU-mHs`). With these innovations, though, come new challenges. Now that conceptual models are event-centric, the code also needs to be event-centric so it can express the conceptual model. This is where the domain events design pattern adds value.

Part II focused on events that are sent as asynchronous messages between bounded contexts using transports like a message bus or REST. However, the domain events design pattern is generally used as a single-threaded pattern inside a domain model within a bounded context. In this chapter you will see examples that explore the different implementation options and overall trade-offs involved when using domain events.

Before getting started, it's important to acknowledge that using domain events does not necessitate using event sourcing. Unfortunately, this is a common misconception; it is entirely possible to use traditional persistence options like a SQL database. However, applying event sourcing or asynchronous messaging in combination with a domain model that uses domain events is often a good combination. Chapter 22, "Event Sourcing," will clarify this relationship.

Throughout this chapter, each implementation pattern for domain events will be used to implement a single use case from an online pizza delivery service. This way you can see direct comparisons. The use case being modeled is the delivery guarantee, where a customer will receive a discount if their pizza is not delivered in a certain timeframe.

> **WARNING** Domain events *occur in the problem domain, whereas the* domain events design pattern *is a way to model them in software. So theoretically, the term* domain events *can apply to both of these concepts. This ambiguity can sometimes lead to confusion. In this chapter, it mostly refers to the design pattern except where noted otherwise.*

ESSENCE OF THE DOMAIN EVENTS PATTERN

If you understand the publish-subscribe pattern or C# events, you will grasp the domain events pattern very quickly. Essentially, you use an infrastructural component, sometimes from within your domain model, to publish events. By default, events are then processed synchronously inside the same thread by each event handler that has been registered for that type of event. Asynchronous event handling is also possible with this pattern.

Important Domain Occurrences That Have Already Happened

Events are just immutable classes with public properties (data objects, Plain Old C# Objects [POCOs], Plain Old Java Objects [POJOs]) representing important events in the problem domain. Listing 18-1 shows a `DeliveryGuaranteeFailed` event, which is fired in the domain when a pizza delivery took longer than promised.

LISTING 18-1: An Event

```
public class DeliveryGuaranteeFailed
{
    public DeliveryGuaranteeFailed(OrderForDelivery order)
    {
        Order = order;
    }

    public OrderForDelivery Order { get; private set; }
}
```

Notice in Listing 18-1 that the `DeliveryGuaranteeFailed` event is just a class with properties; this is all that your events need to be.

Reacting to Events

In response to events, event handlers are executed. Listing 18-2 shows how you can register an event handler as a method to be called (`onDeliveryFailure`) when the event (`DeliveryGuaranteeFailed`) is raised.

LISTING 18-2: An Event Handler

```
public void Confirm(DateTime timeThatPizzaWasDelivered, Guid orderId)
{
    using(DomainEvents.Register<DeliveryGuaranteeFailed>(onDeliveryFailure))
    {
        var order = orderRepository.FindBy(orderId);
        order.ConfirmReceipt(timeThatPizzaWasDelivered);
    }
}

private void onDeliveryFailure(DeliveryGuaranteeFailed evnt)
{
    // handle domain event
}
```

Optional Asynchrony

Using the domain events pattern can be synergistic with asynchronous workflows, including communication between bounded contexts. As you saw in Part II on bounded context integration, asynchronous messages between bounded contexts are a modern solution that help with reliability and scalability. You can trigger these processes from within your domain events event handlers. Sometimes you may even want asynchronous, reliable communication within a bounded context, for scenarios like implementing eventually consistent aggregates. Using the decoupled nature of domain events can help you in both scenarios.

Importantly, when creating event handlers that trigger asynchronous workflows, you need to be clear about transactional boundaries. For example, if one event handler updates the database, and another publishes a message, you would want both operations to roll back if either of them failed, as Figure 18-1 shows.

> **NOTE** *Aggregates are cohesive groupings of entities and value objects. Chapter 19, "Aggregates," provides in-depth coverage of aggregates.*

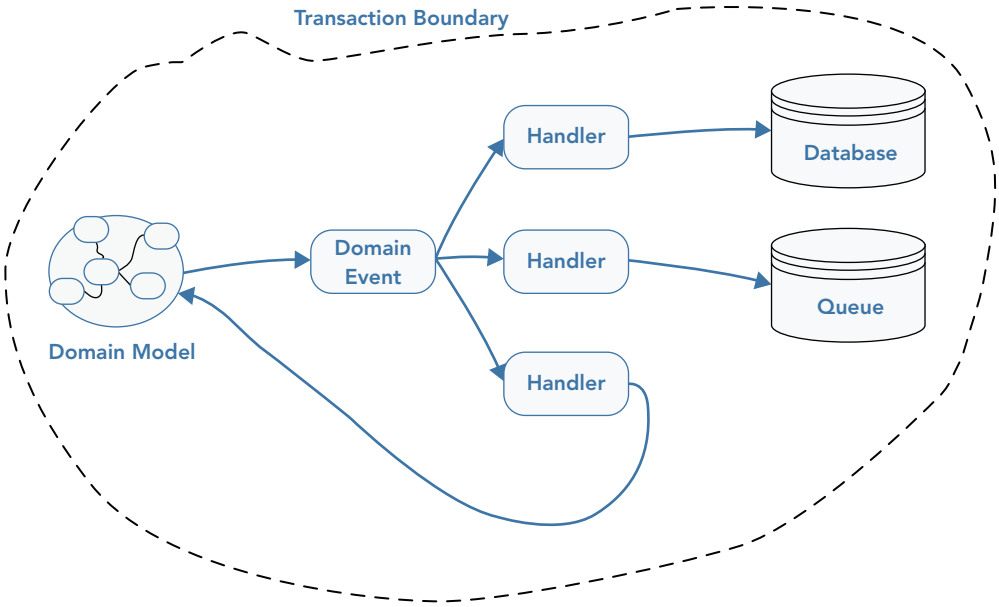


FIGURE 18-1: Ensuring correct transactional behavior

Internal vs External Events

An important distinction needs to be made when using the domain events pattern to avoid confusion that can lead to poor technical implementations. It is crucial that you are aware of the difference between internal and external events. Internal events are internal to a domain model–they are not shared between bounded contexts. In this chapter, you will see how the domain events pattern uses internal events, whereas you saw external events in Part II of this book.

Differentiating internal and external events is important because they have different characteristics. Because internal events are limited in scope to a single bounded context, it is Ok to put domain objects on them, as the example in Listing 18-1 showed. This poses no risk, because other bounded contexts cannot become coupled to these domain objects. Conversely, external events tend to be flat in structure, exposing just a few properties—most of the time just correlational IDs, as typified in Listing 18-3.

LISTING 18-3: External Events Cannot Expose Domain Objects

```
public class DeliveryGuaranteeFailed
{
    public DeliveryGuaranteeFailed(Guid orderId)
    {
        this.OrderId = orderId;
    }

    public Guid OrderId { get; private set; }
}
```

You learned in Part II that external events need to be versioned to avoid breaking changes. This is another differentiator with internal events, because if you make breaking changes to an internal event your code will not compile (if using a compiled programming language). So there's no need to version internal events.

As you start to implement domain events, you will see that in a typical business use case there may be a number of internal events raised, and just one or two external events that are raised by the service layer. Figure 18-2 illustrates how the sequence of events may occur in a typical use case.

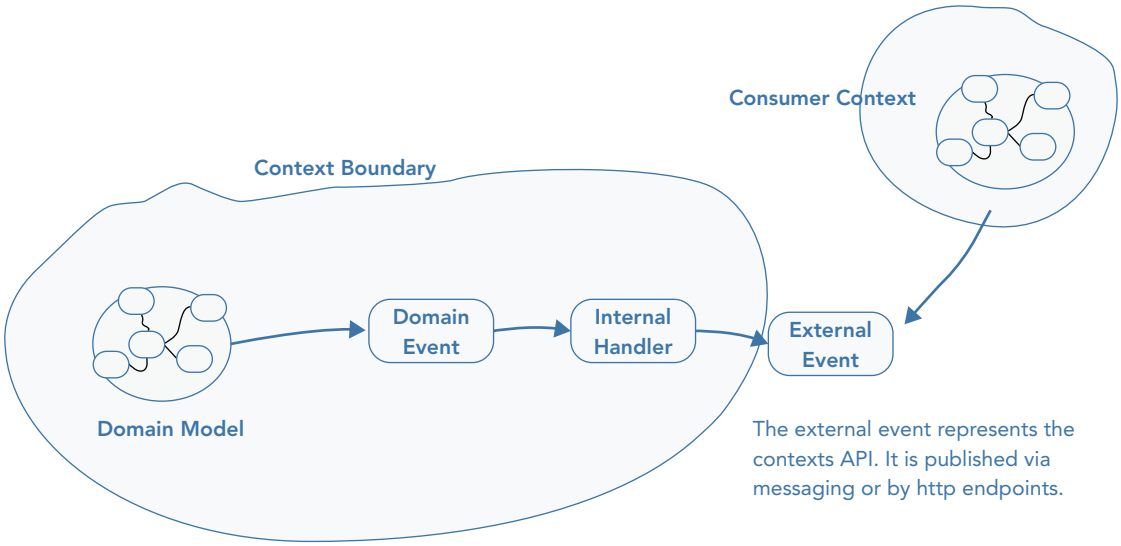


FIGURE 18-2: Flow of internal and external events in a typical business use case

With all of these differences in mind, it makes sense to put your events in different namespaces to accentuate those that are internal from those that are external.

EVENT HANDLING ACTIONS

Before seeing concrete examples of the domain events pattern it's important to have an understanding of the distinction between handlers in the domain and handlers in the application service layer. Even though they look the same, their responsibilities are significantly different. Domain event handlers invoke domain logic, such as invoking a domain service, whereas application service layer event handlers are more infrastructural in nature, carrying out tasks like sending e-mails and publishing events to other bounded contexts.

Invoke Domain Logic

Event handlers that exist within the domain model can handle events that occur there. These scenarios are modeling sequences of interaction that occur in the problem domain. For example, in the online takeaway store, an order validator validates an order, verifying that the customer is not blacklisted

after failing to pay for previous orders. The validator then raises events of its own to trigger the next part of the business process. It's common to see domain event handlers delegating to a domain service.

Invoke Application Logic

There is a benefit to having event handlers that live in the application service layer in addition to those that live in the domain. These event handlers tend to carry out infrastructural tasks like sending e-mails. Note that these handlers are not part of UL or the domain.

One important responsibility of application service layer handlers is that they trigger communication with external bounded contexts, using the techniques presented in Part II. They also manage communication with external services, like payment gateways.

DOMAIN EVENTS' IMPLEMENTATION PATTERNS

Domain events is a generic pattern that basically adds domain semantics to the publish-subscribe pattern. This gives you a lot of freedom to implement the solution. You'll see a variety of options in the following examples that range from synchronous use of native language constructs, to message bus-based alternatives.

Use the .Net Framework's Events Model

An easy way to get started with domain events is to rely on the native capabilities provided by the language or platform you are using, if applicable. In C# this means using the `event` keyword.

To define a domain event using the `event` keyword, you can create a public field on an entity named after the domain event. You also need to create a delegate that represents the contract of the event. Both of these details are shown in Listing 18-4.

LISTING 18-4: Defining a Domain Event with the Event Keyword in C#

```
public class OrderForDelivery
{
    ...

    public delegate void
        DeliveryGuaranteeFailedHandler(DeliveryGuaranteeFailed evnt);

    public event DeliveryGuaranteeFailedHandler DeliveryGuaranteeFailed;

    ...
}
```

In Listing 18-4 an event called `DeliveryGuaranteeFailed` is created using the `event` keyword. You can see that this event has a type of `DeliveryGuaranteeFailedHandler` signifying that all event handlers of the `DeliveryGauranteeFailed` event must have the same signature as the `DeliveryGuaranteeFailedHandler`. You can see the signature of the `DeliveryGuaranteeFailedHandler` on the line above where it is declared using the `delegate` keyword. Its signature is a void return type with a single parameter of type `DeliveryGuaranteeFailed`. You can see a handler with this signature being registered in Listing 18-5.

LISTING 18-5: Subscribing to a Native C# Event

```
public class ConfirmDeliveryOfOrder
{
    private IOrderRepository orderRepository;
    private IBus bus;

    public ConfirmDeliveryOfOrder(IOrderRepository orderRepository, IBus bus)
    {
        this.orderRepository = orderRepository;
        this.bus = bus;
    }

    public void Confirm(DateTime timeThatPizzaWasDelivered, Guid orderId)
    {
        var order = orderRepository.FindBy(orderId);
        order.DeliveryGuaranteeFailed += onDeliveryGuaranteeFailed;
        order.ConfirmReceipt(timeThatPizzaWasDelivered);
    }

    private void onDeliveryGuaranteeFailed(DeliveryGuaranteeFailed evnt)
    {
        bus.Send(new RefundDueToLateDelivery() { OrderId = evnt.Order.Id });
    }
}
```

`onDeliveryGuaranteeFailed` is a method with a void return type that takes a single parameter of type `DeliveryGuaranteeFailed`. Therefore it can successfully be registered as a handler for the `DeliveryGuaranteeFailed` event, as shown in Listing 18-5. It is registered using C#'s special syntax for registering event handlers: `+=`.

To raise C# events you invoke them like methods, as shown in Listing 18-6, where the `DeliveryGuaranteeFailed` event is raised. It's important to check if the event is null first, because if an event has no subscribers it will be null, causing a null reference exception to be thrown.

LISTING 18-6: Invoking a Native C# Event

```
public class OrderForDelivery
{
    ...

    public delegate void
        DeliveryGuaranteeFailedHandler(DeliveryGuaranteeFailed evnt);

    public event DeliveryGuaranteeFailedHandler DeliveryGuaranteeFailed;

    ...

    public void ConfirmReceipt(DateTime timeThatPizzaWasDelivered)
    {
```

continues

LISTING 18-6: *(continued)*

```
            if (Status != FoodDeliveryOrderSteps.Delivered)
            {
                TimeThatPizzaWasDelivered = timeThatPizzaWasDelivered;
                Status = FoodDeliveryOrderSteps.Delivered;
                if(DeliveryGuaranteeOffer.IsNotSatisfiedBy(
                   TimeOfOrderBeingPlaced, TimeThatPizzaWasDelivered))
                {
                    if (DeliveryGuaranteeFailed !=  null)
                        DeliveryGuaranteeFailed(new DeliveryGuaranteeFailed(this));
                }
            }
        }

        ...
    }
```

To trigger an asynchronous process you can create, and register, an event handler that publishes an asynchronous message, writes to a queue, or updates a database. You need to ensure that the action is enlisted as part of the current transaction, so that if the transaction rolls back, so will the action that triggers the asynchronous process. When using C# and the `event` keyword, it's likely you will want to configure the Distributed Transaction Coordinator (DTC).

> **WARNING** *When registering event handlers that carry out infrastructural concerns, like talking to databases, you should register them in the service layer to keep the domain model free from technical pollution. Alternatively, you could have a domain service contract that is implemented in the service layer (see Chapter 17, "Domain Services," for more details).*

As you can see from Listings 18-4 to 18-6, the up-front investment when using native events can be minimal by just using the `event` keyword. However, this approach has a cost; there is a tight coupling between publishers of an event and subscribers. You can see this in Listing 18-5, where the handler of an event must know which object is publishing the event. Alternative implementations of the domain events pattern have a looser coupling of publishers and subscribers, as shown in the next section.

Use an In-Memory Bus

To decouple publishers and subscribers you can implement the domain events pattern using an in-memory message bus. This gives you the option to easily make some aspects of the system asynchronous when they can benefit from it, assuming the message bus supports both synchronous and asynchronous transit. NServiceBus is one message bus that does provide both options.

> **NOTE** *The examples in this section are based on version 4.3.3 of NServiceBus, which is available to download and install from:* `https://github.com/Particular/NServiceBus/releases/download/4.3.3/Particular.NServiceBus-4.3.3.exe`. *You will also need to add a reference to NServiceBus in your project using the following command in the NuGet package manager console (making sure to substitute the name of your project)*:
>
> ```
> Install-Package NServiceBus.Host –ProjectName YourProjectNameHere
> -Version 4.3.3
> ```

With a message bus you again focus on events and handlers. But instead of wiring them up with each other, you publish events through the bus, and register handlers through the bus. With NServiceBus there is no need to actually register subscribers, though, because all event handlers implement the NServiceBus `IHandleMessages<T>` interface, where `T` is the type of event being handled, so NServiceBus can easily find event handlers automatically. All NServiceBus requires is to be told which classes are the events, as shown in Listing 18-7.

> **NOTE** *It is recommended that you configure NServiceBus in the service layer to keep the technical concerns out of the domain model. As long as the project with the configurations references the domain project, the conventions will detect the events.*

LISTING 18-7: Configuring NServiceBus to Automatically Register Subscribers

```
public class EndpointConfig : IConfigureThisEndpoint,
        AsA_Server, AsA_Publisher, IWantCustomInitialization
{
    public void Init()
    {
        Configure.With()
                 .DefiningEventsAs(t => t.Namespace != null
                      && t.Namespace.Contains("Events"));
    }

}
```

Listing 18-7 uses the convention that any class inside a namespace called `Events` will be considered an event by NServiceBus. Therefore, if the event in Listing 18-8 was published (notice its namespace contains `Events`), NServiceBus would invoke the handler shown in Listing 18-9.

LISTING 18-8: An NServiceBus Event

```
namespace OnlineTakeawayStore.NServiceBus.Model.Events
{
    public class DeliveryGuaranteeFailed
```

continues

LISTING 18-8: *(continued)*

```
    {
        public DeliveryGuaranteeFailed(OrderForDelivery order)
        {
            Order = order;
        }

        public OrderForDelivery Order { get; private set; }
    }
}
```

LISTING 18-9: An NSerivceBus Event Handler

```
// Automatically found by NServiceBus due to inheriting IHandleMessages<T>
public class RefundOnDeliveryGuaranteeFailureHandler :
    IHandleMessages<DeliveryGuaranteeFailed>
{
    // Injected by NServiceBus
    public IBus Bus { get; set; }

    public void Handle(DeliveryGuaranteeFailed message)
    {
        Bus.Send(new RefundDueToLateDelivery() { OrderId = message.Order.Id });
    }
}
```

Listing 18-10 then shows how this message is published using NServiceBus's in-memory bus. Notice how the subscriber does not know, or care, who the publisher of the event is and vice-versa.

LISTING 18-10: Publishing Events with the NSerivceBus In-memory Bus

```
public class OrderForDelivery
{
    ...

    private IBus Bus { get; set; }

    ...

    public void ConfirmReceipt(DateTime timeThatPizzaWasDelivered)
    {
        if (Status != FoodDeliveryOrderSteps.Delivered)
        {
            TimeThatPizzaWasDelivered = timeThatPizzaWasDelivered;
            Status = FoodDeliveryOrderSteps.Delivered;
            if (DeliveryGuaranteeOffer.IsNotSatisfiedBy(
                TimeOfOrderBeingPlaced, TimeThatPizzaWasDelivered))
            {
                Bus.InMemory.Raise(new DeliveryGuaranteeFailed(this));
            }
```

```
            }
        }

        ...
    }
```

One of the tricky details with using an in-memory bus is passing it around the domain model. Ideally you don't want technical concerns, like a message bus, cluttering the domain model at all. On the other hand, you do need some way of publishing events. For some teams, this is a situation where it is an acceptable trade-off to allow technical concerns into the domain because it allows the model to be more expressive of domain concepts overall.

NServiceBus's default mode of operation is sending asynchronous messages, so if you decide that you need to fire some domain events asynchronously, the transition is relatively easy. You add a new event handler that subscribes to an in-memory event and publishes asynchronous events. Or, you could just publish the initial event asynchronously using `Bus.Publish()` instead of `Bus.InMemory.Raise()`, in which case all subscribers would execute asynchronously on a different thread.

It's important to be aware of transactional requirements when deciding to publish events asynchronously. If you have two event handlers that need to atomically succeed or fail, then you will need to ensure they execute synchronously. Otherwise you will be open to bugs arising from inconsistency.

> **NOTE** *If you don't want to couple your domain model to a particular messaging technology, you could encapsulate it behind an interface. Shortly you will see the static* `DomainEvents` *class, which is one option you could use.*

Udi Dahan's Static DomainEvents Class

Traditional DDD advice has been to remove all infrastructural concerns from the domain model. Traditional object-oriented guidelines warn that you should avoid relying on static methods because of coupling. You might be surprised to hear then that another popular version of the domain events pattern relies on a static class called `DomainEvents` that deals with infrastructural concerns and lives in the domain model. Listing 18-11 shows the implementation of the static `DomainEvents` class, which builds on the work of DDD expert Udi Dahan (`http://www.udidahan.com/2008/08/25/domain-events-take-2/`).

LISTING 18-11: The Static DomainEvents Class

```
public static class DomainEvents
{
    [ThreadStatic]
    private static List<Delegate> _actions;
    private static List<Delegate> Actions
```

continues

LISTING 18-11: *(continued)*

```
    {
        get
        {
            if (_actions == null)
            {
                _actions = new List<Delegate>();
            }

            return _actions;
        }
    }

    public static IDisposable Register<T>(Action<T> callback)
    {
        Actions.Add(callback);

        return new DomainEventRegistrationRemover(
            () => Actions.Remove(callback)
        );
    }

    public static void Raise<T>(T eventArgs)
    {
        foreach (Delegate action in Actions)
        {
            Action<T> typedAction = action as Action<T>;
            if (typedAction != null)
            {
                typedAction(eventArgs);
            }
        }
    }

    private sealed class DomainEventRegistrationRemover : IDisposable
    {
        private readonly Action _callOnDispose;

        public DomainEventRegistrationRemover(Action toCall)
        {
            _callOnDispose = toCall;
        }

        public void Dispose()
        {
            _callOnDispose();
        }
    }
```

`DomainEvents` has two important methods: `Register<T>` will register a callback that is executed when an event of type `T` is raised by the other important method: `Raise<T>`.

Another detail to discern in Listing 18-11 is the use of the private `DomainEventsRegistrationRemover` class. You can see that `Raise<T>` uses an instance of this class to automatically unregister event handlers once they have executed a single time.

Listing 18-12 shows how you can register an event handler with `DomainEvents` while Listing 18-13 then shows how you can publish an event.

LISTING 18-12: Registering an Event Handler with DomainEvents

```
public void Confirm(DateTime timeThatPizzaWasDelivered, Guid orderId)
{

    using(DomainEvents.Register<DeliveryGuaranteeFailed>(onDeliveryFailure))
    {
        var order = orderRepository.FindBy(orderId);
        order.ConfirmReceipt(timeThatPizzaWasDelivered);
    }
}

private void onDeliveryFailure(DeliveryGuaranteeFailed evnt)
{
    // handle internal event and publish external event
    bus.Send(new RefundDueToLateDelivery() { OrderId = evnt.Order.Id });
}
```

LISTING 18-13: Publishing an Event with DomainEvents

```
public class OrderForDelivery
{
    ...

    public void ConfirmReceipt(DateTime timeThatPizzaWasDelivered)
    {
        if (Status != FoodDeliveryOrderSteps.Delivered)
        {
            TimeThatPizzaWasDelivered = timeThatPizzaWasDelivered;
            Status = FoodDeliveryOrderSteps.Delivered;
            if (DeliveryGuaranteeOffer.IsNotSatisfiedBy(
                TimeOfOrderBeingPlaced, TimeThatPizzaWasDelivered)
            )
            {
                DomainEvents.Raise(new DeliveryGuaranteeFailed(this));
            }
        }
    }
}
```

Handling Threading Issues

One of the big challenges involved with using the static `DomainEvents` class is managing threading issues. This is because any handler registered in one thread is executed when an event of the type

it handles is published by any other thread. As alluded to, this is a result of `DomainEvents` being static; every thread uses the same instance.

A technique that can help to avoid threading issues is to apply the `ThreadStatic` attribute to the collection of handlers and collection of callbacks on the `DomainEvents` class. Using this attribute, each thread gets its own collection, meaning that handlers registered on one thread are not visible to another thread. Listing 18-14 shows how to use the `ThreadStatic` attribute with domain events.

LISTING 18-14: Applying ThreadStatic to the Handlers and Callbacks of DomainEvents

```
public static class DomainEvents
{
    [ThreadStatic] // each thread gets its own callbacks
    private static List<Delegate> _actions;

    ...
}
```

However, there is cause for concern when using this approach in ASP.NET web applications. Scott Hanselman explains on his blog (`http://www.hanselman.com/blog/ATaleOfTwoTechniquesTheThreadStaticAttributeAndSystemWebHttpContextCurrentItems.aspx`) that `ThreadStatic` should, in fact, never be used in ASP.NET applications.

Avoid a Static Class by Using Method Injection

If you're not prepared to couple your domain to a static event publisher, as many DDD practitioners aren't, you can instead supply one as an argument to methods in your domain model. This pattern is also a good consideration when you don't want the previously mentioned problems associated with the `ThreadStatic` attribute. Listing 18-15 shows an instance of an event dispatcher being passed into a domain method and invoked.

LISTING 18-15: Using Non-static DomainEvents Class with Method Injection

```
public void ConfirmReceipt(DateTime timeThatPizzaWasDelivered,
    IEventDispatcher dispatcher)
{
    if (Status != FoodDeliveryOrderSteps.Delivered)
    {
        TimeThatPizzaWasDelivered = timeThatPizzaWasDelivered;
        Status = FoodDeliveryOrderSteps.Delivered;
        if (DeliveryGuaranteeOffer.IsNotSatisfiedBy(
            TimeOfOrderBeingPlaced, TimeThatPizzaWasDelivered)
        )
        {
            dispatcher.Raise(new DeliveryGuaranteeFailed(this));
        }
    }
}
```

Return Domain Events

Another take on the domain events pattern is to decouple the publishing and handling of events, so that side effects are isolated. This approach is implemented by storing a collection of events on the aggregate root and publishing them once the aggregate root has completed its task. Significantly, a dispatcher is called from the service layer to publish the events, keeping the technical concern out of the domain model.

Listing 18-16 shows an alternative version of the `OrderForDelivery` entity that keeps a collection of events (its `RecordedEvents` property). Each time the entity is loaded from persistence, this collection will be empty. It only contains events that have occurred inside the current transaction.

LISTING 18-16: Storing a Collection of Events on an Entity

```
public class OrderForDelivery
{
    ...

    public List<Object> RecordedEvents { get; private set; }

    public void ConfirmReceipt(DateTime timeThatPizzaWasDelivered)
    {
        if (Status != FoodDeliveryOrderSteps.Delivered)
        {
            TimeThatPizzaWasDelivered = timeThatPizzaWasDelivered;
            Status = FoodDeliveryOrderSteps.Delivered;
            if (DeliveryGuaranteeOffer.IsNotSatisfiedBy(
                TimeOfOrderBeingPlaced, TimeThatPizzaWasDelivered))
            {
                RecordedEvents.Add(new DeliveryGuaranteeFailed(this));
            }
        }
    }

    ...
}
```

When the `ConfirmReceipt` behavior is triggered, instead of publishing an event (as per Listing 18-16), an event is added to the `OrderForDelivery` entity's `RecordedEvents` collection, as shown in Listing 18-16. Note how the collection is public so that it can be accessed outside of the domain model.

Any side effects that arise from handling the event will not immediately occur, as is the case when events are simply published. This is seen as desirable by some developers because the side effects aren't interleaved with the execution of the current method.

When the entity has finished carrying out its logic, the thread of execution will return to the application service that is managing the business use case. It is at this point that the events from the entity can be fed into a dispatcher, as shown in Listing 18-17.

LISTING 18-17: Dispatching Events in the Service Layer

```
public class ConfirmDeliveryOfOrder
{
    private IOrderRepository orderRepository;
    private IEventDispatcher dispatcher;

    public ConfirmDeliveryOfOrder(IOrderRepository orderRepository,
        IEventDispatcher dispatcher)
    {
        this.orderRepository = orderRepository;
        this.dispatcher = dispatcher;
    }

    public void Confirm(DateTime timeThatPizzaWasDelivered, Guid orderId)
    {
        var order = orderRepository.FindBy(orderId);
        order.ConfirmReceipt(timeThatPizzaWasDelivered);

        foreach (var evnt in order.RecordedEvents)
        {
            dispatcher.Dispatch(evnt);
        }
    }
}
```

It is also possible to use method injection if you do not want to pass an event dispatcher into the constructor of your domain objects. Listing 18-14 shows how an event dispatcher is passed into `ConfirmReceipt()`, where domain events are recorded on the dispatcher rather than the entity. Note that the dispatcher will not dispatch the events immediately. Instead, it will record them and will dispatch them once instructed to from the application service layer, as previously shown in Listing 18-18.

LISTING 18-18: Recording Events on a Dispatcher

```
public class OrderForDelivery
{
    ...

    public void ConfirmReceipt(DateTime timeThatPizzaWasDelivered,
        IEventDispatcher dispatcher)
    {
        if (Status != FoodDeliveryOrderSteps.Delivered)
        {
            TimeThatPizzaWasDelivered = timeThatPizzaWasDelivered;
            Status = FoodDeliveryOrderSteps.Delivered;
            if (DeliveryGuaranteeOffer.IsNotSatisfiedBy(
                TimeOfOrderBeingPlaced, TimeThatPizzaWasDelivered))
            {
                Dispatcher.Record(new DeliveryGuaranteeFailed(this));
            }
```

```
            }
        }

        ...
    }
```

Clearly, passing a dispatcher into methods is going to pollute the signature of your domain model, so you'll want to be extra careful about using this pattern. You may want to store events on an entity in the majority of use cases, and reserve passing around a dispatcher for when there is a clear benefit.

To implement a dispatcher you can copy most of the details from Listing 18-18, except you won't need to make the class static. Or you can use an IoC container, as the next example demonstrates.

> **NOTE** *Storing events on the aggregate root is a fundamental aspect of creating event-sourced domain models. So if you are considering event sourcing, this approach is highly relevant. You can find out more in Chapter 22.*

Use an IoC Container as an Event Dispatcher

If you are using an IoC container you can use it as an event dispatcher. This can be a good choice when you are already using an IoC container or don't want to build your own event dispatcher. It can also be a good choice when you want advanced features provided by IoC containers, including lifecycle management, or convention-based handler registrations. Convention-based handler registration is shown in Listing 18-19 where a StructureMap convention scans all assemblies in a project and automatically registers each event handler with its associated event. For this pattern, though, you will need to create an interface to represent event handlers. Listing 18-19 shows an `IHandles<T>` interface being used.

LISTING 18-19: Automatic Handler Registration with StructureMap

```
return new Container(x =>
{
    x.Scan(y =>
    {
        y. AssembliesFromApplicationBaseDirectory();
        y.AddAllTypesOf(typeof(IHandles<>));
        y.WithDefaultConventions();
    });
});
```

You can see in Listing 18-20 how once all your event handlers are registered, you can use a container from within an application service to dispatch all events recorded during the current transaction.

LISTING 18-20: Using an IoC to Dispatch Events

```
public void Confirm(DateTime timeThatPizzaWasDelivered, Guid orderId)
{
    var order = orderRepository.FindBy(orderId);
    order.ConfirmReceipt(timeThatPizzaWasDelivered);

    foreach(var handler in Container.ResolveAll<IHandles<T>>())
        handler.Handle(order.RecordedEvents);
}
```

TESTING DOMAIN EVENTS

Testing code that uses the domain events pattern is no more complicated than traditional object-oriented code. In places it can even be simplified because collaborators do not need to be mocked. The testing patterns are a little different, though, so this section provides a short guide with examples.

Unit Testing

Many of your unit tests want to verify that an entity or domain service raised an event to signify a change occurring. In the online takeaway store scenario, this could be demonstrated with the previously discussed `DeliveryGuaranteeFailed` event. Listing 18-21 shows a unit test that validates the event is raised using the static `DomainEvents` version of the pattern.

LISTING 18-21: Unit Testing an Entity That Raises Domain Events

```
[TestClass]
public class Delivery_guarantee_events_are_raised_on_guarantee_offer_failure
{
    ...

    public bool eventWasRaised = false;

    [TestInitialize]
    public void When_confirming_an_order_that_is_late()
    {
        offer.Stub(x =>
            x.IsNotSatisfiedBy(timeOrderWasPlaced, timePizzaDelivered)
        ).Return(true);

        var order = new OrderForDelivery(id, customerId,
            restaurantId, menuItemIds, timeOrderWasPlaced, offer
        );

        using (DomainEvents.Register<DeliveryGuaranteeFailed>(setTestFlag))
        {
            order.ConfirmReceipt(timePizzaDelivered);
        }
    }

    private void setTestFlag(DeliveryGuaranteeFailed obj)
```

```
        {
            eventWasRaised = true;
        }

        [TestMethod]
        public void A_delivery_guarantee_failed_event_will_be_raised()
        {
            Assert.IsTrue(eventWasRaised);
        }
    }
```

Having a static `DomainEvents` class doesn't actually make testing difficult, as Listing 18-21 illustrates. When using the domain events pattern, you don't want to mock the `DomainEvents` class, meaning the tight coupling is not really a problem. This means that in a unit test, you can verify the appropriate event was raised by registering a callback for it that sets a flag. In Listing 18-21, this flag is the `eventWasRaised` variable. If `eventWasRaised` is `true` at the end of the test, it's clear that the event was raised.

In a similar fashion, when recording domain events, you again invoke the desired domain behavior, but instead verify that the event was recorded as opposed to being published. This is shown in Listing 18-22.

LISTING 18-22: Unit Testing an Aggregate That Returns Domain Events

```
[TestClass]
public class Delivery_guarantee_events_are_recorded_on_guarantee_offer_failure
{
    ...

    [TestInitialize]
    public void When_confirming_an_order_that_is_late()
    {
        offer.Stub(x =>
            x.IsNotSatisfiedBy(timeOrderWasPlaced, timePizzaDelivered)
        ).Return(true);

        order = new OrderForDelivery(id, customerId, restaurantId,
            menuItemIds, timeOrderWasPlaced, offer
        );

        order.ConfirmReceipt(timePizzaDelivered);
    }

    [TestMethod]
    public void A_delivery_guarantee_failed_event_will_be_recorded()
    {
        var wasRecorded = order
            .RecordedEvents
            .OfType<DeliveryGuaranteeFailed>()
            .Count() == 1;

        Assert.IsTrue(wasRecorded);
    }
```

Application Service Layer Testing

Integration testing at the application service-layer level is also possible when using domain events. In fact, your tests probably look very similar to when you're not using domain events. This is because events and event handlers are just implementation details, as the example in Listing 18-23 clarifies.

LISTING 18-23: Testing Application Services When Using Domain Events

```
[TestClass]
public class Delivery_guarantee_failed
{
    IBus bus = MockRepository.GenerateStub<IBus>();
    // when testing at service layer may want to use concrete repository
    IOrderRepository repo = MockRepository.GenerateStub<IOrderRepository>();
    Guid orderId = Guid.NewGuid();
    Guid customerId = Guid.NewGuid();
    Guid restaurantId = Guid.NewGuid();
    List<int> itemIds = new List<int> { 123, 456, 789 };
    DateTime timeOrderWasPlaced = new DateTime(2015, 03, 01, 20, 15, 0);

    [TestInitialize]
    public void If_an_order_is_not_delivered_within_the_agreed_upon_timeframe()
    {
        var offer = new ThirtyMinuteDeliveryGuaranteeOffer();

        // took longer than 30 minutes - failing the delivery guarantee
        var timeOrderWasReceived = timeOrderWasPlaced.AddMinutes(31);

        var order = new OrderForDelivery(
            orderId, customerId, restaurantId, itemIds, timeOrderWasPlaced, offer
        );

        repo.Stub(r => r.FindBy(orderId)).Return(order);

        var service = new ConfirmDeliveryOfOrder(repo, bus);
        service.Confirm(timeOrderWasReceived, orderId);        }

    [TestMethod]
    public void An_external_refund_due_to_late_delivery_instruction_will_be_
published()
    {
        // get first message published during execution of this use case
        var message = bus.GetArgumentsForCallsMadeOn(
            b => b.Send(new RefundDueToLateDelivery()),
            x => x.IgnoreArguments()
        )[0][0];

        var refund = message as RefundDueToLateDelivery;

        Assert.IsNotNull(refund);
        Assert.AreEqual(refund.OrderId, orderId);
    }
}
```

Nowhere in Listing 18-23 is there a hint of the domain events pattern. Test data and services are set up, the `ConfirmDeliveryOfOrder` application service is instantiated and invoked, and finally the assertion is made that NServiceBus published an external `RefundDueToLateDelivery` command to be handled by another bounded context. This test would be the same whether or not the domain events pattern was used as the implementation for the domain model. This shows that testing at the application service layer adds no additional complexity when using domain events. Comfortingly, it also means that you can refactor existing domain models to use the domain events pattern without having to change any of your application service layer tests. And they can also reassure you that you haven't broken anything.

> **NOTE** *Application services and their responsibilities are covered in more detail in Chapter 25, "Commands: Application Service Patterns for Processing Business Use Cases," including testing strategies and techniques.*

THE SALIENT POINTS

- Domain events are significant occurrences in the real-world problem domain; they are part of the ubiquitous language (UL).
- Domain events is also a design pattern that makes domain events more explicit in code.
- Domain events is akin to publish-subscribe, where events are raised and event handlers handle them.
- Unlike messaging between bounded contexts (Chapters 11 through 13), domain events are applied inside a single domain model and are usually synchronous.
- Using domain events can make it easier to transition certain operations or use cases into asynchronous processes.
- Event handlers can be classes or anonymous callbacks/lambdas.
- Some handlers live in the domain, and some live in the service layer.
- Handlers in the domain carry out domain logic.
- Handlers in the service layer normally take some of the responsibilities otherwise handled by application services.
- There are multiple versions of the domain events pattern.
- One approach is to use native language constructs, like C#'s `event` keyword.
- C# events have a tight coupling between publishers and subscribers. For looser coupling you can use an in-memory bus, such as NServiceBus.
- An in-memory bus causes immediate execution of events and their side effects. If this is undesirable, you can store events and dispatch them later with a dispatcher.

- Another version relies on a static `DomainEvents` class, which is similar to an in-memory message bus but much more compact.
- Using domain events does not mean you have to use event sourcing, although a lot of developers find the combination highly effective.
- Ultimately, the pattern is about trade-offs—an extra layer of indirection that distributes code across more files but has a fully encapsulated domain model that accentuates real-world domain events.

19

Aggregates

WHAT'S IN THIS CHAPTER?

- How to deal with complex entity and value object relationships
- Why the aggregate is the most powerful tactical pattern
- How to size and structure aggregates
- The role of the aggregate root
- Understanding aggregate consistency guidelines

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 19 download and individually named according to the names throughout the chapter.

A domain model's structure is composed of entities and value objects that represent concepts in the problem domain. It is, however, the number and type of relationships between domain objects that can cause complexity and confusion when implementing a domain model in code. Associations that do not support behavior and exist only to better reflect reality add unnecessary complexity to a domain model. Associations that can be traversed in more than one direction also increase complexity. This is why designing relationships between domain objects is equally as important as designing the domain objects themselves.

Even when all associations in a model are justified, a large model still has technical challenges, making it difficult to choose transactional and consistency boundaries that both reflect the problem domain and perform well.

This chapter covers best practices for keeping the relationships between domain objects simple and aligned with domain invariants. It also introduces the Domain-Driven Design

(DDD) concept of an aggregate—a consistency boundary that decomposes large models into smaller clusters of domain objects that are technically easier to manage. Both of these topics focus on helping you manage complexity within your domain models.

Aggregates are an extremely important pattern at your disposal when designing a domain model. They help you manage technical complexity, and they add a higher level of abstraction that can simplify talking and reasoning about the domain model.

MANAGING COMPLEX OBJECT GRAPHS

During the early stages of designing a model, novice DDD practitioners tend to focus on the entities and value objects of the domain, paying little attention to the relationships between the domain objects. They often default to associations that mirror real life or, even worse, the underlying data model. The result of not justifying each association and ensuring it matches a domain invariant is a domain model that hides important concepts, confuses both domain experts and developers, and is technically challenging to implement.

Dependencies can become overwhelming, especially when you have many-to-many relationships. It's important to remind yourself that the domain model is not the same as the data model. And, perhaps just as important, the purpose of the domain model is to support invariants and uses cases rather than user interfaces.

Avoiding complex object graphs doesn't have to be difficult. You can easily reduce the number of associations between entities and value objects by only allowing relationships in a single direction. To decrease the number of relationships, you can justify their inclusion. If the relationship is not required (does not work to fulfill an invariant), don't implement it. Basically, don't model real life. The relationship may exist in the problem domain, but it may not provide a benefit by existing in your code.

To summarize, modeling domain object associations based on the needs of use cases and not real life can ensure a simpler domain model and a more performant system. Associations between domain objects are there to support invariants, not user interface concerns. Simplify associations in object graphs. Model relationships from the point of view of a single traversal direction. Simplify the model, and you will make it easier to implement and maintain in code.

Favoring a Single Traversal Direction

A model that is built to reflect reality will contain many bidirectional object relationships. This means that two objects contain a reference to each other. Take the example of a procurement system, as shown in Figure 19-1. You can traverse between `Supplier` and `PurchaseOrders` in both directions. Similarly, you can traverse from a `Supplier` to all the `Products` that they can supply and back from a `Product` to the collection of `Suppliers` that it is available from. You also can traverse from a `PurchaseOrder` to the list of `Products` that it contains and from a `Product` to the list of purchase orders that have been raised against it. ORMs have made it too easy to create these types of bidirectional relationships in code, which can lead to deep object graphs and degraded runtime performance.

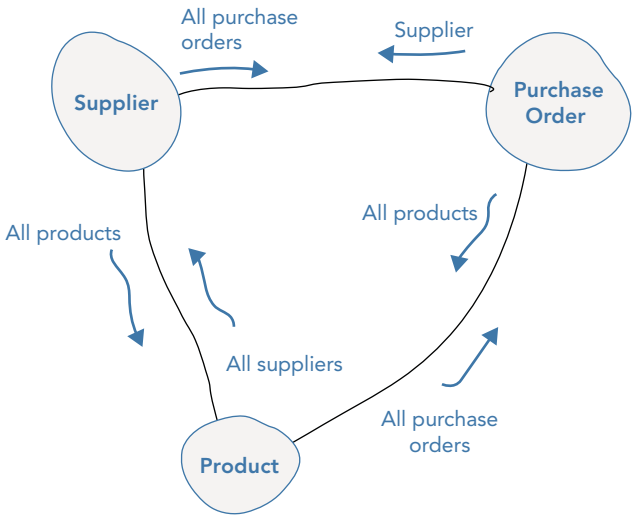


FIGURE 19-1: Complexity-causing bidirectional relationships.

> **NOTE** *Sometimes a two-way relationship can seem necessary to produce reports. However, reports can often be considered a special concern with special implementation patterns. Chapter 26, "Queries: Domain Reporting," contains relevant examples.*

LISTING 19-1: Relationships Implemented As Object References

```
public class Supplier
{
    ...

    public IList<PurchaseOrder> PurchaseOrders { get; private set; }
    public IList<Product> Products { get; private set; }

    ...
}

public class PurchaseOrder
{
    ...

    public Supplier Supplier {get; private set;}
```

continues

LISTING 19-1 *(continued)*

```
        public IList <Product> Products {get; private set;}

        ...
    }
    public class Product
    {
        ...

        public IList<Supplier> Suppliers {get; private set;}
        public IList<PurchaseOrder> PurchaseOrders {get; private set;}

        ...
    }
```

Is it really necessary to load all the `Products` a `Supplier` has when you only want to make a change to its contact details? Does it matter if a `Product` does not have a navigable list of `Suppliers`? As you can see, a bidirectional relationship adds technical complexity and obscures domain concepts. An especially important domain concept that it hides is the owner of the relationship. To simplify the relationship, you can constrain it to a single traversal direction, as shown in Figure 19-2.

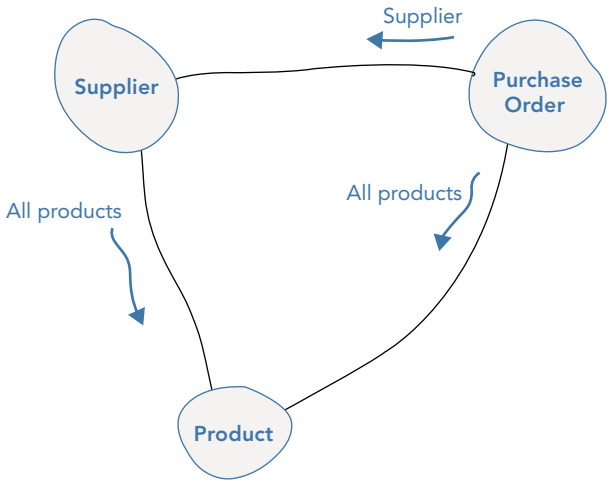


FIGURE 19-2: Constraining a bidirectional association.

By selecting a uni-directional relationship between the entities the domain model has been simplified. Essentially, a bidirectional relationship has been converted into a unidirectional relationship. This is a fundamental pattern for reducing complexity that you can be liberal with.

When defining object relationships, it is good practice to ensure that you explicitly ask: What is the behavior that an association is fulfilling, and who needs the relationship to function? This will help you avoid creating unnecessary bidirectional relationships. The procurement scenario is a simple example, but in a larger domain model with many bidirectional associations, things can quickly become extremely complicated, especially when it comes to persistence and retrieval.

Qualifying Associations

If you are implementing associations as object references to support domain invariants and those associations are one- or many-to-many, you should qualify the associations to reduce the number of objects that need to be hydrated. Consider Figure 19-3. The domain object `Contract` represents

a mobile phone contract, and `Calls` represents all calls made on that contract. `Calls` is required to understand how many free minutes are available for the current period so that a customer can be billed correctly. Customers on the twenty-third month of a 24-month contract will have many hundreds of call data. Loading all these `Calls` reduces the performance. However, this is inefficient and unnecessary if all the invariant requires is the calls for the current period to compute any remaining free minutes.

To qualify the association, only the calls for the current period can be retrieved, as highlighted in Figure 19-4.

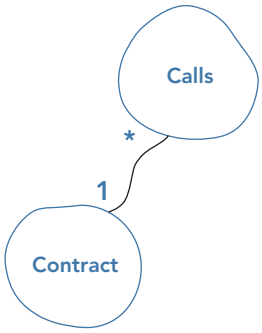


FIGURE 19-3: Qualifying associations.

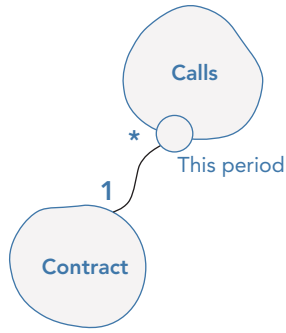


FIGURE 19-4: Qualifying associations with filter criteria from the Ubiquitous Language (UL).

To optimize performance, you can apply this qualification at the data access/database level using the following code snippet.

LISTING 19-2: Applying a Relationship Qualification Using a Repository

```
public class ContractRepository
{
    public Contract GetBy(ContractId id, Period period)
    {
        ....
    }
}
```

In general, the greater the number of items in an associated collection—the *cardinality*—the more complex the technical implementation will become. Therefore, aim for lower cardinality by adding constraints to collections. It's best to be explicit when it comes to collections; don't fall into the trap of simply re-creating the data model in code.

Preferring IDs Over Object References

As repeated throughout this book, the primary purpose of your domain model is to model the invariants of your system to support business use cases. Therefore, relationships between domain objects should exist only for behavioral needs. Relationships that do not support behaviors can increase complexity in the implementation of the domain model. Object references are the classic

example of adding unnecessary, complexity-increasing relationships to a domain model.

Understandably, many developers find the natural way to model a relationship in code as an object reference. For example, in real life a customer has many orders, but in the solution space of the application, there may be no invariant that requires a `Customer` object to hold a collection of all `Orders` belonging to that customer. Modeling this relationship using object references, as shown in Figure 19-5, adds unnecessary complexity.

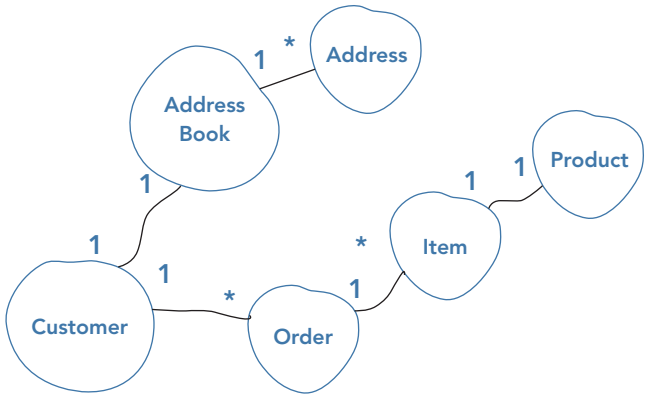


FIGURE 19-5: Modeling relationships with object references increases complexity.

Because every association is implemented as an object reference, to place an order, you could use the following code to traverse the object graph.

LISTING 19-3: Placing an Order with Object Associations Implemented As Object References

```
public class OrderApplicationService
{
    public void PlaceOrder(Guid customerId, IEnumerable<Product> products)
    {
        var customer = _customerRepository.FindBy(customerId);

        customer.AddOrder(products);
    }
}
```

This code could be slow because a large object graph (the `Customer` and all its existing `Orders`) needs to be loaded for the basic use case of placing an order. All that really needs to happen is for the retrieved customer to have an order added to its collection with the default delivery address of the customer set as the order dispatch address. But many of the objects loaded are not involved in the use case. These problems arise because the relationship is modeled as an object reference, requiring all objects in the object graph to be loaded from persistence collectively. You might argue that lazy loading makes the problems go away, but lazy loading can further complicate the model; also, it communicates little of how the domain objects are used to fulfill business cases.

The alternative method to implement associations is storing the ID of the object and using a repository within an application service to fetch domain objects that are required for a use case. By using the repository, you can reduce the object references in the model and therefore the complexity. Figure 19-6 shows the clearer object model without the unnecessary object references.

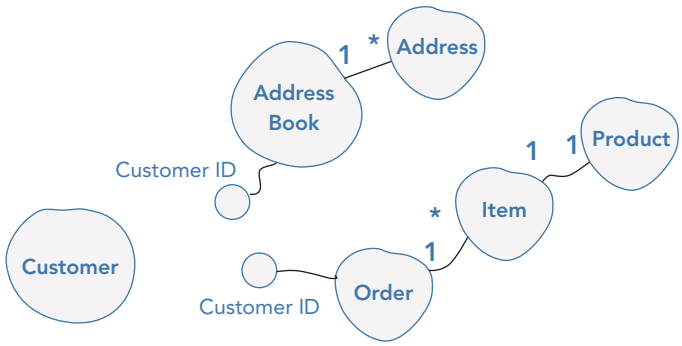


FIGURE 19-6: Simplified relationships using IDs instead of object references.

By preferring IDs and repositories, you need to carry out the coordination in an application service, as shown in Listing 19-4. In this case, the call to `_addressBookRepository.FindBy()` is the alternative to having an object reference on the `Customer` object itself.

LISTING 19-4: Placing an Order with Object Associations Implemented As IDs Only

```
public class OrderApplicationService
{
    ...
    public void PlaceOrder(Guid customerId, IEnumerable<Product> products)
    {
        var customer = _customerRepository.FindBy(customerId);

        var addressBook = _addressBookRepository.FindBy(customerId);

        var order = customer.CreateOrderFor(products);

        order.DispatchTo(addressBook.Default);

        _orderRepository.Add(order);
    }
```

You can ascertain if an object reference is necessary by asking the question: Is the association supporting a domain invariant for a specific use case? In the example of customers and orders, a `Customer` has `Orders`, but an `Order` only needs a `Customer`'s ID to meet invariants; it doesn't require a navigable reference. Try not to carelessly model real life or add a traversal object reference to match the data model; add object references only when they are necessary to meet the requirements of an invariant. In all other cases, favor IDs and repositories to reduce the coupling in your domain model.

Figure 19-7 visualizes the common structure of a domain model that needlessly models real life and contains a proliferation of object references. Alternatively, Figure 19-8 shows a partitioned model with fewer associations between objects, favoring IDs instead of object references. If you follow the advice in this section, Figure 19-8 is what you should be aiming for if you want an explicit domain model with reduced technical complexity. In the remainder of this chapter, you will see how DDD aggregates help you achieve it.

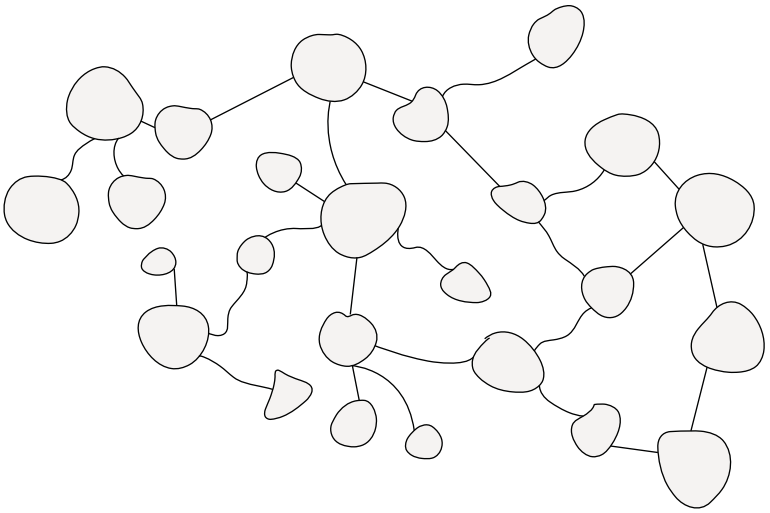


FIGURE 19-7: Complex domain model with an abundance of unnecessary associations.

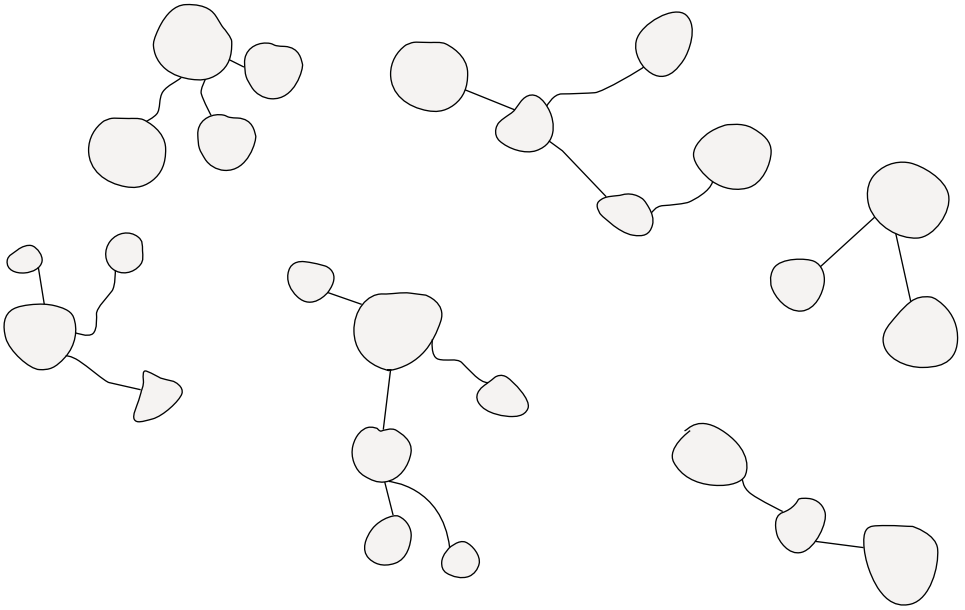


FIGURE 19-8: Clearer domain model based only on essential associations.

AGGREGATES

Reducing and constraining relationships between domain objects simplifies the technical implementation and reflects a deeper insight into the domain. This is highly desirable for making the code easier to manage and concepts easier to communicate, but there is still a need to judiciously

group objects that are used together so that a system performs well and is reliable. Aggregates help you achieve all these goals by guiding you into cohesively grouping objects around domain invariants while also acting as a consistency and concurrency boundary.

Aggregates are the most powerful of all tactical patterns, but they are one of the most difficult to get right. There are many guidelines and principles you can lean on to help with the construction of effective aggregates, but often developers focus only on the implementation of these rules and miss the true purpose and use of an aggregate, which is to act as a consistency boundary.

Design Around Domain Invariants

Domain invariants are statements or rules that must always be adhered to. If domain invariants are ever broken, you have failed to accurately model your domain. A basic example of an invariant is that winning auction bids must always be placed before the auction ends. If a winning bid is placed after an auction ends, the domain is in an invalid state because an invariant has been broken and the domain model has failed to correctly apply domain rules.

Using invariants as a design heuristic for aggregates makes sense because invariants often involve multiple domain objects. When all domain objects involved in an invariant reside within the same aggregate, it is easier to compare and coordinate them to ensure that their collective states do not violate a domain invariant.

In the winning bid scenario, both the auction and the winning bid would be modeled as different objects. If both of them resided within a different aggregate, no single aggregate would have a reference to both objects and thus be able to ensure that at no point was the invariant broken, even temporarily. This is bad because invariants should never be broken.

> **NOTE** *You met invariants back in Chapter 15,"Value Objects." Although closely related, the invariants in that chapter applied to a single entity, whereas those discussed in this chapter often apply to a combination of domain objects.*

Higher Level of Domain Abstraction

By grouping related domain objects, you can refer to them collectively as a single concept—a desirable characteristic that lets you communicate and reason more efficiently. Aggregates afford these benefits of abstraction to your domain model by allowing you to refer to a collection of domain objects as a single concept. Instead of an order and order lines, you can refer to them collectively as an order, for example.

Consistency Boundaries

To ensure a system is usable and reliable, there is a strong need to make good choices about which data should be consistent and where transactional boundaries should lie. When applying DDD,

these choices arise from grouping objects that are involved in the same business use case(s). These cohesive groups of domain objects are aggregates.

Transactional Consistency Internally

One option for consistency is to have a single aggregate by wrapping the entire domain model in a single transactional boundary. The problem with this is that in a collaborative domain when many changes are being performed there is the potential for a conflict for changes that are completely unrelated. The problem would likely manifest as blocking issues at the database level or failed updates (due to pessimistic concurrency).

Figure 19-9 demonstrates problems that can arise through choosing suboptimal transactional boundaries that span too many objects. User A wants to add an address to a customer record, whereas user B wants to change the state of the same customer's order. There are no invariants that state while an order is being updated the personal details of a customer cannot change, but in this scenario if both updates are made at the same time, one of the user's changes will be blocked or rejected.

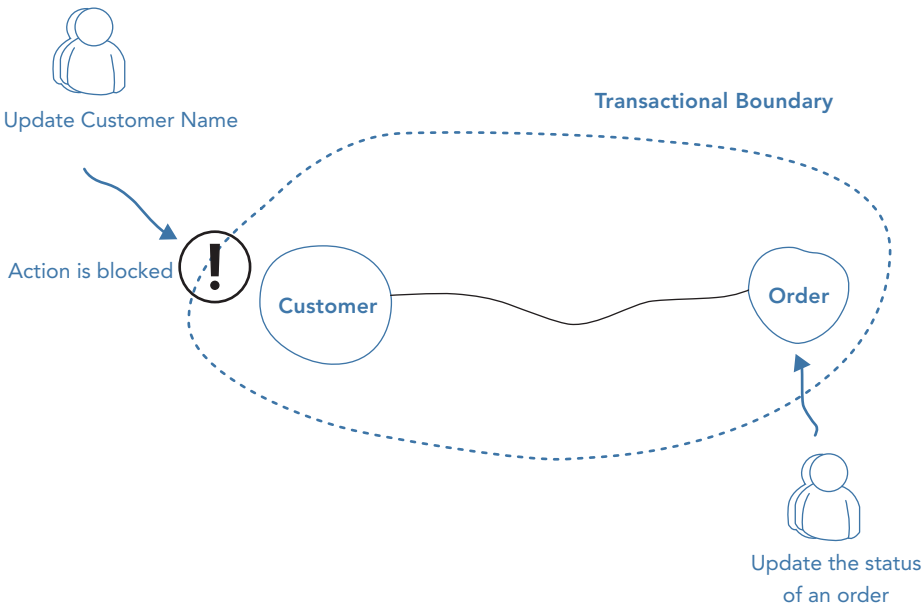


FIGURE 19-9: Locking caused by large transactional boundary.

You might be tempted to assume that having no transactional boundaries would solve all the problems. That would be dangerous, though, because then consistency problems can arise that cause domain-invariant violations. Figure 19-10 shows how two different users are updating the same order; one is applying a discount code to the order, while another is modifying an order line. Because there is no concurrency check and the change is not transactional, both changes will be

made, resulting in a broken invariant. In this case, the order could have a massive discount applied that is no longer applicable.

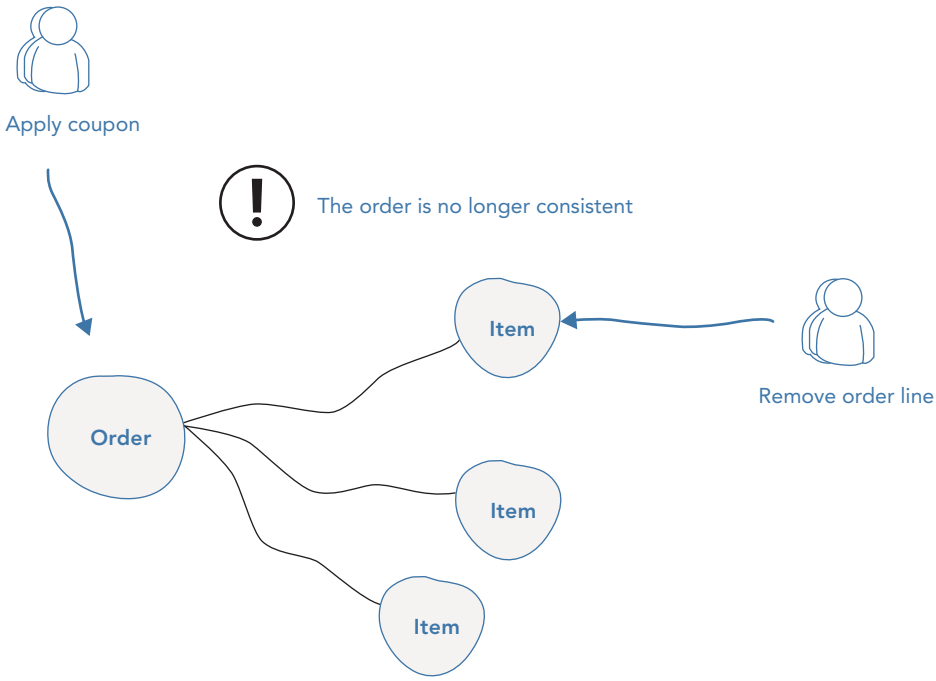


FIGURE 19-10: Inconsistent data arising from lack of transactional boundaries.

To correctly apply consistency and transactional boundaries, DDD practitioners rely on the aggregate pattern. As alluded to, an *aggregate* is an explicit grouping of domain objects designed to support the behaviors and invariants of a domain model while acting as a consistency and transactional boundary. An aggregate treats the cluster of domain objects as a conceptual whole in that there are no order lines—only an order. The order lines do not exist or make sense outside the concept of an order. The aggregate defines the boundary of the cluster of domain objects and separates it in terms of consistency and transactional mechanism from all other domain objects outside it.

Figure 19-11 shows the result of applying invariant-driven aggregate design to the order example. You can see the customer and order are treated as two independent aggregates because there are no invariants in the domain that involve both of them.

To enforce consistency in the cluster of domain objects, all interaction needs to go through a single entity known as the aggregate root (explained in more detail later in the chapter), as highlighted in Figure 19-12. Objects outside of the aggregates can have no access to any of the internal objects of the aggregate; this ensures control of the domain objects and ensures consistency within the aggregate.

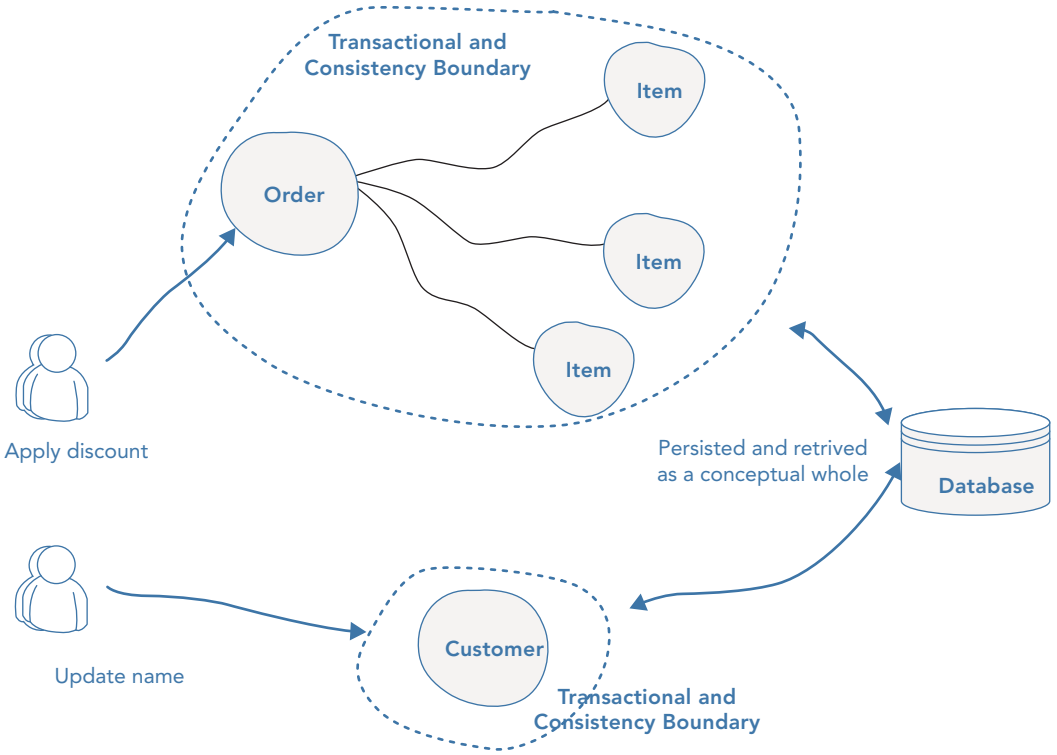


FIGURE 19-11: Aligning transactional boundaries with domain invariants.

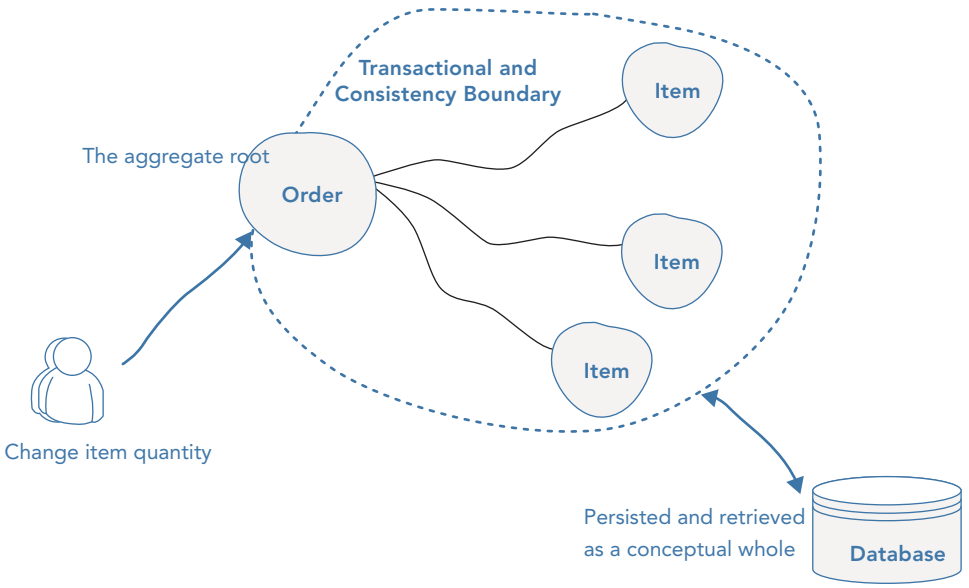


FIGURE 19-12: Enforcing consistency with help from aggregate roots.

Domain objects are not retrieved or persisted individually. The aggregate as a whole is pulled from and committed to the datastore via a repository. Aggregates are the only things that can be persisted and retrieved from the database. No parts of the aggregates can be separately pulled from the data store unless it is purely for reporting. This leads to eventual consistency between aggregates.

Eventual Consistency Externally

Because aggregates are persisted and retrieved atomically, as a conceptual whole, a rule that spans two or more aggregates will not be immediately consistent (if only one aggregate is modified per transaction). Instead, the rule will be eventually consistent. This is because the aggregate being updated only ensures transactional consistency internally. It is not responsible for updating anything outside its consistency boundary. Therefore, aggregates sometimes have a stale piece of information they retrieved from another aggregate, as shown in Figure 19-13.

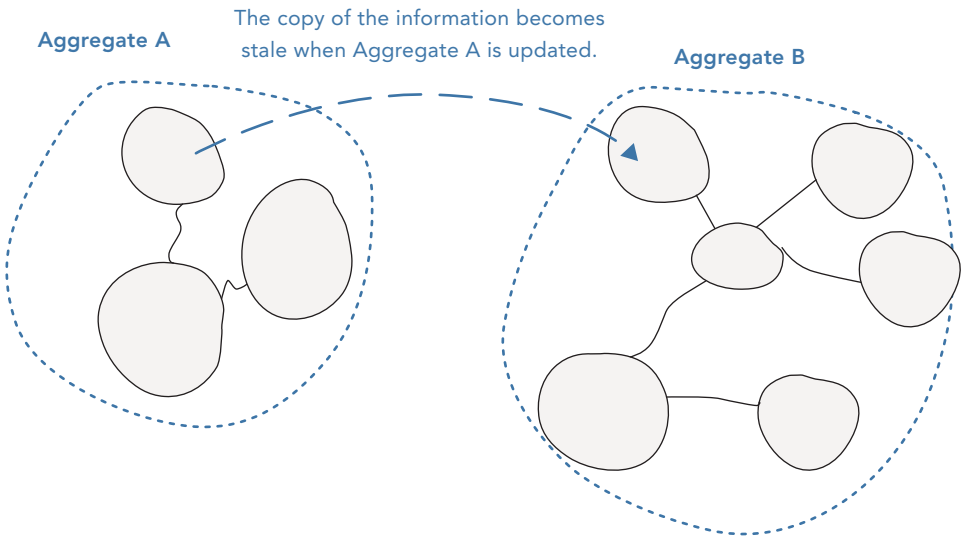


FIGURE 19-13: Aggregates are eventually consistent externally.

To demonstrate an eventually consistent rule spanning multiple aggregates, consider a loyalty policy: If a customer has spent more than $100 in the past year, she gets 10% off all further purchases. In the domain model, there are separate order and loyalty aggregates. When an order is placed, the `Order` aggregate is updated inside a transaction exclusively. At that point, the `Loyalty` aggregate does not have a consistent view of the customer's purchase history because it was not updated in the same transaction. However, the `Order` aggregate can publish an event signalling the order was created, which the `Loyalty` aggregate can subscribe to. In time the loyalty object will be able to update the customer's loyalty when it handles the event, as Figure 19-14 shows.

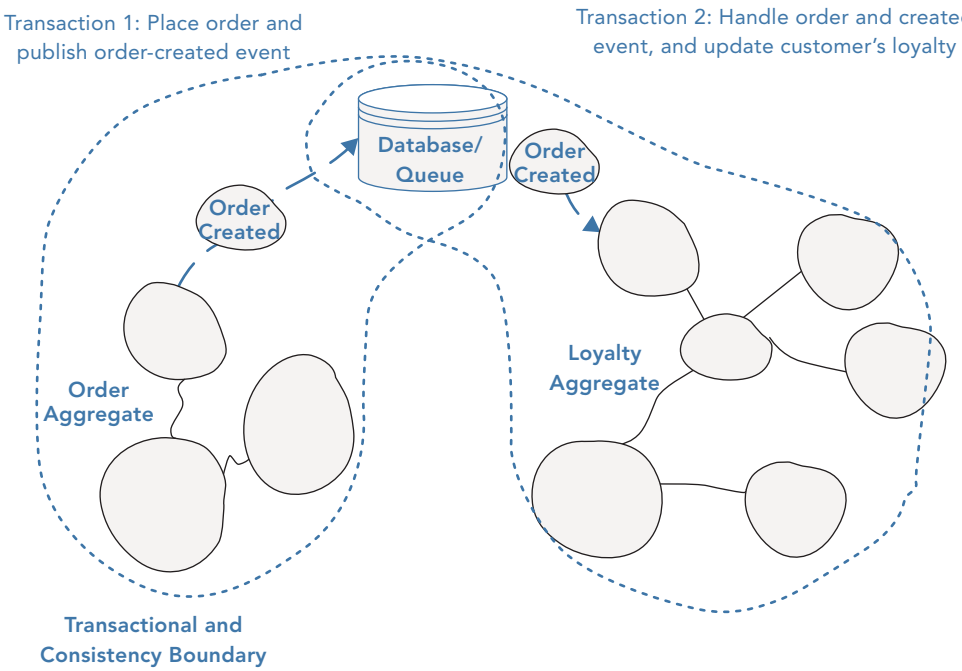


FIGURE 19-14: Eventually consistent Loyalty aggregate.

Importantly, there is a period in which a customer's loyalty does not accurately reflect how much she has spent. This is an undesirable characteristic in isolation, but as a trade-off it prevents you from having to save lots of objects within a single transaction.

It is vital that you get business buy-in when designing eventually consistent aggregates due to user experience drawbacks and edge cases that may occur due to out-of-sync aggregates.

> **NOTE** *Patterns for implementing eventual consistency appear later in this chapter.*

Special Cases

Sometimes it is actually good practice to modify multiple aggregates within a transaction. But it's important to understand why the guidelines exist in the first place so that you can be aware of the consequences of ignoring them.

When the cost of eventual consistency is too high, it's acceptable to consider modifying two objects in the same transaction. Exceptional circumstances will usually be when the business tells you that the customer experience will be too unsatisfactory. You shouldn't just accept the business's decision,

though; it never wants to accept eventual consistency. You should elaborate on the scalability, performance, and other costs involved when not using eventual consistency so that the business can make an informed, customer-focused decision.

Another time it's acceptable to avoid eventual consistency is when the complexity is too great. You will see later in this chapter that robust eventually consistent implementations often utilize asynchronous, out-of-process workflows that add more complexity and dependencies.

To summarize, saving one aggregate per transaction is the default approach. But you should collaborate with the business, assess the technical complexity of each use case, and consciously ignore the guideline if there is a worthwhile advantage, such as a better user experience.

> **NOTE** *Try not to confuse this guideline with loading or creating aggregates. It is perfectly fine to load multiple aggregates inside the same transaction as long as you save only one of them. Equally, it is permissible to create multiple aggregates inside a single transaction because adding new aggregates should not cause concurrency issues.*

Favor Smaller Aggregates

In general, smaller aggregates make a system faster and more reliable, because less data is being transferred and fewer opportunities for concurrency conflicts arise. Accordingly, you should have a bias for small aggregates during design. Try starting small and justifying the addition of each new concept to the aggregate. However, it is still crucial to faithfully model the domain so the size of an aggregate is not the only criteria of good design.

Some consequences of large aggregate design will now be discussed so that you can consciously design aggregates rather than just aimlessly try to make them small. Sometimes you will benefit by having larger aggregates, so understanding when the guidelines don't apply is useful.

Large Aggregates Can Degrade Performance

Each member of an aggregate increases the amount of data that needs to be loaded from and saved to a database, directly affecting performance. Performance can really be harmed, though, when an aggregate spans many tables or documents in a database. Each table requires an additional query or join, which can definitely hurt the local performance of the query and potentially the overall stress on database servers.

Admittedly, the overhead of big aggregates could be inconsequential in many cases. But at the other extreme, the performance degradation may be severe enough to have a damaging business impact. By aiming for smaller aggregates, you reduce the risk of performance problems. If you do ever need to make performance optimizations, a smaller aggregate means compromises to your domain model can be confined to a smaller area.

> **WARNING** *Relying on small aggregates alone to provide a performant system is not a recommended practice. You should still monitor round-trip and database performance so you have a deeper understanding of your data access efficiency.*

Large Aggregates Are More Susceptible to Concurrency Conflicts

A large aggregate is likely to have more than one responsibility, meaning it is involved in multiple business use cases. Subsequently, there is greater opportunity for multiple users to be making changes to a single aggregate. As a further consequence, the likelihood of a concurrency conflict increases, reducing the usability and user satisfaction of your application.

You can use this knowledge when you design your aggregates. You can quantify the number of business use cases an aggregate is involved in. The higher the number, the more you should question your aggregate boundaries and experiment with alternative designs. Again, though, domain alignment is important. Sometimes a single use case might be the optimal number, whereas some aggregates might genuinely be necessary in multiple business use cases.

Large Aggregates May Not Scale Well

Aggregate design is also influenced by scalability concerns. Larger aggregates may span many database tables or documents. This is a form of coupling at the database level, which may prevent you from relocating or repartitioning subsets of the data that have a negative impact on scalability.

With domain-focused aggregates, there are fewer dependencies between your data. This enables you to refactor or relocate data on a more granular basis. For example, if you were working on an e-commerce system and wanted to move just the order data into a different database, there would be less friction than if you had to repartition your aggregates so that order data was not unnecessarily coupled to other kinds of data, such as customer addresses or loyalty.

You can find many stories online of companies relocating parts of their data into different databases. In fact, this trend has been coined *polyglot persistence*. One high-profile example is British broadcasting giant Sky, which decided to move storage of its online checkout data from MySQL to Cassandra (`http://www.computerworlduk.com/in-depth/applications/3474411/sky-swaps-oracle-for-cassandra-to-reduce-online-shopping-errors/`) to combat severe performance degradations as it rapidly acquired new online customers.

In many cases, each bounded context having its own datastore (see Chapter 11, "Introduction to Bounded Context Integration") makes for a solid foundation for scalability. But in some cases, you may have hotspots within a bounded context; favoring smaller aggregate design may put you in good shape to deal with them.

DEFINING AGGREGATE BOUNDARIES

An aggregate's boundary determines which objects will be consistent with each other and which domain invariants will be easy to enforce. It is arguably the most important aspect of designing an aggregate. This section contains a number of principles that can help you decide which objects should be grouped as an aggregate.

eBidder: The Online Auction Case Study

To provide a realistic example of defining aggregates, in this section you follow a fictitious application that is based on the domain of an online auction site, somewhat like eBay, called eBidder. Figure 19-15 shows the bounded contexts that have been defined for the solution space.

The Listing bounded context deals with the items for sale and the format that they are being sold in, whether that is an auction or a fixed price. The Disputes bounded context covers any disputes raised between a seller and member over a listing. The Membership bounded context covers membership to the eBidder site. Finally, the Selling Account bounded context looks after fees and selling activities. In this section, just the Listing bounded context is used to demonstrate aggregate design principles.

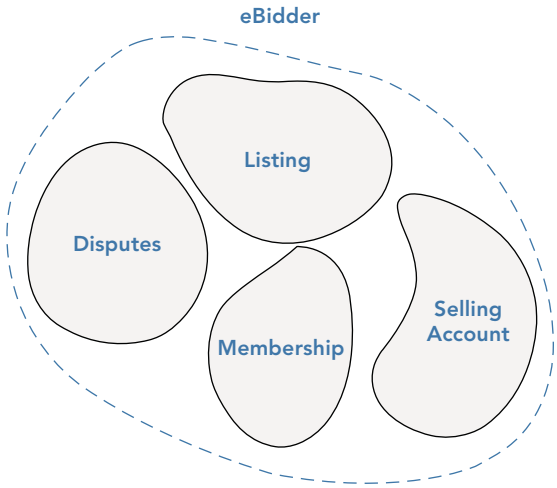


FIGURE 19-15: eBidder bounded contexts.

Figure 19-16 shows the domain model for the Listing bounded context. The full source code for this application is available as part of this chapter's sample code.

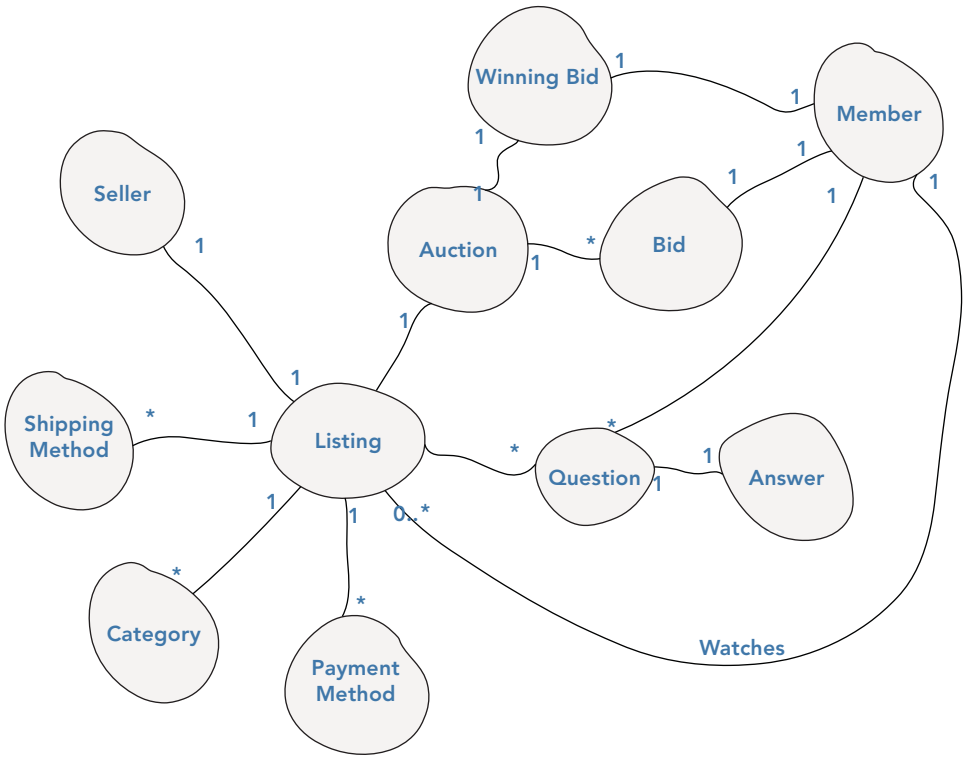


FIGURE 19-16: Listing bounded context aggregates.

An instance of the `Listing` entity represents an item being sold by a seller. Once sold, it can be dispatched via a number of shipping methods, can be listed in categories, and can accept many forms of payment method. A listing can also have questions raised about it from members; these questions can have answers from the seller. A listing has a selling format such as an auction, and an auction can have many members bid on it. At any time, an auction will always have a winning bid after the first bid has been placed. Finally, a member can watch different listings without bidding.

Aligning with Invariants

The most fundamental rule for creating an aggregate is that the cluster of domain objects must be based on domain invariants. This was touched on earlier in the chapter and will be demonstrated in further detail here.

When defining aggregates, you need to identify objects that work together and must be consistent with each other to fulfill business use cases. Here are some of the domain rules that can form the basis of deciding which clusters of objects should be aggregates in the Listing bounded context:

- Each listing must be in at least one category and offer at least one payment type and one shipping method.
- A listing is sold via an auction. An auction has a start and end date and keeps track of the winning bid.
- When a member places a bid, he can enter the maximum amount that he would be happy to pay for the item. However, only the amount required to become the winning bidder is bid. If a second member bids, the auction automatically bids up to the maximum amount. Each automatic bid is logged as a bid.
- A member can ask questions about a listing. A seller can then provide an answer that closes the question.
- An auction can generate many bids, but the current price is defined by the winning bid.
- Members can watch items.

Questions can be asked of a listing; however, there are no invariants that requires data from both questions and listings apart from a reference that can be implemented via an ID property. A listing as a concept can exist without a question, and a question does not depend on any other domain objects apart from an answer. Therefore, an aggregate boundary can be defined around questions and answers, as shown in Figure 19-17.

An auction represents the format of the listing. It holds data on the start and end dates along with the current winning bid, including the maximum amount that the member would bid up to. For bidding to occur, the auction has to be active, and the value of the winning bid must be less than the intended bid. The auction does not depend on the details of the listing to perform its role. With this information, an aggregate boundary can be defined around the auction and winning bid domain objects, as highlighted in Figure 19-18.

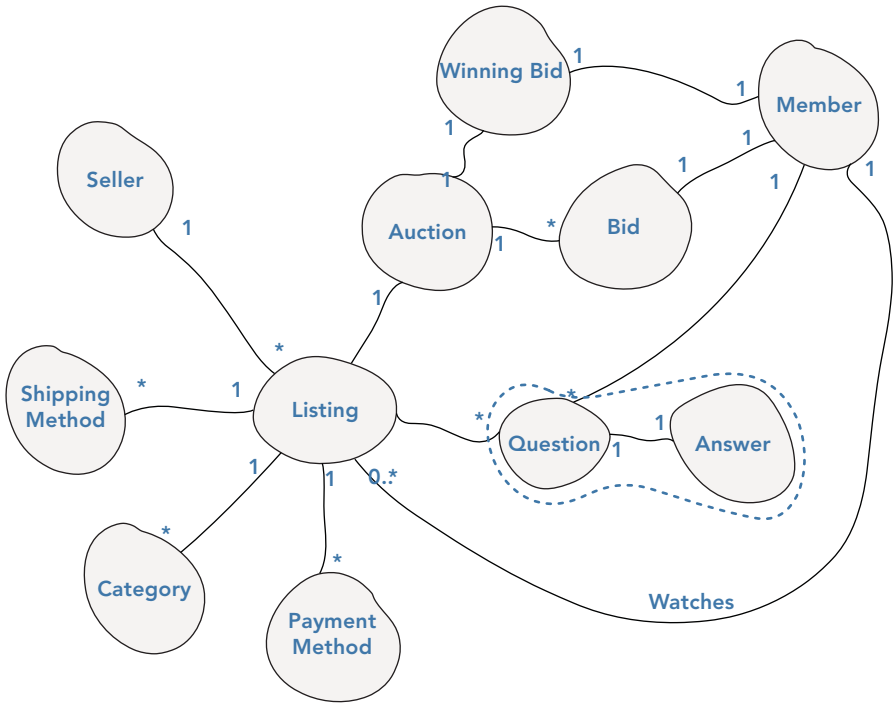


FIGURE 19-17: Question aggregate boundary definition.

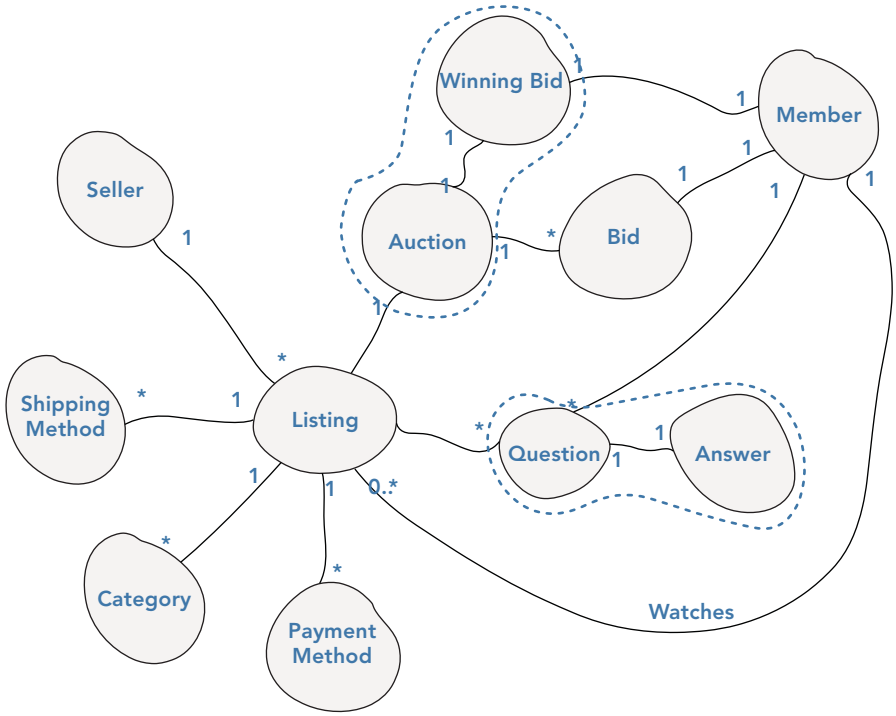


FIGURE 19-18: Auction aggregate boundary definition.

A listing contains all information on the item being sold, including what category it is in, what payment methods can be used to pay for the item, and which shipping methods are available for the item. An invariant requires a listing to have a shipping method, category, and payment method. Accordingly, the listing aggregate can be defined as shown in Figure 19-19.

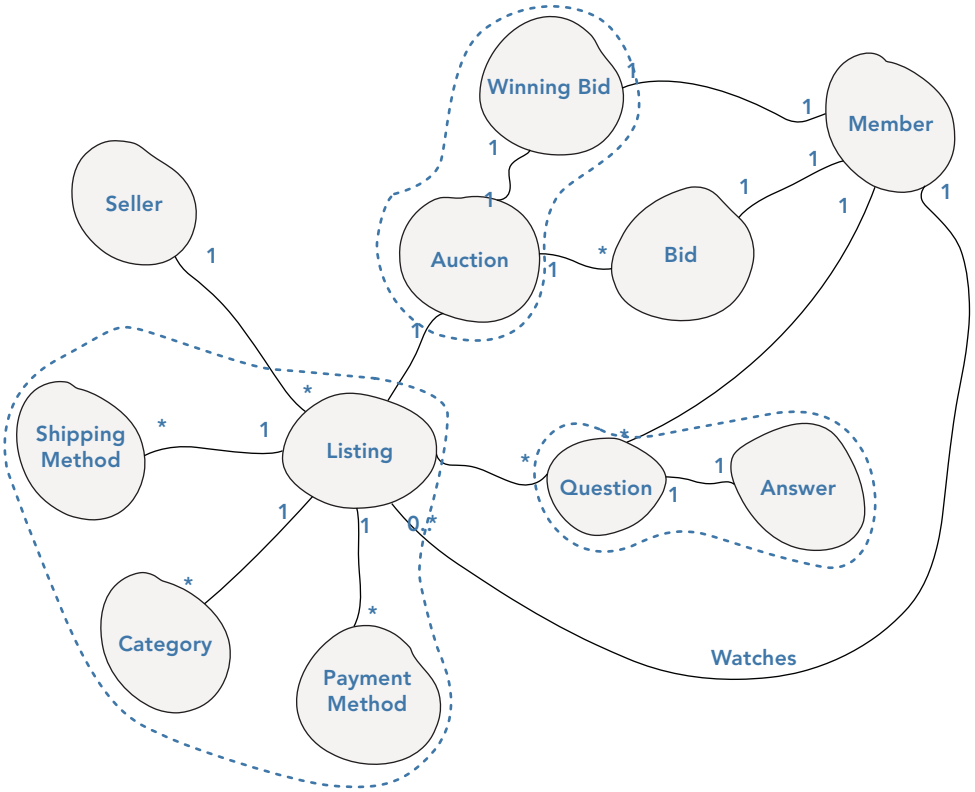


FIGURE 19-19: Listing aggregate boundary definition.

A bid is a historical event; therefore, it can exist as its own aggregate because it is not involved in any invariants. The member and seller only have their identifiers shared so they, too, can become their own aggregates. A member can watch an auction but doesn't have any invariants and is just a container with the listing ID and member ID; it, too, can be its own aggregate, as illustrated in Figure 19-20.

Aligning with Transactions and Consistency

You should try to align your aggregate boundaries with transactions, because the higher the number of aggregates being modified in a single transaction, the greater the chance of a concurrency failure. Therefore, strive to modify a single aggregate per use case to keep the system performant. Figure 19-21 shows how the auction and listing boundaries are aligned with transactions; each aggregate is modified inside a separate transaction.

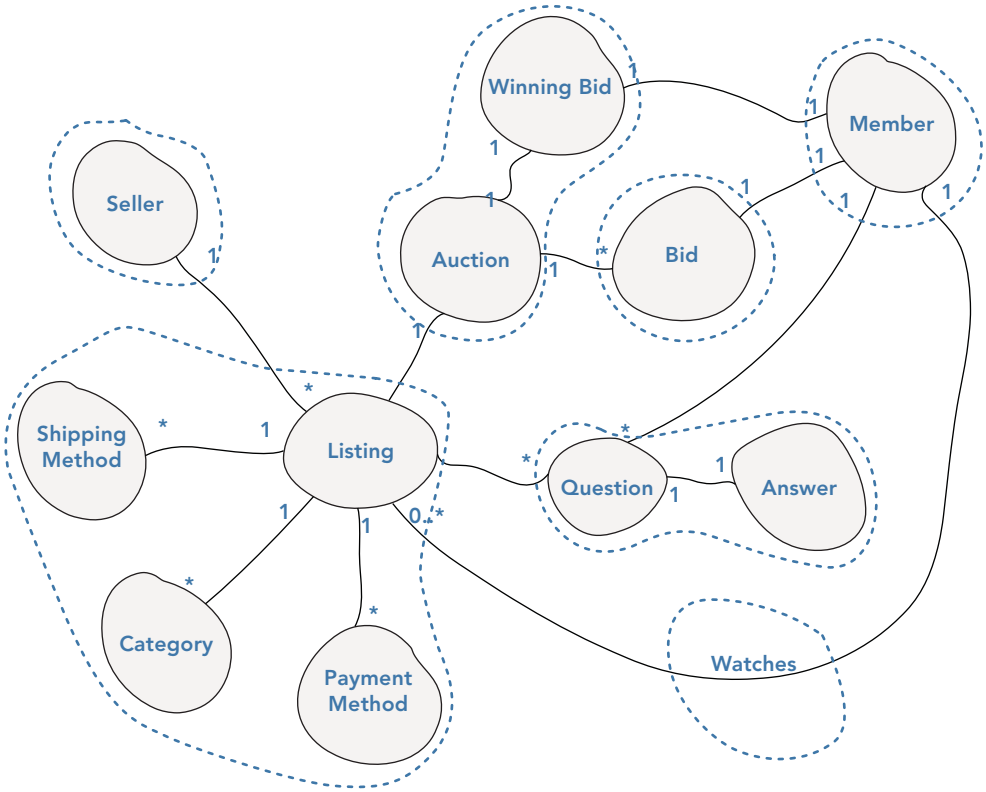


FIGURE 19-20: All aggregate boundary definitions.

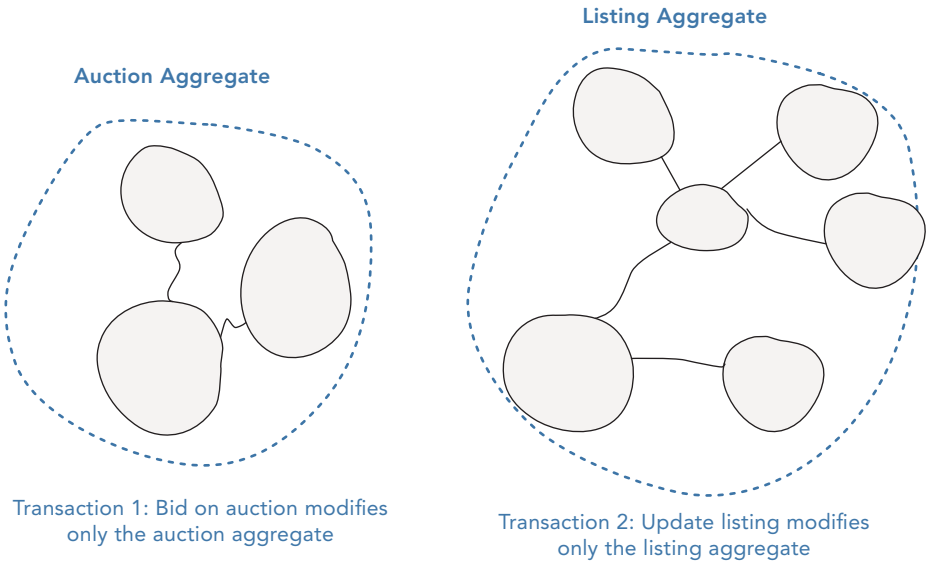


FIGURE 19-21: Aligning aggregates with transactional boundaries.

If you find that you are modifying more than one aggregate in a transaction, it may be a sign that your aggregate boundaries can be better aligned with the problem domain. You should try to find new insights by discussing the use case with domain experts or experimenting with your model. With the latter approach, see if you can resolve the issue by making your aggregates eventually consistent so that only one needs to be updated inside the transaction. It is advisable to get a business perspective on what parts of a system may acceptably be eventually consistent and include them in your aggregate design. In the eBidder application, it's not acceptable for the `Auction` and `WinningBid` to be eventually consistent. This is why they are members of the same aggregate.

Ignoring User Interface Influences

Aggregates should not be designed around UIs. On a listing's page, you would find details of the seller, the current auction price, and the item's details. If the aggregate boundaries were to be defined based on UI needs, the size of the aggregate would become large and would cause locking if a seller wanted to amend a description at the same time as a member wanted to place a bid.

Instead of creating large aggregates to satisfy UIs, it's common practice to map from multiple aggregates onto a single view model that contains all the data a page needs. This will usually be in the form of an application service making multiple repository calls to load aggregates, and then mapping information from the aggregates onto the view model.

> **NOTE** *In some cases, you may want to protect domain structure by not exposing an aggregate's internal data. However, you may still need to present this information on a web page. Chapter 26, "Queries: Domain Reporting" shows how you can achieve this with the mediator pattern, and Chapter 25, "Commands: Application Service Patterns for Processing Business Use Cases," shows an alternative option using the memento pattern.*

In spite of the guidelines to populate a view model from multiple aggregates, there are still drawbacks to this approach that should make you consider alternatives in some cases. One clear sign that you may need to try a different solution is when you are querying three or more repositories to populate a single page. Three database calls can lead to poor performance and excessive load on your servers. In such a scenario, you may want to consider CQRS, which is covered in Chapter 24, "CQRS: An Architecture of a Bounded Context "

Avoiding Dumb Collections and Containers

A common aggregate misconception is that they are merely collections or containers for other objects. This can be a dangerous misconception that results in a lack of clarity in your domain model. Whenever you see a collection or a container-like concept, you shouldn't blindly assume that it is an aggregate.

In the eBidder application, you could look at the auction entity and be tempted to bring its collection of bids into the aggregate. This is logical because, conceptually, an auction has a collection of bids. As you saw in the previous section, though, bids don't belong to an aggregate because there is no domain invariant that applies to both concepts. This type of thinking can lead to complex object graphs, bloated aggregates, and none of the advantages that aggregates bring.

Don't Focus on HAS-A Relationships

Your aggregates should not be influenced by your data model. Associations between domain objects are not the same as database table relationships. Data models need to represent each HAS-A relationship to support referential integrity and build reports for business intelligence and user interface screens. A listing HAS questions, and a listing HAS AN auction, but this does not need to be modeled as a single aggregate. Remember, an aggregate represents a concept in your domain and is not a container for items. A listing does not need a question or a collection of questions to exist, nor does it need to hold this collection to meet any domain invariants. Why would you need to load all the questions to add another?

Auctions and listings are slightly different because there is a one-to-one relationship here. However, from the perspective of use cases and the invariants of the domain, all behaviors on listings need not be consistent with the auction aggregate because there are no invariants that span across them.

When including domain objects in an aggregate, don't simply focus on the HAS-A relationship; justify each grouping and ensure that each object is required to define the behavior of the aggregate instead of just being related to the aggregate.

Refactoring to Aggregates

Defining aggregate boundaries is a reversible and continual activity. There's no need to put yourself under pressure to get your design perfect at the initial attempt. Instead, you should continually be looking for improvements as you learn more about the domain. One scenario that can be particularly enlightening is the addition of a new business use case to your model. A new use case may involve existing entities and uncover new relationships. Consequently, new domain invariants may arise that don't fit well with your existing aggregate designs.

Technical insights can also affect your aggregates—especially performance. If you find that saving—or more likely loading—an aggregate is outside the acceptable performance range, it might be a sign that your aggregate is too large. Undoubtedly, a major performance problem is an acceptable reason to redefine your aggregate boundaries even if conceptually there is an acceptable fit. However, it's also possible that the performance problem is a symptom of a suboptimal conceptual design, so you may want to engage with domain experts or experiment with alternative designs first.

Satisfying Business Use Cases—Not Real Life

Focus on modeling aggregates from the perspective of your business use cases. Ask what invariants must be met to fulfill a use case. By taking this approach, you are less likely to fall into the trap of modeling real life. Instead, you will have small behavior-focused aggregates.

Listen 19-5 shows the use case of placing a bid. You will notice that it doesn't require information on the listing, its category, or description to place a bid. This is why the listing and auction aggregates were modeled separately.

LISTING 19-5: Handling the Use Case of Placing a Bid

```
public class BidOnAuctionService
{
    // .....

    public void Bid(Guid auctionId, Guid memberId, decimal amount)
    {
        using (DomainEvents.Register(OutBid()))
        using (DomainEvents.Register(BidPlaced()))
        {
            var member = _memberService.GetMember(memberId);

            if (member.CanBid)
            {
                var auction = _auctions.FindBy(auctionId);

                var bidAmount = new Money(amount);

                var offer = new Offer(memberId, bidAmount, _clock.Time());

                auction.PlaceBidFor(offer, _clock.Time());
            }
        }
    }
```

Aggregates represent concepts in the solution space; they don't reflect real life. They are merely abstractions used to solve problems in the most effective way while reducing technical complexity. With that in mind, define the boundaries of your aggregates from the point of view of your business use cases—outside in and with a focus on the domain invariants.

IMPLEMENTING AGGREGATES

Aggregate design is a continuous process that is influenced by feedback from your implementation. Persistence, consistency, and concurrency are all important implementation details that can be tricky to get right and may cause you to rethink your aggregate boundaries. Understanding what references are allowed between aggregates is particularly tricky to remember and is likely to feed back into your design process. This is where the concept of an aggregate root can guide you.

Selecting an Aggregate Root

For an aggregate to remain consistent, its constituent parts should not be shared throughout the domain model or made accessible to the service layer. Following this guideline prevents other parts of an application from putting an aggregate into an inconsistent state. But an aggregate still needs to

provide behavior somehow. The answer is to choose an entity for each aggregate to be its aggregate root. All communication with an aggregate should then occur only via its root.

An aggregate root is an entity that has been chosen as the gateway into the aggregate. Figure 19-22 illustrates how the `Auction` entity is the root of the auction aggregate.

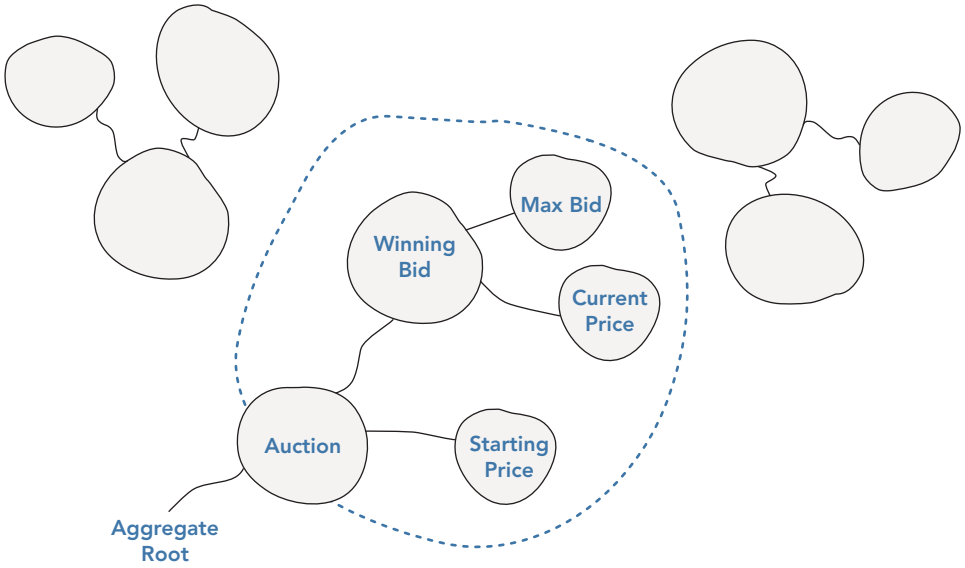


FIGURE 19-22: Aggregate roots are the gateway into an aggregate.

An aggregate root coordinates all changes to the aggregate, ensuring that clients cannot put the aggregate into an inconsistent state. It upholds all invariants of the aggregate by delegating to other entities and value objects in the aggregate cluster.

Domain objects only exist as part of an aggregate, as part of a conceptual whole. As previously mentioned, without the root, clients would have access to the internal structure of an aggregate, be able to bypass the behavior, and interact with member entities directly, as can be seen in the following code snippet:

```
basket.Items.Find(x => x.ProductId == productId).Quantity = newQuantity;
```

The invariant being broken in this example is that an order may have a maximum quantity of ten for each item. By moving the behavior to the root, as shown in the following code snippet, the aggregate's internal members can be encapsulated to protect the invariants of the aggregate.

```
basket.ChangeQuantityOf(productId, newQuantity);
```

Within the body of `ChangeQuantity()` is logic that prohibits a `Customer`, or any other client of the aggregate, from increasing the quantity of any item to greater than ten. In doing so, the root is carrying out its duty of enforcing the invariant.

Exposing Behavioral Interfaces

As with entities and other domain objects, it's highly desirable to expose an aggregate's behavior so that your model explicitly communicates domain concepts. For an aggregate, this means exposing expressive methods on the root for other aggregates to interact with. An aggregate root mediates between other members of an aggregate and is thus the entry point for all external communication. These characteristics are demonstrated by the skeleton of the auction aggregate shown in the following snippet.

LISTING 19-6: The Auction Entity

```
public class Auction : Entity<Guid>
{
    public Auction(Guid id, Guid itemId, Money startingPrice, DateTime endsAt)
    {
        ...
    }

    private Guid ItemId { get; set; }
    private DateTime EndsAt { get; set; }
    private Money StartingPrice { get; set; }
    private WinningBid WinningBid { get; set; }
    private bool HasEnded { get; set; }

    public void ReduceTheStartingPrice()
    {
        ...
    }

    public bool CanPlaceBid()
    {
        ...
    }

    public void PlaceBidFor(Offer offer, DateTime currentTime)
    {
        ...
    }

}
```

`ReduceTheStartingPrice()`, `CanPlaceBid()`, and `PlaceBidFor()` are the behaviors exposed by the aggregate root, and therefore the aggregate. These methods express domain concepts and operate on other members of the aggregate but do not expose them. As discussed previously, this enables the `Auction` aggregate root to ensure that the entire aggregate is always consistent. In this example, the other members of the aggregate are the encapsulated references to the `StartingPrice` value object and the `WinningBid` value object.

Not all members of an aggregate are directly accessible from the root. It's acceptable, and sometimes good design, for them to be a level or two down the object graph. Examples of this in the auction aggregate are the `MaximumBid` and `CurrentAuctionPrice` value objects that belong to the `WinningBid` and not the `Auction` aggregate root, as shown by the WinningBid skeleton in Listing 19-7.

LISTING 19-7: The Winning Bid Value Object

```
public class WinningBid : ValueObject<WinningBid>
{
    public WinningBid(Guid bidder, Money maximumBid,
        Money bid, DateTime timeOfBid)
    {
        ...
    }

    public Guid Bidder { get; private set; }
    public Money MaximumBid { get; private set; }
    public DateTime TimeOfBid { get; private set; }
    public Price CurrentAuctionPrice { get; private set; }

    public WinningBid RaiseMaximumBidTo(Money newAmount)
    {
        ...
    }

    public bool WasMadeBy(Guid bidder)
    {
        ...
    }

    public bool CanBeExceededBy(Money offer)
    {
        ...
    }

    public bool HasNotReachedMaximumBid()
    {
        ...
    }

}
```

> **WARNING** *Keep in mind that this chapter started off by warning about deep object graphs. It's certainly the exception rather than the rule.*

Protecting Internal State

You learned in previous chapters that encapsulating domain structure is important because it enhances the ability to refactor the domain model as your knowledge of the domain improves. Those suggestions, like being wary of getters and setters, are also highly applicable to aggregates. If you follow the advice in the previous section to expose behavioral interfaces, you are already taking a big step toward protecting the internal structure of an aggregate.

If you look back to Listing 19-6 showing the `Auction` skeleton, you see a behavior-focused aggregate root. The public interfaces consist only of behavior. All references to internal members of the objects are encapsulated as private member variables. If an aggregate exposes getters and setters,

the internals of the aggregate may be exposed. Other domain objects or application services might then become coupled to them, damaging your ability to refactor the domain model as you gain new insights. Further, external clients of the aggregate may then be able to put it into an inconsistent state.

Allowing Only Roots to Have Global Identity

You may hear DDD practitioners referring to local and global identity. That's just a concise way of expressing that an aggregate root has a global identity because it can be accessed from outside the aggregate, whereas other members of an aggregate have a local identity because they are internal to the aggregate. Figure 19-23 visualizes this.

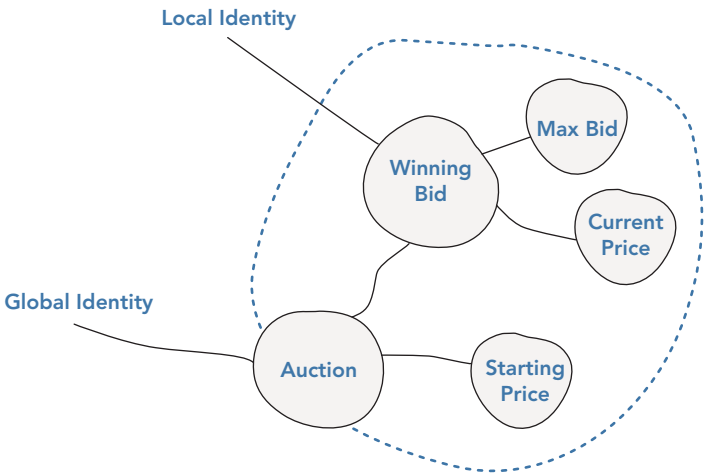


FIGURE 19-23: Aggregate roots have global identity.

Referencing Other Aggregates

Almost always, aggregate roots should keep a reference to the ID of another aggregate root and not a reference to the object itself. This is an important guideline, especially when using ORMs, because loading one aggregate from the database may load all the others that it holds a reference to. Such an occurrence can cause critical performance degradation and hard-to-debug lazy loading issues.

In the Listing bounded context of the eBidder application, the auction aggregate contains a reference to the ID of the `Listing` being auctioned, as the following snippet shows. This is because the `Listing` is not part of the `Auction` aggregate, and an object reference can cause the previously discussed persistence problems.

```
public class Auction : Entity<Guid>
{
    ...

    private Guid ItemId { get; set; }

    ...
}
```

As you saw in the first part of this chapter, referencing other aggregate roots by ID incurs the small cost of relying on repositories to carry out an on-demand lookup of the referenced aggregate in the service layer, similar to the following snippet:

```
var auction = auctionRepository.FindById(auctionId);
var listing = listingRepository.FindById(auction.ItemId);
// carry out business use-case
```

Each additional repository call is likely to be an additional round-trip to the database. You might think this is suboptimal compared to a single query that fetches all the required data in a single round-trip. Admittedly, in isolation this is not ideal, but by applying the aggregate pattern across your domain model, you should see benefits to the system as a whole by an overall more efficient data-access strategy.

There might still be cases in which you are making three of four repository calls. If performance is any kind of concern, you should feel free to reevaluate trade-offs. Maybe it would be better to have one large aggregate to optimize the use case. If the rest of your aggregates are in good shape, it's unlikely to be a problem. However, you might instead want to consider using CQRS or event sourcing.

Admittedly, there are a few subtleties regarding which types of references between aggregates are and aren't allowed. The remainder of this section shows examples that clarify them.

Nothing Outside An Aggregate's Boundary May Hold a Reference to Anything Inside

An easy rule to remember, and one that has been repeated throughout this chapter, is that nothing outside an aggregate should hold a reference to its inner members, as Figure 19-24 illustrates. As discussed, this is important because it protects the inner structure of the aggregate and prevents the aggregate from being put into an inconsistent state.

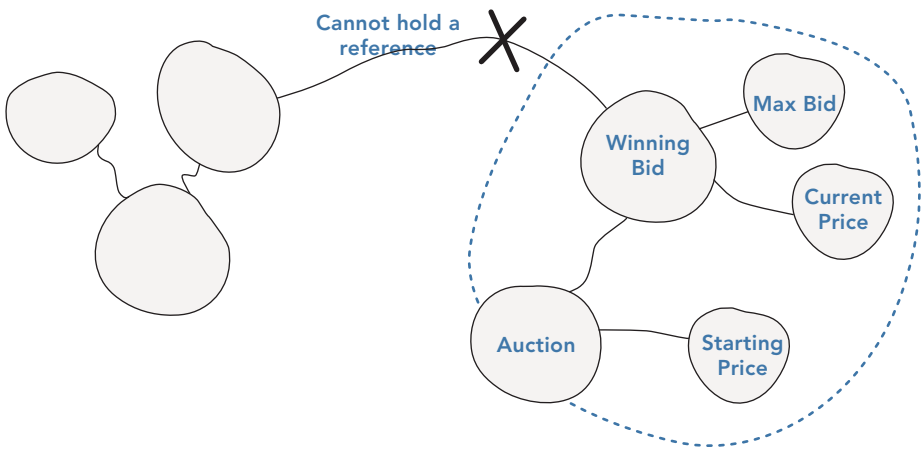


FIGURE 19-24: Consumers of an aggregate may not hold a reference to the aggregate's internal members.

Consumers of the aggregate may only hold a reference to the aggregate root—except for the special case discussed next.

The Aggregate Root Can Hand Out Transient References to the Internal Domain Objects

Even though an aggregate cannot contain a reference to an internal member of another aggregate, it is acceptable to hold a transient reference—one that is used, ideally, inside a single method.

If you think about it, a transient reference to the inner member of an aggregate that is held in a temporary variable is not likely to cause persistence issues. It is not part of the aggregate's object graph, so it is not loaded when the aggregate that requires a transient reference is loaded. However, although in theory this is safe, in practice you should be careful about handing out object references due to the potential for abuse and coupling. Instead, it's advisable to prefer sharing copies or views of an object rather than a reference as shown in Figure 19-25.

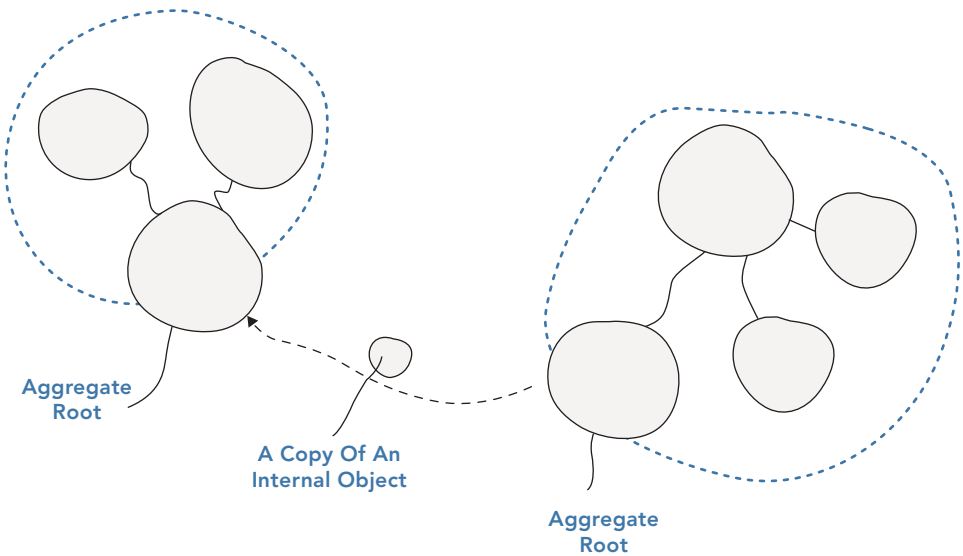


FIGURE 19-25: Sharing information between aggregates by using copies of internal objects.

Objects within the Aggregate Can Hold References to Other Aggregate Roots

Paradoxically, it *is* okay for nonaggregate roots to hold a reference to aggregate roots from other aggregates, as shown in Figure 19-26 where the `WinningBid` from inside the auction aggregate is holding a reference to the root of another aggregate. (This is a unidirectional relationship.)

In this scenario, holding a reference specifically means storing the ID rather than the object reference or pointer, as Listing 19-8 shows. The `WinningBid` is holding a reference to the `Member` that placed the bid by storing its ID.

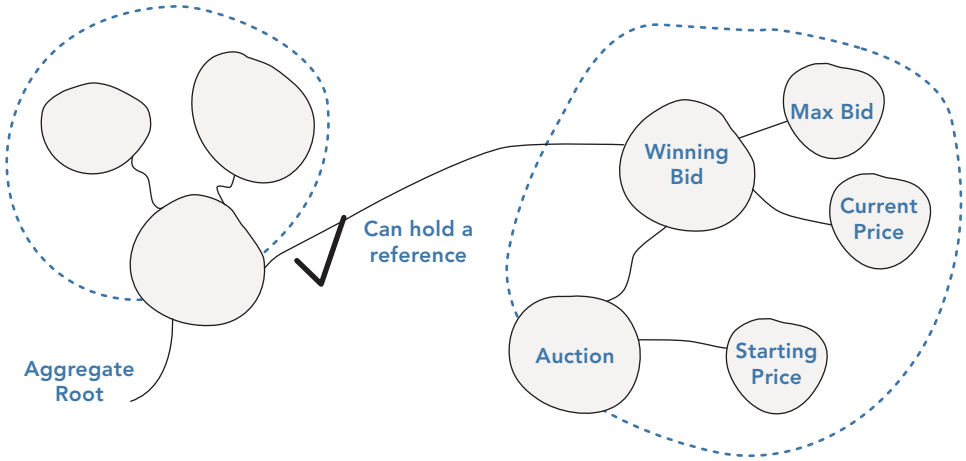


FIGURE 19-26: Non aggregate roots can hold a reference to roots from other aggregates.

LISTING 19-8: The Winning Bid Value Object

```
public class WinningBid : ValueObject<WinningBid>
{
        ...

    public WinningBid(Guid bidder, Money maximumBid,
                      Money bid, DateTime timeOfBid)
    {
        if (bidder == Guid.Empty)
            throw new ArgumentNullException("Bidder cannot be null");

        ...

        Bidder = bidder;
        ...
    }

    public Guid Bidder { get; private set; }

    ...

    public bool WasMadeBy(Guid bidder)
    {
        return Bidder.Equals(bidder);
    }


    ...

    protected override IEnumerable<object>
         GetAttributesToIncludeInEqualityCheck()
```

continues

LISTING 19-8 *(continued)*

```
        {
            return new List<Object>()
            {
              Bidder, MaximumBid, TimeOfBid, CurrentAuctionPrice
            };
        }
    }
```

As previously discussed, entire aggregates are loaded from persistence. So if an object stores a direct reference to another object, both aggregates need to be loaded from persistence. This is covered in more detail in the next section.

Implementing Persistence

Only aggregate roots can be obtained directly with database queries. The domain objects that are inner components of the aggregate can be accessed only via the aggregate root. Each aggregate has a matching repository that abstracts the underlying database and that will only allow aggregates to be persisted and hydrated. This is crucial in ensuring that invariants are met and aggregates are kept consistent. You are susceptible to these problems if child objects of an aggregate can be accessed by directly connecting to the database.

Figure 19-27 shows how aggregates should and should not be loaded from a database if you want to ensure that invariants are not broken and aggregates are fully consistent.

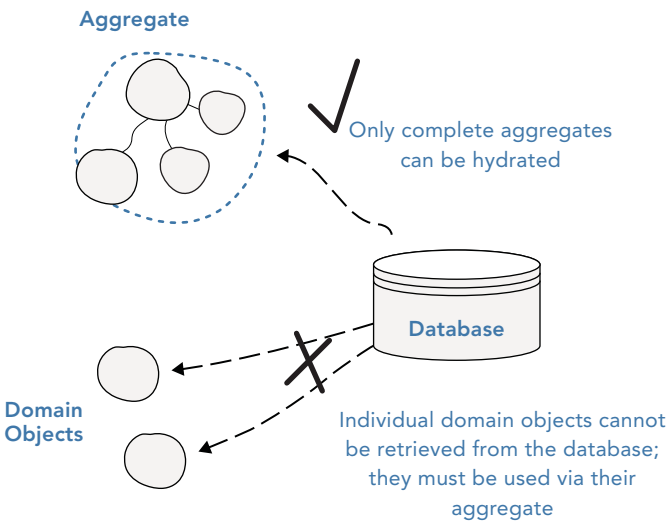


FIGURE 19-27: Aggregates should be loaded from a database entirely to protect their integrity.

Persisting at the granularity of aggregates can be easier to reason about and maintain knowing that you only need to have a one-to-one mapping between aggregates and repositories. It also means that you can think in terms of aggregates. Whenever you need information or behavior,

you just need to know which aggregate to load and which repository to use. This is exemplified by Listing 19-9 that shows the trivial implementation of the `AnswerAQuestionService` application service that only has to care about one repository and loading a single aggregate with it.

LISTING 19-9: Answer a Question Service

```
public class AnswerAQuestionService
{
        private IQuestionRepository _questions;

        ...

        public void Answer(Guid questionId, Guid sellerId, string answer,
                           bool publishOnListing)
        {
            var question = _questions.FindBy(questionId);

            using (DomainEvents.Register(QuestionAnswered()))
            {
                question.SubmitAnAnswer(
                     answer, sellerId, publishOnListing, _clock.Time()
                );
            }
        }

        ...

    }
```

You can see the interface for the `IQuestionRepository` in the next snippet. Notice how it only saves and loads the entire aggregate:

```
public interface IQuestionRepository
{
    Question FindBy(Guid id);

    void Add(Question question);
}
```

Your repositories aren't limited to aggregate ID lookups. In fact, there are few rules outside the fact that methods should save and load entire aggregates. For instance, you may want to load all questions for a certain member, as the following snippet shows:

```
public interface IQuestionRepository
{
    Question FindBy(Guid id);

    void Add(Question question);

    IEnumerable<Question> FindByMemberId(Guid memberId);
}
```

Implementations of your repositories will vary according to your chosen data access and data storage technologies. With an ORM like NHibernate, you can often just pass the aggregate root into the session and the entire aggregate is persisted. On the other hand, with raw SQL, you need to manually store each member of the aggregate.

You may be concerned that there are cases in which loading an entire aggregate is inefficient or unnecessary, when instead you just want to load a single domain object. However, after some experience of applying DDD, you may come to realize that these scenarios are rare. Take an order line, for example. Would you ever want to load a single order line without the rest of the order aggregate? What could you possibly do with it? Scenarios requiring single domain objects are the exception to the rule. More often, they indicate dubious aggregate boundaries.

Access to Domain Objects for Reading Can Be at the Database Level

When handling business use cases, an aggregate is all that you should pull from the database. For reporting on the state of the domain, however, you need not worry about aggregates. Reporting or querying can be performed directly at the database level without the need to hydrate domain objects. Instead, thin view models can be populated by viewing specific queries. As shown in Figure 19-28, an aggregate should be the only domain concept pulled from the database when fulfilling a use case, but for reporting, a simple query against the database is fine.

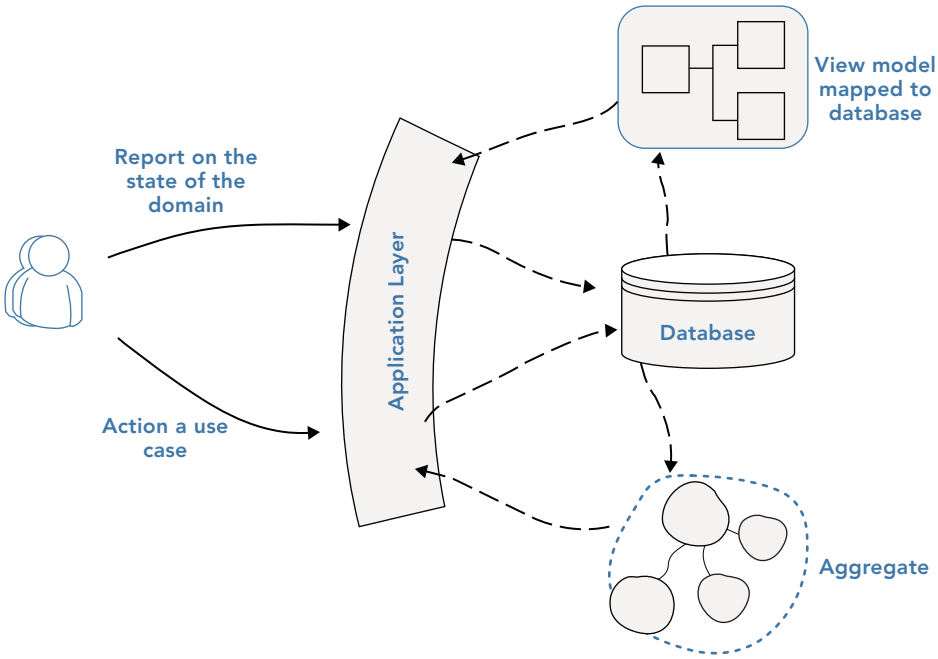


FIGURE 19-28: Loading aggregates versus going directly to the database.

> **NOTE** *Chapter 26, "Queries: Domain Reporting," provides examples of bypassing the domain model to create reports.*

A Delete Operation Must Remove Everything within the Aggregate Boundary at Once

When you delete an aggregate root, you must remove all the child domain objects as well within the same transaction. Components of an aggregate root cannot live on after the root is removed because they have no context and no meaning without the root. For example, an order line has no meaning if the order is removed.

Avoiding Lazy Loading

Many ORMs allow you to load data from a database only when it is accessed in code. Although the intention is to improve performance, the unpredictability of this feature, known as lazy loading, can actually lead to severe performance and reliability penalties. It is highly recommended that you seriously consider avoiding lazy loading unless you have an extremely good reason. If you keep your aggregates small, you are unlikely to need lazy loading.

One of the many ways lazy loading can hurt the performance of your application is by causing the dreaded *select n+1*. Essentially, this problem involves each item in a collection being retrieved individually from the database, rather than collectively in a single query. Each additional query requires an extra database connection and network transportation overhead, leading to inefficient and slow data access. Consider Listing 19-10, for example.

LISTING 19-10: The Booking Entity and Aggregate Root

```
// Aggregate root
public class Booking
{
    ...

    private List<GuestContactDetails> Guests { get; set; }

    public void NotifyGuestsOfConfirmation()
    {
        foreach(var guest in Guests)
        {
            // send a confirmation e-mail
        }
    }

    ...
}
```

This code is from a special events domain model, where groups of people can book adventure days to celebrate birthdays or corporate events. If this code were using lazy loading, each iteration of the `foreach` loop might cause an additional query to the database to get the details for a single guest. Imagine a Christmas celebration booking for an office of 50 colleagues. That could be an astonishing 51 database queries (one for the booking, and one for each guest). That's not a problem if you can afford to give away performance, but in many systems the poor performance of queries like this destroys user experience.

> **NOTE** *You can read about the concept of lazy loading in detail on Martin Fowler's website:* `http://martinfowler.com/eaaCatalog/lazyLoad.html`. *There's also an interesting post on Ayende Rahien's blog related to select n+1 problems arising from lazy loading:* `http://ayende.com/blog/156577/on-umbracos-nhibernates-pullout`.

Implementing Transactional Consistency

As you've learned, an aggregate must be fully persisted or fully rolled back within a transaction to retain consistency. It doesn't matter if an aggregate is stored in one table or many; when an aggregate is persisted, it needs to commit in a single transaction to ensure that, in the event of failure, the aggregate isn't stored in an inconsistent state.

Figure 19-29 shows how the boundary for the auction aggregate is its consistency boundary. All members of the auction aggregate must be updated or rolled back atomically.

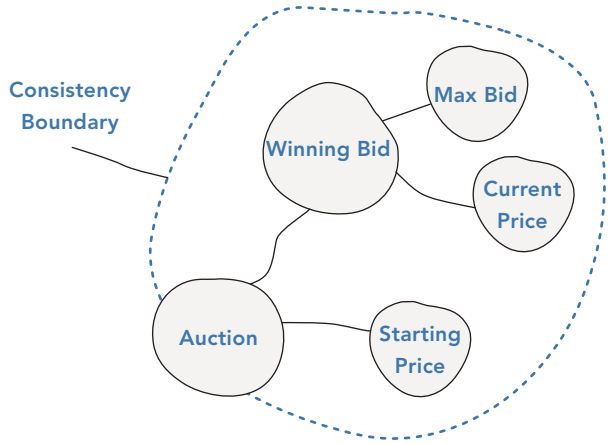


FIGURE 19-29: Aggregate boundaries are consistency boundaries.

Being consistent means that each member of an aggregate has access to the latest state of other members of the aggregate. In the auction aggregate, this means that whenever `WinningBid` is updated, the `Auction` entity must immediately know about it. This rule is easy to apply within an aggregate because objects within an aggregate can contain direct object references to each other. The following snippet highlights that the `Auction` aggregate root is implemented with transactional consistency:

```
public class Auction : Entity<Guid>
{
    ...

     private WinningBid WinningBid { get; set; }

    ...
}
```

Whenever an `Auction`'s `WinningBid` is updated, the `Auction` immediately has access to the updated value. This is because the `Auction` has a reference to the `WinningBid` object. So it always gets the latest value from the source of truth.

Outside of an aggregate, and subsequently outside the consistency boundary, the rules are the opposite; consistency does not have to be strict. This is because references to the innards of an

aggregate are not allowed. In the `Auction` example, a reference to `WinningBid` may not be held by other aggregates. Therefore, when the `WinningBid` is updated, it is not easy for other aggregates or services that use this value to be notified. Instead, they must be eventually consistent—having to deal with stale data, as you saw earlier in the chapter.

Implementing transactional consistency largely depends on your persistence technology and choice of client library. ORMs like Hibernate and NHibernate offer explicit transactions that commit or roll back everything that happens inside it. Some databases, like RavenDB, provide a similar interface.

> **NOTE** *It's important to remember that transactions and other technical details should be handled in the application service layer. Chapter 25, "Commands: Application Service Patterns for Processing Business Use Cases," contains further guidance and examples.*

Implementing Eventual Consistency

Fundamentally, eventual consistency is implemented by aggregates handing out copies of their data to other aggregates, and consequently, aggregates being able to deal with the fact that the information they have received may be stale. This is because, when one aggregate is updated, other aggregates that received a partial copy of its state are not updated immediately with a copy of the new value. As you learned earlier in the chapter, this is why an aggregate's boundary is also a consistency boundary.

There are strategies for implementing eventual consistency. At a basic level, you can simply run transactions synchronously. A benefit of this approach is that the period when aggregates are inconsistent is minimal. This solution is not recommended because it can be risky and difficult to implement well. The common strategy for implementing eventual consistency is the asynchronous approach, using an out-of-process technology like NServiceBus that was introduced in Chapter 12, "Integrating via Messaging." Aggregates stay inconsistent for longer periods of time with the asynchronous approach, but the implementation is more reliable.

Rules That Span Multiple Aggregates

As you learned earlier in the chapter, eventual consistency usually arises when domain rules span multiple aggregates. In this example, you will see how to implement eventual consistency for an e-commerce loyalty program; customers who spend more than $1,000 in one year get a 10% reduction on all future purchases. Initially, you might be tempted to update the order and loyalty aggregate inside the same transaction. As you've learned, though, this increases the chances of a concurrency conflict and reduces scalability options.

Loyalty is often a scenario that does not need immediate consistency. As a customer, it's nice to have the 10% discount applied immediately, but you're probably content to wait until the next working day for the discount to come into effect. Remember, before implementing eventual consistency, you should have a business agreement. Assuming that stakeholders agreed this rule could have a

next-working-day service level agreement (SLA), the following example demonstrates implementing it using eventual consistency and asynchronous domain events.

Asynchronous Eventual Consistency

Asynchronous eventual consistency leads to more reliable systems because operations can usually be retried when failure occurs. However, the drawback is that you need to introduce new technologies and services to carry out and manage the asynchronous behavior. If you're already using a messaging technology like Kafka, Akka, or NServiceBus for sending messages between bounded contexts, the costs of implementing asynchronous eventual consistency for aggregates will not be as high.

Eventually, consistent aggregates can be implemented in the same style that Chapters 11 through 13 implemented eventual consistency between bounded contexts—by using asynchronous domain events. This was illustrated previously in Figure 19-14 where, within a single transaction, an aggregate is updated and an event is published. And then in a subsequent transaction, the event is handled and acted upon.

One pattern that works well with eventually consistent aggregates is using the domain event patterns and its `DomainEvents` class to trigger the process. Inside an event handler, you publish the event with a messaging framework. Listing 19-11 shows the updated `OrderApplicationService` using NServiceBus's `IBus` interface to publish an event indicating the order has been placed.

LISTING 19-11: The PlaceOrder Method Raising a Domain Event

```
public class OrderApplicationService
{
    ...

    public void PlaceOrder(Guid customerId, IEnumerable<Product> products)
    {
        var customer = _customerRepository.FindBy(customerId);

        var order = customer.AddOrder(products);

        // publish as part of transaction
        _bus.Publish(new OrderCreated(order));

        // commit transaction
        _unitOfWork.SaveChanges();
    }
}
```

> **NOTE** *Chapter 18, "Domain Events," is dedicated to the domain events pattern, including a variety of examples.*

At some point in the future, probably on a different thread or even a different server, the messaging framework invokes a handler that runs the second transaction to update the loyalty. Again, using NServiceBus, the handler appears similar to Listing 19-12.

LISTING 19-12: The Order-Created Event Being Handled

```
// NServiceBus handler
public class PlaceOrderHandler : IHandleMessages<OrderCreated>
{
    ...

    public void Handle(OrderCreated @event)
    {
        var loyalty = _loyaltyRepository.FindByCustomerId(
            @event.Order.CustomerId
        );

        loyalty.RegisterPurchaseAmount(@event.Order.TotalSpend);

        // publish next event
    }
}
```

> **WARNING** *If you are saving to a database and publishing messages within the same transaction, you should understand how your messaging system supports two-phase commit (2pc). With NServiceBus and MS SQL Server, you should look into using the Microsoft Distributed Transactions Coordinator (MSDTC), as described in the NServiceBus documentation (*`http://docs.particular.net/nservicebus/transactions-message-processing`*).*

Using a messaging framework isn't the only type of asynchronous solution. Another option that you learned about in Chapter 11, "Introduction to Bounded Context Integration," is database integration. In this example, instead of a message being fired, a flag could be set in the database. Later, a scheduled job could run periodically that scans all new orders, and updates the loyalty of each customer, as shown in Figure 19-30.

As you may be thinking, asynchronous eventual consistency has inherently more complexity, including messaging technologies, asynchronous workflows, and operational monitoring. It's recommended that you decide on a case-by-case basis whether you want to update multiple aggregates in a single transaction or use one of the approaches to eventual consistency.

Implementing Concurrency

An aggregate root can protect invariants by mediating access to internal components. However, it cannot prevent an aggregate from becoming inconsistent due to multiple users modifying it. In collaborative environments, when multiple users change the state of a business object and try

to concurrently persist it to the database, a mechanism needs to be in place to ensure that one user's modification does not negatively affect the state of the transaction for other concurrent users.

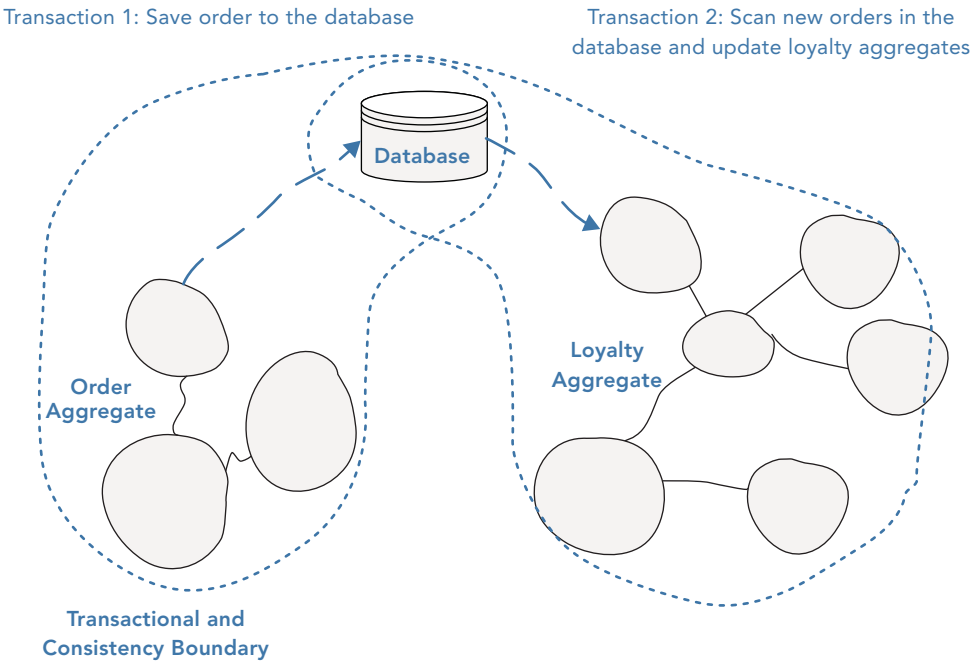


FIGURE 19-30: Eventual consistency using database integration.

There are two forms of concurrency control: optimistic and pessimistic. The optimistic concurrency option assumes that there are no issues with multiple users making changes simultaneously to the state of business objects, also known as *last change wins*. For some systems, this is perfectly reasonable behavior; however, when the state of your business objects needs to be consistent with the state when retrieved from the database, pessimistic concurrency is required.

Pessimistic concurrency can come in many flavors, from locking the data table when a record is retrieved to keeping a copy of the original contents of a business object and comparing that to the version in the data store before an update is made. This ensures that no changes from other parties are made during a transaction. Concurrency can be implemented by having a version/timestamp on aggregates. In this section, you use a version number to check whether a business entity has been amended since being retrieved from the database. Upon an update, the version number of the business entity is compared to the version number residing in the database before committing a change. If the version numbers don't match an exception is raised. This ensures that the business entity has not been modified since being retrieved.

A common three-step approach to implementing concurrency support in an aggregate is to add a version number to the root:

```
public class Loyalty : Entity<Guid>
{
    ...
    public int Version { get; private set; }
    ...
}
```

Then increment the version number before the transaction is committed. But before that, check the version number against the latest version number.

LISTING 19-13: Handling Concurrency Within Repository

```
public class LoyaltyRepository : ILoyaltyRepository
{
   ...
   public void Save(Loyalty loyalty)
   {
       var currentVersion = GetCurrentVersionOf(loyalty);
       if (currentVersion != loyalty.Version)
       {
           var error = string.Format(
               "Expected: {0}. Found: {1}",
               Loyalty.Version, currentVersion
           );
           throw new OptimisticConcurrencyException(error);
       }

       // no error - continue to save
   }

}
```

Admittedly, this generic approach to concurrency will sometimes only be a risk mitigation strategy. Depending on features provided by your data-access library, there could still be a tiny period of time between checking for the latest version number and successfully committing the transaction. Unfortunately, it is possible for the aggregate to be saved inside another transaction inside this period of time. This is why it is desirable to enable datastore or persistence-library level concurrency checking. Many persistence libraries, like NHibernate, Entity Framework, and the RavenDB client, can be configured to manage concurrency for you.

> **NOTE** *You can see a similar example of adding concurrency support to event-sourced entities in Chapter 22, "Event Sourcing." Those examples show both application-level and datastore-level consistency checking.*

THE SALIENT POINTS

- Reduce bidirectional relationships in the domain to communicate more about relationships between domain objects, decrease the implementation complexity, and show traversal direction bias, which helps with where to place domain logic.
- Qualify associations by adding constraints to reduce technical complexity.
- Only add object references when you need to traverse an association to fulfill an invariant; otherwise, simply a reference via an identifier.
- Include associations that support domain invariants; don't simply model real life or replicate the data model in code.
- Aggregates decompose large object graphs into small clusters of domain objects to reduce the complexity of the technical implementation of the domain model.
- Aggregates represent domain concepts, not just generic collections of domain objects.
- Align aggregate boundaries with domain invariants to help enforce them.
- Aggregates are a consistency boundary to ensure the domain model is kept in a reliable state.
- Aggregates ensure transactional boundaries are set at the right level of granularity to ensure a usable application by avoiding blocking at the database level.
- Aim for smaller aggregates to reduce transactional locking and reduce consistency complexities.
- An aggregate root is an entity of the aggregate that is chosen as the gateway into the aggregate. The aggregate root ensures the aggregate is always consistent and invariants are enforced.
- An aggregate root has global identity; the domain objects within the aggregate only have identity within the context of the aggregate root.
- Domain objects outside the aggregate can only hold a reference to the aggregate root.
- When an aggregate is deleted all the domain objects within it must be removed as well.
- No domain objects outside the boundary of the aggregate can hold a reference to internal objects.
- An aggregate root can return copies or references of internal domain objects for use in domain operations, but any changes to these objects must be via the aggregate root.
- Aggregates, not individual domain objects, are persisted and hydrated from the database via the repository.
- An aggregate's internal domain objects can hold references to other aggregate roots.
- Transactions should, ideally, not cross aggregate boundaries.
- There can be inconsistencies between aggregates; use domain events to update aggregates in a separate transaction.

20

Factories

WHAT'S IN THIS CHAPTER?

- How factories separate use from construction
- Applying factory methods to aggregates
- Using a factory method for reconstruction
- Knowing when to use a factory

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 20 download and individually named according to the names throughout the chapter.

How do you create, persist, and retrieve domain objects while maintaining a domain model that is not distracted by technical concerns? The life cycle of complex domain objects may need coordination to begin and when being persisted. Ensuring invariants are met on creation is achieved through the utilization of the Gang of Four factory pattern.

THE ROLE OF A FACTORY

Aggregates, entities, and value objects can become complex when you're creating a domain model for large and rich domains. If intimate knowledge is required to ensure valid instances of a dependent object are created, it can cloud the expressiveness of the domain. The knowledge of other objects' invariants breaks the Single Responsibility Principle (SRP). It is recommended that you separate use from construction and explicitly encapsulate creation logic within a factory object if it is complex or if it can be expressed better. Object creation is not a domain concern, but it does live within the domain layer of an application. You will rarely talk about factories to domain experts, but they do play an important role.

Separating Use from Construction

You can use factories to reconstitute a domain object from a persistence model, or you can use them to create new domain objects, encapsulating complex creation logic. The factory method pattern belongs to the creational group of the Gang of Four design patterns and handles the issue of creating objects without specifying the exact class of object to be created.

The main objective of the factory pattern is to hide the complexities of creating objects. Complexities can include deciding what class to instantiate if a client depends on an abstraction, or it could be checking invariants. A secondary objective of a factory is to clearly express the intent behind variations of an object instantiation, which is typically hard to achieve using constructors alone. The standard implementation of a factory class is to have a static method that returns an abstract class or interface. The client usually, but not always, supplies some kind of information; using the supplied information, the factory then determines which subclass to create and return. The ability to abstract away the responsibility of creating subclasses allows your client code to be completely ignorant of how dependent classes are created. This follows the Dependency Inversion Principle. Another benefit of the factory method pattern is that you centralize the code for the creation of objects; if a change is required in the way an object is generated, it can be easily located and updated without affecting the code that depends on it.

Encapsulating Internals

When you're adding elements to an aggregate, it's important not to expose the structure of the aggregate. Listing 20-1 shows how an application service needs to have detailed knowledge of the basket to add items to it.

LISTING 20-1: Code in the Application Service Layer That Requires Intimate Knowlegde of Valid Object Instantation Logic

```
namespace Application
{
    public class AddProductToBasket
    {
        // ......

        public void Add(Product product, Guid basketId)
        {
            var basket = _basketRepository.FindBy(basketId);

            var rate = TaxRateService.ObtainTaxRateFor(product.Id, country.Id);

            var item = new BasketItem(rate, product.Id, product.price);

            basket.Add(item);

            // ...
        }
    }
}
```

In Listing 20-1, the application service method is required to understand the logic behind how a `BasketItem` is constructed. This is a responsibility it should not have as it should be concerned with coordination only.

You can avoid exposing the internals of the aggregate by adding a factory method. Listing 20-2 shows the `Basket` object with a new `Add` method that now hides the implementation of how the basket stores items from the application service. You can see that the responsibility has been shifted, with the `Basket` aggregate able to ensure the integrity of its internal collections because it can enforce invariants. The client, the application service, is much simpler now and has no knowledge of how the `Basket` stores products.

LISTING 20-2: Shifting Responsibility to the Domain Layer

```
namespace Application
{
    public class AddProductToBasket
    {
        // ......

        public void Add(Product product, Guid basketId)
        {
            var basket = _basketRepository.FindBy(basketId);

            basket.Add(product);

            // ...
        }
    }
}

namespace DomainModel
{
    public class Basket
    {
        // ......

        public void Add(Product product)
        {

            if (Contains(product))
                GetItemFor(product).IncreaseItemQuantitBy(1);
            else
            {
                var rate = TaxRateService.ObtainTaxRateFor(product.Id,
                                                           country.Id);

                var item = new BasketItem(rate, product.Id, product.price);

                _items.Add(item);
            }
        }
    }
}
```

However, there is still the issue of the `Basket` being responsible for creating dependencies of the `BasketItem` that it does not own (the tax rate). To create a valid item from a product, the `Basket` needs to supply a valid tax rate. To create this tax rate, it relies on a tax rate service. The `Basket` has now taken on a secondary responsibility in that it must always understand how to create a valid item and where to obtain valid tax rates.

To avoid the issue of the `Basket` being responsible for more than it needs to and to hide the internal structure of the `BasketItem`, you can introduce a factory object to encapsulate the creation of the `BasketItem`, including the sourcing of a correct tax rate. Listing 20-3 shows the updated code and how it delegates to the `BasketItemFactory`. If there are changes to how tax rates are calculated or if the `BasketItem` needs other types of information, the `Basket` class is unaffected.

LISTING 20-3: Delegating Object Construction to a Factory

```
namespace DomainModel
{
    public class Basket
    {
        // ......

        public void Add(Product product)
        {

            if (Contains(product))
                GetItemFor(product).IncreaseItemQuantitBy(1);
            else
                _items.Add(BasketItemFactory.CreateItemFor(product,
                                                           deliveryAddress));
        }
    }

    public class BasketItemFactory
    {
        public static BasketItem CreateBasketFrom(Product product, Country country)
        {
            var rate = TaxRateService.ObtainTaxRateFor(product.Id, country.Id);

            return new BasketItem(rate, product.Id, product.price);
        }
    }
}
```

You could have called the service within the constructor of the `BasketItem`, but this is not a good idea. Constructors should be simple. If you find you have complex code within a constructor, it may be a sign that you need a factory.

Hiding Decisions on Creation Type

You can also use a factory in the domain layer to abstract the type that a class requires if there are multiple choices and if this choice is not the responsibility of the client class. The client codes against

an interface or abstract class and leaves the `Factory` class responsible for creating the concrete type if the correct type can't be anticipated.

Listing 20-4 shows that an order can create consignments; this is itself a factory method. What is interesting in this method is that to create a valid consignment, you must select a `Courier`. The `Order` class doesn't know which `Courier` to create, so it delegates to a `CourierFactory` and works against a `Courier` abstract class. The factory creates the specific implementation.

LISTING 20-4: Delegating to a Factory to Construct the Correct Subclass

```
namespace DomainModel
{
    public class Order
    {
        // ...

        public Consignment CreateFor(IEnumerable<Item> items, destination)
        {

          var courier = CourierFactory.GetCourierFor(items,
                                                     destination.Country);

          var consignment = new Consignment(items, destination, courier);

          SetAsDispatched(items, consignment);

          return consignment;

        }
    }


    public class CourierFactory
    {
        public static Courier GetCourierFor(IEnumerable<Item> consignmentItems,
                                            DeliveryAddress destination)
        {

             if (AirMail.CanDeliver(consignmentItems, destination))
             {
               return new AirMail(consignmentItems, destination);
             }
             else (TrackedService.CanDeliver(consignmentItems, destination))
             {
               return new TrackedService(consignmentItems, destination);
             }
             else
             {
               return new StandardMail(consignmentItems, destination);
             }
        }
    }
}
```

The factory class itself delegates to a method on the `Courier` implementations to check whether they can handle the consignment items. Inside the constructor of each courier implementation, the factory class also checks that it can satisfy the request using the same method internally.

Factory Methods on Aggregates

Factories don't always need to be standalone static classes. A factory method can exist on an aggregate to hide the complexities of object creation from clients. In Listing 20-5, the `Basket` class, which you looked at previously, now exposes the ability to create a `WishListItem` from a `BasketItem`. The resulting `WishListItem` object is an entity for the `WishList` aggregate. The `Basket` has nothing to do with the `WishListItem` after its construction, but it does contain the data required for it. The factory method removes the need for the client, the application service, to know how to extract a `BasketItem` and turn it into a `WishListItem`, by moving this control into the `Basket` where it naturally fits.

LISTING 20-5: A Factory Method on an Aggregate

```
namespace DomainModel
{
    public class Basket
    {
        // .....

        public WishListItem MoveToWishList(Product product)
        {
            if (BasketContainsAnItemFor(product_snapshot))
            {
             var wishListItem = WishListItemFactory.CreateFrom(
                                                    GetItemFor(product));

             RemoveItemFor(product);

             return wishListItem;
            }
        }
    }
}
```

A factory method can also create an aggregate itself. Listing 20-6 shows that you can use an `Account` to create an order if there is enough credit within the account.

LISTING 20-6: A Factory Method Constructing an Aggregate

```
namespace DomainModel
{
    public class Account
    {
```

```
        // .....

        public Order CreateOrder()
        {
            if (HasEnoughCreditToOrder())
               return new Order(this.Id,
                                this.PaymentMethod, this.Address);
            else
               throw new InsufficentCreditToCreateAnOrder();
        }
    }
}
```

You can have another factory method that calls a different constructor on the `Order` object if you want to bypass the credit checking. You can see in Listing 20-7 that without the factory methods, it would be difficult to understand the intent behind multiple constructors for the `Order` object.

LISTING 20-7: An Alternative Factory Method on an Aggregate

```
namespace DomainModel
{
    public class Account
    {
        // .....

        public Order CreateAnOrderIgnoringCreditRating()
        {
            return new Order(this.Id, this.PaymentMethod,
                             this.Address, PaymentType.PayBeforeShipping);

        }
    }
}
```

Factories for Reconstitution

If you are not using an object relational mapper that can map a data model to your domain model directly with reflection or if you are retrieving your domain objects from a legacy system, be it via a web service or some kind of flat file, you need to reconstruct your domain objects while ensuring all invariants are met. Using a factory to reconstitute domain objects is slightly more complex than object creation.

Listing 20-8 shows that the repository retrieves the basket persistence model from an external service in a raw data-only state. The repository then delegates to a `BasketFactory`, which is in the domain layer, to create the `Basket` instance from the raw data. You can see that the `BasketFactory` creates a `DeliveryOption` object based on the raw data. It then delegates to it to ensure that it can be used for the basket items in this instance. If the `DeliveryOption` is not valid, it is not used; instead, the null object pattern is used to ensure that a new `DeliveryOption` is chosen.

LISTING 20-8: Using a Factory for Object Reconstitution

```
namespace Infrastructure
{
    public class BasketRepository : IBasketRepository
    {
        // ......

        public Basket FindBy(Guid id)
        {
            BasketDTO rawData = ExternalService.ObtainBasket(id.ToString());

            var basket = BasketFactory.ReconstituteBasketFrom(rawData);

            return basket;
        }
    }
}

namespace DomainModel
{
    public class BasketFactory
    {
        // ...

        public static Basket ReconstituteBasketFrom(BasketDTO rawData)
        {
             Basket basket;

             // ...

             var deliveryOption = new DeliveryFactory.Create(
                                        rawData.DeliveryOptionId);

             if (deliveryOption.CanBeUsedFor(rawData.Items))
                 basket.Set(deliveryOption);
             else
                 basket.Set(DeliveryFactory.CreateNonChosen());

             // .....

             return basket;
        }
    }
}
```

When an aggregate is reconstituted, it is vital that its invariants are met; however, it is also important to realize the reality of the situation in that the object exists in the persisted state. An option with the preceding scenario could have been to throw an exception and have some kind of workflow to repair the object; however, in this instance, it was easier to build the aggregate and replace the delivery option with a placeholder so that the user could choose the appropriate strategy for delivery.

Use Factories Pragmatically

Factories can be effective at de-cluttering your domain model to ensure that it remains expressive. However, they should only be used where they are effective and by no means everywhere an instance of an object needs to be instantiated. Use a factory when it is more expressive than a constructor or if it provides convenience where there is the confusion of more than one constructor. Use a factory where elements needed for construction logic are not the concern of the dependent class.

THE SALIENT POINTS

- A factory separates use from construction in the domain.
- Knowledge of creating a complex domain object can be hidden from client code and domain objects by employing a factory class.
- A factory can be used to create the appropriate instance of a domain object based on the needs of the caller.
- Factory methods can encapsulate the internal state of an aggregate.
- When you have multiple constructers for different purposes use a factory object to increase the expressiveness of the code.

21

Repositories

WHAT'S IN THIS CHAPTER?

- The role and responsibilities of the repository pattern
- The misunderstandings of the repository pattern
- The difference between a domain model and a persistence model
- Strategies for persisting your domain model based on the capabilities of your persistence framework

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 21 download and individually named according to the names throughout the chapter.

How do you create, persist, and retrieve domain objects while maintaining a domain model that is not distracted by technical concerns? For domain objects that need to live beyond in-memory use and be retrieved later, a mapping to a persistence store is required. In order to avoid blurring the lines of responsibility between domain logic and infrastructure code, repositories can be employed. Repositories ensure that persistence ignorance is retained by storing domain aggregates behind a collection façade disguising the true underlying infrastructural mechanism. Repositories ensure technical complexities are kept out of the domain model.

REPOSITORIES

A repository is used to manage aggregate persistence and retrieval while ensuring that there is a separation between the domain model and the data model. It mediates between these two models by using a collection façade that hides the complexities of the underlying storage

platform and any persistence framework. The repository provides similar functionality to a .NET collection for storing and removing aggregates, but it can also include more explicit querying functionality and offer summary information on the aggregate collection.

Repositories differ from traditional data access strategies in three ways:

- They restrict access to domain objects by only allowing the retrieval and persistence of aggregate roots, ensuring all changes and invariants are handled by aggregates.
- They keep up the persistence-ignorant façade by hiding the underlying technology used to persist and retrieve aggregates.
- Most importantly, they define a boundary between the domain model and the data model.

So what does a repository look like? Listing 21-1 shows an interface for a repository that manages the persistence and retrieval of customer aggregates. The interface is kept within the domain model namespace because it is part of the domain model, with the implementation residing in the technical infrastructure namespace.

LISTING 21-1: The Interface of a Customer Repository

```
namespace DomainModel
{
  public interface ICustomerRepository
  {
    Customer FindBy(Guid id);
    void Add(Customer customer);
    void Remove(Customer customer);
  }
}
```

A typical client of a repository is the application service layer. A repository defines all the data-access methods that an application service requires to carry out a business task. The implementation comes under the infrastructure namespace and is usually backed by a persistence framework to do the heavy lifting. Listing 21-2 shows an implementation of the `ICustomerRepository` using the NHibernate framework.

LISTING 21-2: An Implementation of the ICustomerRepository

```
namespace Infrastructure.Persistence
{
    public class CustomerRepository : ICustomerRepository
    {
        private ISession _session;

        public CustomerRepository (ISession session)
        {
            _session = session;
        }

        public IEnumerable<Customer> FindBy(Guid id)
```

```
        {
            return _session.Load<Order>(id);
        }

        public void Add(Customer customer)
        {
            _session.Save(customer);
        }

        public void Remove(Customer customer)
        {
            _session.Delete(customer);
        }
    }
}
```

The implementation of the repository in this case simply delegates to NHibernate's `ISession` interface, which acts as a gateway to the data model. The actual mapping of domain objects to the data model is handled via XML, fluent code, or attribute mapping. Because NHibernate provides POCO persistence (persistence without requiring the domain model objects to inherit or implement any classes), it can map the domain model directly to the data model. Don't worry if you have no experience with NHibernate or Object Relational Mappers (ORMs); some implementation exercises at the end of this chapter explain how to use RavenDB, Entity Framework, and NHibernate to implement a repository.

The salient point to take away is that repositories hide technical complexity from your domain model, enabling domain objects to focus solely on business concepts and logic.

A MISUNDERSTOOD PATTERN

There is much misunderstanding and confusion around the repository pattern, with many regarding it as unnecessary ceremony and needless abstraction. When it's not used in conjunction with a rich domain model, the repository pattern is overly complex and can be avoided for a simpler data access object or better by using a persistence framework directly. However, when modeling a solution for a complex domain, the repository is an extension of the model. It reveals the intent behind aggregate retrieval and can be written in a manner that is meaningful to the domain rather than a technical framework.

Is the Repository an Antipattern?

Many developers have negatively blogged that a repository is an antipattern because it hides the capabilities of an underlying persistence framework. This is actually the point of the repository. Instead of offering an open interface into the data model that supports any query or modification, the repository makes retrieval explicit by using named query methods and limiting access to the aggregate level. By making retrieval explicit, it becomes easy to tune queries, and more importantly express the intent of the query in terms a domain expert can understand rather than in SQL. Besides queries, the repository exposes meaningful persistence methods instead of blindly allowing all the create, read, update, and delete (CRUD) methods, regardless of how appropriate each may be.

The point of the repository pattern is not to make it easier to test your code or to make it easy to swap out underlying persistence storage options. It's to keep your domain model and technical persistence framework separate so your model can evolve without being affected by the underlying technology. Without a repository layer, your persistence infrastructure will likely leak into your domain model and weaken its integrity and ultimately usefulness.

The backlash against the repository pattern seems to stem from a lack of understanding about where and why it should be used. It is a pattern that is useful in certain circumstances but not all. Just like a domain model pattern or any design pattern, blindly applying it to all problems results in more complexity, not less.

The Difference between a Domain Model and a Persistence Model

If your persistence store is a relational database and you are using an ORM—which is designed to remove the need to manually map objects to rows and properties to columns, and write raw SQL—you might be wondering why to bother implementing the repository pattern because you already have a framework that abstracts away the persistence technology. The issue, however, is that an ORM only abstracts the relational data model. It merely represents the data model in an object-oriented manner, enabling you to manipulate data easily in code.

The persistence model that your ORM maps to is different from your domain model, as shown in Figure 21-1. The persistence model is the one within your relational database; it's made up of tables and columns, not entities and value objects. For some domain models, the data model may appear similar, or even the same, but conceptually they are very different. Your domain model is an abstraction of your problem domain, rich in behavior and language. The data model is simply a storage structure to contain the state of your domain model at a given time. ORMs map to the data model and make using it easier. They have little to do with domain models. The role of the repository is to keep the two models separate and not to let them blur into one. An ORM is not a repository, but a repository can use it to persist the state of domain objects.

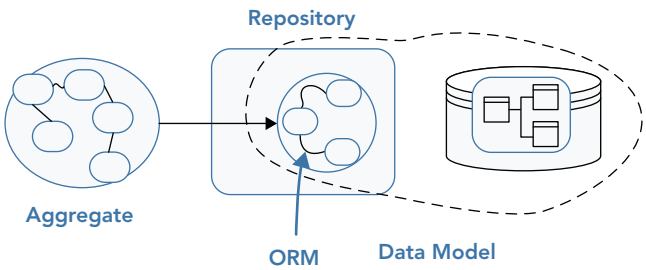


FIGURE 21-1: An ORM maps between the domain and persistence model.

If your domain model is similar to your data model, a sophisticated ORM like NHibernate may be able to map your domain objects directly to the datastore. However, if not, you may be best served having a completely separate data model instead of compromising your domain model. This chapter

examines some of the options to achieve this a little later. For document stores, this problem doesn't exist, and the domain model can be serialized without compromise.

The benefit to separating the data model from the domain model is that it allows you to evolve the domain model without having to constantly think of the data storage and how it will be persisted. Of course, it will ultimately need to be persisted, and you may need to take a pragmatic view and make compromises, but these compromises should be made only when absolutely required, and after the modeling effort is complete rather than up front with the creation of a model.

The Generic Repository

Developers love to reuse code. They generalize a concept to create a common class that can be used for all variations, even though separate explicit solutions would convey the intent of each concept far better. Often you see a form of generic repository contract defined within the domain model. Listing 21-3 shows such an interface with all the operations you would need, including an open-to-extension query method.

LISTING 21-3: A Generic Repository Interface

```
namespace DomainModel
{
  public interface IRepository<TAggregate, TId>
  {
    IEnumerable<TAggregate> FindAllMatching(Expression<Func<T, bool>> query);
    IEnumerable<TAggregate> FindAllMatching(string query);
    TAggregate FindBy(TId id);
    void Add(TAggregate aggregate);
    void Remove(TAggregate aggregate);
  }
}
```

A repository contract defined in the domain layer would inherit this interface and would resemble Listing 21-4.

LISTING 21-4: Inheriting from the Generic Repository Interface

```
namespace DomainModel
{
  public interface ICustomerRepository : IRepository<Customer, Guid>
  {
  }
}
```

The problem with a contract like this is that it assumes that all your aggregates support the same behavior and have the same needs. Some aggregates may be read only, and some may not support the remove method. When an aggregate does not support a concept, you will often find the repository implementation throwing an exception, as in Listing 21-5.

LISTING 21-5: A Concrete Implementation of the Customer Repository

```
namespace Infrastructure.Persistence
{
  public class CustomerRepository : ICustomerRepository
  {
     // ....

     public void Remove(Customer aggregate)
     {
       throw new NotImplementedException(
                     "A customer cannot be removed from the collection.");
     }
  }
}
```

Another problem with the interface is the leaky abstraction of the `FindAllMatching` methods. By providing a method open to extension, you are making it impossible to control queries and optimize fetching strategies. This support of ad hoc queries can quickly lead to severe performance problems in relational databases.

Trying to apply a generalizing strategy to all repositories is a bad idea. It says nothing about the intent behind the retrieval of aggregates. Like all problems, it is better to be explicit. Define your repositories based on their individual needs, and be explicit when naming query methods, as shown in Listing 21-6.

LISTING 21-6: A Specific Customer Repository Interface

```
namespace DomainModel
{
  public interface ICustomerRepository
  {
    Customer FindBy(Guid id);
    IEnumerable<Customer> FindAllThatAreDeactivated();
    void Add(Customer customer);
  }
}
```

When it comes to implementation, there is a place for the generic repository. Behind the concrete implementation, a generic repository can be used to avoid code duplication. As shown in Listing 21-7, this couples an explicit contract with the benefit of code reuse.

LISTING 21-7: The Customer Repository Uses Composition and Does Not Inherit from the Generic Repository

```
namespace Infrastructure.Persistence
{
  public class CustomerRepository : ICustomerRepository
  {
    private IRepository<Customer, Guid> _customersRepository;

    public Customers(IRepository<Customer, Guid> customersRepository)
    {
```

```
            _customersRepository = customersRepository;
        }

        // ....

        public IEnumerable<Customer> FindAllThatAreDeactivated()
        {
               _customersRepository.FindAllMatching(new
                                CustomerDeactivatedSpecification());
        }

        public void Add(Customer customer)
        {
            _customersRepository.Add(customer);
        }
      }
    }
```

If you are using a persistence framework, there is no need to abstract it; you can use this directly within the concrete implementation of a repository. An example using the RavenDB framework is shown in Listing 21-8.

LISTING 21-8: The Customer Repository Does Not Abstract RavenDB

```
namespace Infrastructure.Persistence
{
    public class CustomerRepository : ICustomerRepository
    {
       private IDocumentSession _documentSession;

       public Customers(IDocumentSession documentSession)
       {
           _documentSession = documentSession;
       }

       // ....

       public IEnumerable<Customer> FindAllThatAreDeactivated()
      {
         return _documentSession.Query<Customer>()
                             .Where(x => x.Deactivated == true)
                             .ToList();

      }

       public void Add(Customer customer)
       {
           _documentSession.Store(customer);
       }
    }
}
```

Here you don't abstract away from the underlying framework because there is no value in doing so. Remember that you are only keeping technical concerns out of the domain model. All other parts

of the application need not abstract and hide the implementation details, because this misdirection only proves to confuse readers of the code.

AGGREGATE PERSISTENCE STRATEGIES

Your design strategy, the shape of your aggregates, and whether you are working in a greenfield or a brownfield environment will affect the options for the way you persist your domain objects. The thing that should be at the front of your mind, however, is that you should create your domain objects without thinking about persistence requirements; that is the job of the repository. Your domain objects should be free from any infrastructural code and be as POCO (Plain Old C# Objects) as possible. Compromises should be made only as a last resort.

Using a Persistence Framework That Can Map the Domain Model to the Data Model without Compromise

If you are mapping to a relational database in a greenfield environment and you are using an ORM that supports persistent ignorant domain objects, you will be able to map your domain model directly to the data model, as shown in Figure 21-2.

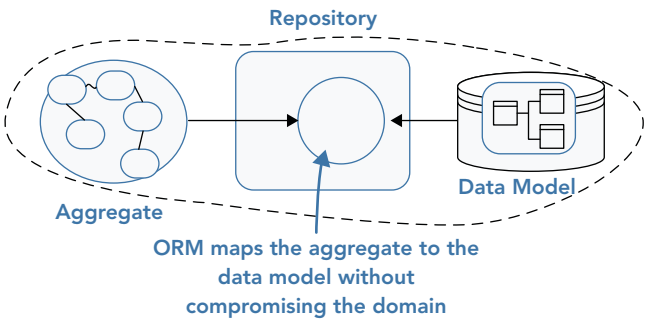


FIGURE 21-2: An ORM maps between the domain and the persistence model.

An ORM usually uses reflection to persist your domain objects and supports the persistence of a fully encapsulated domain model with private getters and setters. Some ORMs require small additions to your domain objects, such as a parameterless constructor in the case of NHibernate, but this is a small price to pay for a seamless map from the domain to the data model.

If you are using a document store where the framework can serialize your aggregate, as shown in Figure 21-3, you will be able to map the aggregate directly without needing an additional mapping to a data model.

A framework that allows your domain model to be free of infrastructural concerns can enable you to evolve your model without compromise and can have a big impact on how you approach modeling.

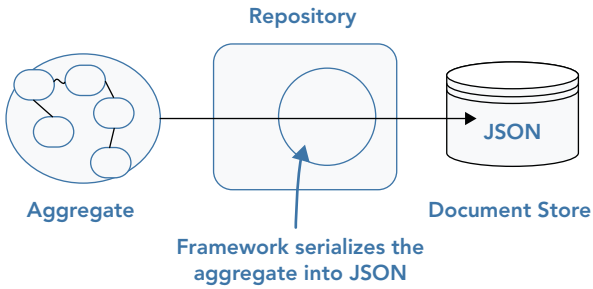


FIGURE 21-3: An aggregate can be serialized and stored.

Using a Persistence Framework That Cannot Map the Domain Model Directly without Compromise

If you are using a persistence framework that does not allow your domain model to be persistence ignorant, you need to take a different approach to the way you persist and retrieve your domain objects so they remain free of infrastructural concerns. There are a number of ways that you can achieve this, but all affect the domain model and the shape of your aggregates. This is, of course, the compromise you need to make your application work.

Public Getters and Setters

A simple method to enable the persistence of domain objects is to ensure that all properties have public getters and setters. The problem is that this leaves your domain objects open to being left in an invalid state because clients can bypass your state-altering methods. You can avoid this through code reviews and governance. However, exposing your properties does make it easy to persist your domain model because your repository has easy access to the domain object's state.

In Listing 21-9, you can see that the `Basket` domain object has public properties for both its collection of items and its delivery cost. If a client bypasses the `Add` method and instead directly adds an item to the collection, the delivery cost is not updated.

LISTING 21-9: Making All Properties Public on the Basket Class

```
namespace DomainModel
{
    public class Basket
    {
        public Money DeliveryCost { get; set;}
        public IList<Item> Items { get; set; }

        public void Add(Product product)
        {
            if (AlreadyContains(product))
                FindItemFor(product).IncreaseItemQuantityBy(1);
            else
```

continues

LISTING 21-9 *(continued)*

```
                Items.Add(BasketItemFactory.CreateItemFor(product));

                DeliveryCosts = DeliveryOption.CalculateDeliveryCostsFor(Items);
            }
        }
    }
```

On the plus side, however, a repository implementation can keep the data model separate from the domain model. Listing 21-10 shows an example of what this might look like.

LISTING 21-10: The Basket Repository Hides the Complexities of Mapping the Basket Object to the Data Store

```
namespace Infrastructure.Persistence
{
    public class BasketRepository : BasketRepository
    {
        // ....

        public Basket FindBy(Guid id)
        {
            Basket basket;

            var basketDataModel = FindDataModelBy(id);

            if (basketDataModel != null)
            {
                basket = new Basket();
                basket.DeliveryCost = basketDataModel.DeliveryOptionCost;

                foreach(var item in basketDataModel.Items)
                {
                    basket.Items.Add(ItemFactory.CreateFrom(item);
                }
            }

            return basket;
        }
    }
}
```

In Listing 21-10, you can see that the data model is retrieved and then mapped to a new instance of the basket domain object. This enables the domain model to be completely unaware of any persistence store, but this is at the cost of exposing the properties of your domain model.

Using the Memento Pattern

If you don't want to expose your domain model's properties and want them to be totally encapsulated, you can utilize the memento pattern. The Gang of Four pattern lets you restore an

object to a previous state. The previous state in this instance can be stored in a database. The way in which it works is that an aggregate produces a snapshot of itself that can be persisted. The aggregate would know how to hydrate itself from the same snapshot. It's important to understand that the snapshot is not the data model; it's merely the state of the aggregates, which again is free from any persistence framework. The repository would still have to map the snapshot to the data model. Figure 21-4 shows a graphical representation of this pattern.

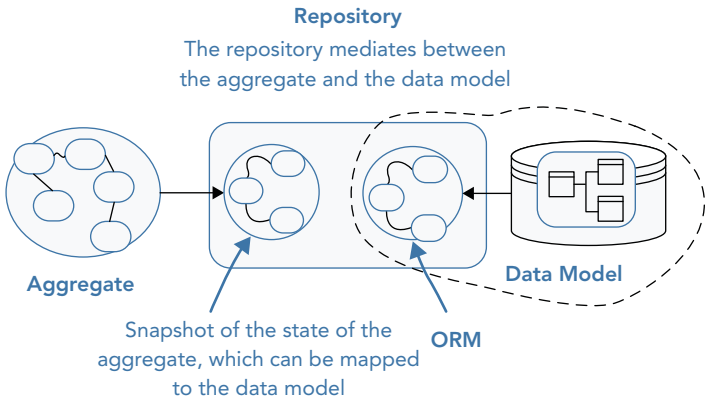


FIGURE 21-4: The memento pattern enables you to map a snapshot of the domain model to the persistence model.

Listing 21-11 shows an implementation of the memento pattern on the `Basket` aggregate.

LISTING 21-11: An Implementation of the Memento Pattern

```
namespace DomainModel
{
    public class Basket
    {
        private Basket(BasketSnapshot snapshot)
        {
            Items = CreateBasketItemsFrom(snapshot.ItemsSnapshot);
            DeliveryCost = snapshot.DeliveryCost;
        }

        // ...

        private Money DeliveryCost { get; set; }
        private IList<Item> Items { get; set; }

        public IEnumerable<ItemView> ShowItems()
        {
            return Items.ConvertToItemView();
```

continues

LISTING 21-11 *(continued)*

```
        }

        public void Add(Product product)
        {
            if (AlreadyContains(product))
                FindItemFor(product).IncreaseItemQuantityBy(1);
            else
                Items.Add(BasketItemFactory.CreateItemFor(product));

            DeliveryCosts = DeliveryOption.CalculateDeliveryCostsFor(Items);
        }

        public BasketSnapshot GetSnapshot()
        {
            var snapshot = new BasketSnapshot();

            snapshot.ItemsSnapshot = CreateItemSnapshotFrom(Items);
            snapshot.DeliveryCost = this.DeliveryCost;

            return snapshot;
        }

        public static void CreateBasketFrom(BasketSnapshot snapshot)
        {
            return new Basket(snapshot);
        }
    }
}
```

The repository implementation would look like Listing 21-12.

LISTING 21-12: The Basket Repository Implementation

```
namespace Infrastructure.Persistence
{
    public class BasketRepository : BasketRepository
    {
        // ....

        public Basket FindBy(Guid id)
        {
            Basket basket;

            BasketDataModel
            basketDataModel = FindDataModelBy(id);

            if (basketDataModel != null)
            {
                var basketSnapshot = ConvertToBasketSnshot(basketDataModel);

                basket = Basket.CreateBasketFrom(basketSnapshot);
```

```
            }

            return basket;
        }
    }
}
```

Another option is to map the snapshot directly to your data model if you are using a persistence framework that allows this but may not allow mapping a full domain object.

Event Streams

Another way to persist your domain model is to use an event stream. Event streams are similar to the memento pattern, but instead of taking a snapshot in time of your aggregate, your repository persists all the events that have occurred to the aggregate in the form of domain events. Listen for events from the domain, and map them to a data model. Again, you need a factory method to reconstruct and replay these events to rebuild the aggregate. Figure 21-5 shows a graphical representation. Event streams are coved in Chapter 22, "Event Sourcing."

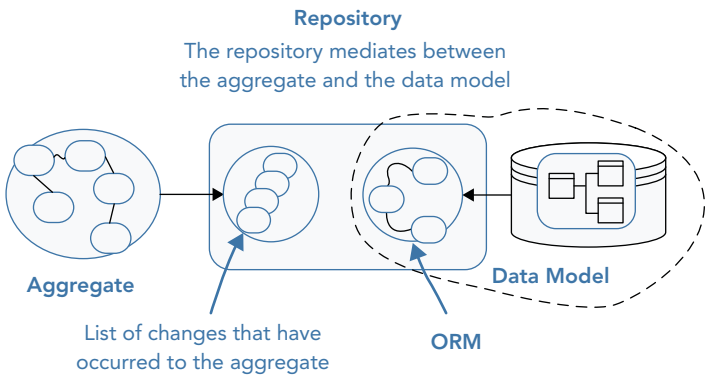


FIGURE 21-5: Save the events that have occurred to an aggregate.

Be Pragmatic

In all the strategies of domain-model persistence, it pays to be pragmatic. A pure domain model should be persistence ignorant in that it should be immune to changes required by the needs of any underlying persistence framework. Purity is good in theory, but in practice it can be difficult to achieve, and sometimes you must choose a pragmatic approach.

If you have a corporate policy on datastores or frameworks, you may need to make compromises when it comes to physically persisting your domain objects. Ensure that you choose a strategy that befits the framework and data storage options available. Don't fight your framework, and recognize that you may need to change your domain model based on performance and scalability. However don't think about these things; start pure, then be pragamatic when you need to. Don't let technical framework drive your design; think in aggregates and explicit retrieval of data rather than ad hoc querying.

A REPOSITORY IS AN EXPLICIT CONTRACT

The contract of a repository is more than just a CRUD interface. It is an extension of the domain model and is written in terms that the domain expert understands. Your repository should be built from the needs of the application use cases rather than from a CRUD-like data access standpoint. The application layer is the client that pulls aggregates from the repository and delegates work to them. Your domain model houses the contracts but rarely uses them except for a specific use case represented in a domain service.

The repository is not an object. It is a procedural boundary and an explicit contract that requires just as much effort when naming methods upon it as the objects in your domain model do. Your repository contract should be specific and intention revealing and mean something to your domain experts.

Consider the repository interface in Listing 21-13. It enables the client to query for domain objects by any means. The contract is flexible and open to extension, but it tells you nothing of the needs or the intent of the retrieving strategies at play in the domain model to obtain collections of aggregates. To truly understand how these methods are used, a developer needs to trawl through all the code to understand the intent and need of queries, let alone how to optimize them. Because the contract is so wide and unspecific, it becomes useless as a boundary and as an interface.

LISTING 21-13: A Repository Interface with Query Methods That Are Open to Extension

```
namespace DomainModel
{
  public interface ICustomerRepository
  {
    Customer FindBy(Guid id);
    IEnumerable<Customer> FindAllThatMatch(Query query);
    IEnumerable<Customer> FindAllThatMatch(String hql);
    void Add(Customer customer);
  }
}
```

Take a look at a different style of repository contract in Listing 21-14. The contract is explicit and tells you a lot about how aggregates are used. Gone are the open-to-extension methods, replaced by two explicit querying methods named in a manner that reveals intent and that is aligned to the language of the domain experts. You are applying the Tell, Don't Ask principle, making the repository do all the hard work by abstracting the query method. It is now clear how you can optimize these queries in the infrastructure implementation.

LISTING 21-14: A Repository Interface with Explicit Query Methods

```
namespace DomainModel
{
  public interface ICustomerRepository
  {
```

```
    Customer FindBy(Guid id);
    IEnumerable<Customer> FindAllThatAreDeactivated();
    IEnumerable<Customer> FindAllThatAreOverAllowedCredit();
    void Add(Customer customer);
  }
}
```

The repository is the contract between the domain model and the persistence store. It should be written only in terms of the domain and without a thought to the underlying persistence framework. Define intent and make it explicit; do not treat the repository contract like object-oriented (OO) code.

TRANSACTION MANAGEMENT AND UNITS OF WORK

Transaction management is chiefly the concern of the application service layer. However, because the repository and transaction management are tightly aligned, it's pertinent to mention them here. (A more detailed examination of transaction management can be found in the following chapter.) A repository is only concerned with the management of a single collection of aggregate roots, whereas a business case could result in updates to multiple types of aggregates.

Transaction management is handled by a unit of work. The role of the unit-of-work pattern is to keep track of all changes to aggregates during a business task. Once all changes have taken place, the unit of work then coordinates the updating of the persistence store within a transaction. To ensure that data integrity is not compromised if an issue arises partway through committing changes to the datastore, all changes are rolled back to ensure that the data remains in a valid state.

In NHibernate, the unit of work pattern is implemented by the session object. In Listing 21-15, an application service is dependent on two repositories to coordinate the task of applying loyalty points. To manage the transaction, a unit of work in the form of NHibernate's session object is required. No changes are persisted to the database until the transaction started by the unit of work is committed.

LISTING 21-15: The Unit-of-Work Pattern Implemented Using NHibernate

```
namespace ApplicationLayer
{
        public class LoyalPointsAccumulator
        {
            private ILoyaltyAccountRepository _loyaltyAccountsRepo;
            private IOrderRepository _ordersRepo;
            private LoyaltyPointsCalculator _pointsCalculator;
            private ISession _session;

            public LoyalPointsAccumulator(
                    ILoyaltyAccountRepository loyaltyAccountsRepo,
                    IOrderRepository ordersRepo,
                    LoyaltyPointsCalculator pointsCalculator,
                    ISession session)
        {
```

continues

LISTING 21-15 *(continued)*

```
                _loyaltyAccountsRepo = loyaltyAccountsRepo;
                _ordersRepo = ordersRepo;
                _pointsCalculator = pointsCalculator;
                _session = session;
        }

        public void CalculatePointsFor(Guid orderId)
        {
            using(var transaction = _session.BeginTransaction())
            {
                var order = _ordersRepo.FindMatching(orderId)
                var account = _loyaltyAccountsRepo.FindMatching(order.CustomerId)

                var points = _pointsCalculator.calculateFor(order);

                account.Add(points, orderid);

                order.Earns(points);

                transaction.Commit();
            }
        }
        }
}
```

You don't need to abstract the underlying persistence framework at the application service level because there is no value to it. Here you are being explicit and using the NHibernate framework directly. It is only at the domain model that you need to keep persistence ignorance. At the application service layer, you don't abstract the framework because it orchestrates both infrastructure and domain logic.

To make things a little clearer, Figure 21-6 shows how the repositories interact with the session object (NHibernate's implementation of the unit-of-work pattern).

To ensure changes are made within the same transaction, the repositories must use the session instance, which the application service layer is using. This is achieved via employment of a factory or inversion of control container to create the application service, which is discussed in detail in the next chapter. The repository instance gets the session injected via its constructor, as shown in Listing 21-16.

LISTING 21-16: NHibernate Session Object Injected via the Constructor of the Repository

```
namespace Infrastructure.Persistence
{
    public class OrderRepository : IOrderRepository
    {
        private ISession _session;

        public OrderRepository(ISession session)
```

```
        {
            _session = session;
        }

        public void Add(Order order)
        {
            _session.Save(order);
        }

        public Order FindBy(Guid id)
        {
            return _session.Load<Order>(id);
        }
    }
}
```

Alternatively, you can use setter-based injection to ensure the repositories have access to the same session instance as the application service. This makes it even more explicit as to what is going on. Listing 21-17 shows how the application service would be updated to leverage setter-based injection on the repository.

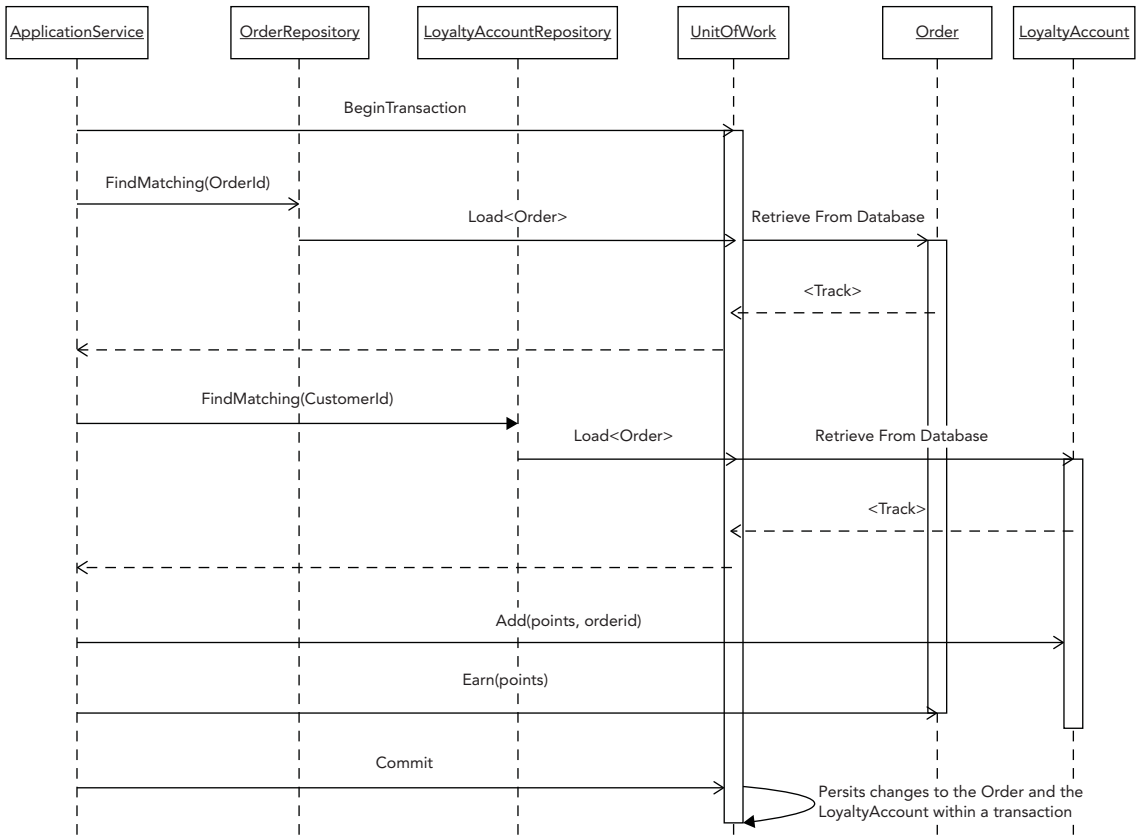


FIGURE 21-6: The unit-of-work pattern.

LISTING 21-17: NHibernate Session Object Injected via the Repository's Public Properties

```
namespace ApplicationLayer
{
       public class LoyalPointsAccumulator
       {
          // ....

         public void CalculatePointsFor(Guid orderId)
         {
           using(var transaction = _session.BeginTransaction())
           {
             _ordersRepo.EnlistIn(_session);
             _loyaltyAccountsRepo.EnlistIn(_session);

             var order = _ordersRepo.FindMatching(orderId)
             var account = loyaltyAccountsRepo.FindMatching(order.CustomerId)

             var points = _pointsCalculator.calculateFor(order);
             account.Add(points, orderid);
             order.Earns(points);

             transaction.Commit();
           }
         }
       }
}
```

Because the unit-of-work pattern has nothing to do with the domain and is purely a technical concern, it doesn't make sense to include the `EnlistIn` method directly on the repository interface contract. You can instead use a separate interface for this to hide the enlist method and ensure you check for the existence of a session in the repository before carrying out any work (see Listing 21-18).

LISTING 21-18: Using Interface Segregation to Separate the Unit of Work and Repository Responsibilities

```
namespace Infrastructure.Persistence
{
    public interface IEnlistInAUnitOfWork
    {
        void EnlistIn(ISession session);
    }
}

namespace Infrastructure.Persistence
{
    public class OrderRepository : IOrderRepository, IEnlistInAUnitOfWork
    {
        private ISession _session;

        public void EnlistIn(ISession session)
        {
```

```
            _session = session;
        }

        public void Add(Order order)
        {
            _session.Save(order);
        }

        public Order FindBy(Guid id)
        {
            return _session.Load<Order>(id);
        }
    }
}
```

TO SAVE OR NOT TO SAVE

A repository should represent a collection of aggregates and behave just like a .NET collection, completely hiding the underlying persistence mechanism. A .NET collection holds references to objects, so any changes made to an object doesn't require them to be explicitly updated within the collection. This means that you won't find a save or update method on a collection; ideally, your repository would behave in the same manner. However, mimicking a collection-like façade is largely down to the capabilities of your persistence framework of choice. A persistence framework that does not support change tracking needs to expose a save method, and the application service that coordinates domain activity needs to ensure that it explicitly persists retrieved aggregates after changes are made to them. If you are choosing not to leverage some kind of persistence framework, you need to do a lot of work. It's best to take a look at some of the options that can help you take the chore out of manual persistence.

Persistence Frameworks That Track Domain Object Changes

ORM frameworks such as NHibernate can track changes in objects that are retrieved by them. Because changes are tracked implicitly, the client of the repository, which is generally the application service, doesn't need to explicitly save these objects. This means there is no requirement for a Save method on the repository contract because changes made are pushed to the persistence store when the unit of work is marked as complete. Consider Listing 21-19, showing both the repository contract and a sample application service. It uses an abstraction on the unit-of-work pattern, which can be implemented by a persistence framework or manually.

LISTING 21-19: NHibernate's Repository Implementation Needs No Save Method

```
namespace DomainModel
{
  public interface ICustomerRepository
  {
    Customer FindBy(Guid id);
    IEnumerable<Customer> FindAllThatAreDeactivated();
    IEnumerable<Customer> FindAllThatAreOverAllowedCredit();
    void Add(Customer customer);
```

continues

LISTING 21-19 *(continued)*

```
  }
}

namespace ApplicationLayer
{
    public class LoyalPointsAccumulator
    {
        // ....

      public void CalculatePointsFor(Guid orderId)
      {
          var order = _ordersRepo.FindMatching(orderId)
          var account = _loyaltyAccountsRepo.FindMatching(order.CustomerId)

          var points = _pointsCalculator.calculateFor(order);

          account.Add(points, orderid);

          order.Earns(points);

          uow.Commit();
      }
    }
}
```

To track changes to domain objects, a persistence framework either takes a snapshot of the retrieved aggregates at read time and compares when persisting the unit of work to determine what to persist, or it returns a proxy of the aggregate and tracks changes as they happen. Once the call to the commit on the unit of work is made, the framework determines if any change has occurred and then generates the SQL required to persist that change in the data model.

Having to Explicitly Save Changes to Aggregates

If you are using a framework to map a data model that does not support change tracking, such as a micro ORM, or you are using raw ADO.NET, you need to support a save method on the repository contract and ensure that the application service calls with the changed domain object. Listing 21-20 shows a repository interface with the Save method and how the application service would use it. You can still call the unit of work to ensure that all changes occur within a transaction.

LISTING 21-20: A Repository Implementation with a Save Method

```
namespace DomainModel
{
  public interface ICustomerRepository
  {
    Customer FindBy(Guid id);
    IEnumerable<Customer> FindAllThatAreDeactivated();
    IEnumerable<Customer> FindAllThatAreOverAllowedCredit();
    void Add(Customer customer);
```

```
    void Save(Customer customer);
  }
}

namespace ApplicationLayer
{
    public class LoyalPointsAccumulator
    {
        // ....
    public void CalculatePointsFor(Guid orderId)
    {
        var order = _ordersRepo.FindMatching(orderId)
        var account = _loyaltyAccountsRepo.FindMatching(order.CustomerId)

        var points = _pointsCalculator.calculateFor(order);

        account.Add(points, orderid);

        order.Earns(points);

        _ordersRepo.Save(order);
        _loyaltyAccountsRepo.Save(account);

        uow.Commit();
    }
    }
}
```

DON'T PUT DIRTY FLAGS ON DOMAIN OBJECTS

It might be tempting to put a flag on a domain object to signify that it has changed. The problem with this is that you are clouding the domain model with persistence concerns. Leave all change tracking to the unit of work and the repository. If you aren't using an ORM, don't worry; there is an exercise showing how you can support persistence ignorance aggregates and repositories at the end of this chapter.

THE REPOSITORY AS AN ANTICORRUPTION LAYER

When you're working in a brownfield environment with an existing persistence store, a repository can help keep the domain model pure by acting as an anticorruption layer, enabling you to create a model without its shape being affected by any underlying infrastructure complexities.

As you have read, a data model and a domain model can be very different. A data model may be spread over several tables or even databases. Also, multiple forms of persistence store can be used, such as flat files, web services, and relational or NoSQL stores. Whatever persistence store you find yourself using, it should not shape the domain model. Repositories map to aggregates, not tables; for example, more than one entity can occupy a single table, as was covered in Chapter 16, where it was shown how to use explicit implementations for different states. Repositories can also store a model over more than a single data store, as shown in Figure 21-7.

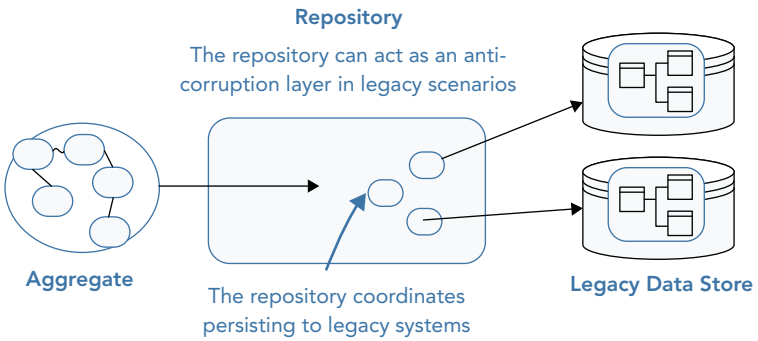


FIGURE 21-7: The repository acts as an anticorruption layer.

OTHER RESPONSIBILITIES OF A REPOSITORY

Besides the persistence and retrieval of aggregates, the repository pattern can expose other facts about its collection of domain objects. For instance, a repository can expose the number of aggregates in its collection, as shown in Listing 21-21.

LISTING 21-21: A Customer Repository with Collection Methods

```
namespace DomainModel
{
  public interface ICustomerRepository
  {
    Customer FindBy(Guid id);
    IEnumerable<Customer> FindAllThatAreDeactivated();
    IEnumerable<Customer> FindAllThatAreOverAllowedCredit();
    void Add(Customer customer);
    int Count();
  }
}
```

Entity ID Generation

If your database or another infrastructural service controls the seeding of IDs, you can abstract this behind the repository and expose it to the application service. It can provide identity explicitly via a method on the interface, as shown in listing 21-22.

LISTING 21-22: Abstracting the Seeding of IDs

```
namespace DomainModel
{
  public interface ICustomerRepository
  {
    Customer FindBy(Guid id);
```

```
    IEnumerable<Customer> FindAllThatAreDeactivated();
    IEnumerable<Customer> FindAllThatAreOverAllowedCredit();
    void Add(Customer customer);
    Guid GenerateId()
  }
}

namespace ApplicationLayer
{
    public class CustomerRegistration
    {
        private ILoyaltyAccountRepository _loyaltyAccountRepo;
        private ICustomerRepository _customerRepo;
        private ISession _session

        public CustomerRegistration(
                                ILoyaltyAccountRepository loyaltyAccountRepo,
                                ICustomerRepository customerRepo,
                                ISession session)
        {
            _loyaltyAccountRepo = loyaltyAccountRepo;
            _customerRepo = customerRepo;
            _session = session;
        }

        public void Register(CustomerDetail details)
        {
            using(var transaction = _session.BeginTransaction())
            {
                var newCustomerId = _customerRepo.GenerateId();
                var customer = CustomerRegistration.CreateFrom(
                                                details, newCustomerId);

                _customerRepo.Add(customer);

                var loyaltyAccount = new LoyaltyAccount(newCustomerId);

                _loyaltyAccountRepo.Add(loyaltyAccount);

                transaction.Commit();
            }
        }
    }
}
```

A repository can also assign an ID during persistence. Usually this is accomplished by having the datastore as the seed for an entity's ID. Listing 21-23 shows part of the XML mapping document for NHibernate. The nested generator tag class attribute defines what generates the ID of the entity. In this case, it is native, and the underlying database automatically generates IDs for the entity via the database identity-seed data type. Later you will see other options for generating the identifier of the entity. The unsaved value attribute is used by NHibernate to compare with the business entity's identifier to help determine if an object has been persisted or is transient (unsaved).

LISTING 21-23: Part of the XML Mapping Document for NHibernate

```
        <?xml version="1.0" encoding="utf-8" ?>
        <hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
           namespace="DomainModel" assembly="DomainModel">

        <class name="Order" table="Orders" lazy="false" >

        <id name="Id" column="OrderId" type="int" unsaved-value="0">
           <generator class="native" />
        </id>

       // ....

  </class>
</hibernate-mapping>
```

Don't worry if you are not familiar with NHibernate because you will be working through some exercises at the end of this chapter to show you how to implement these patterns in practice.

Collection Summaries

Besides counting the number of aggregates you have in a collection, you may want some other summary information on what the collection contains without having to pull back each aggregate and summarize manually. In Listing 21-24, the summary value object can give counts on specific types of customers. For intensive queries and calculations that are best run closer to the raw data, a summary view can be extremely powerful.

LISTING 21-24: A Repository with Summary Methods

```
namespace DomainModel
{
  public interface ICustomerRepository
  {
    Customer FindBy(Guid id);
    IEnumerable<Customer> FindAllThatAreDeactivated();
    IEnumerable<Customer> FindAllThatAreOverAllowedCredit();
    void Add(Customer customer);
    Summary Summary();
  }
}

namespace DomainModel
{
    public class Summary
    {
        public int CustomersCount { get; set; }
        public int DeactivatedCustomers { get; set; }
        public int CustomersOverAllowedCredit { get; set; }
    }
}
```

Concurrency

When multiple users are concurrently changing the state of a domain object, it is important that users are working against the latest version of an aggregate and that their changes don't overwrite other changes that they are not aware of. There are two forms of concurrency control: optimistic and pessimistic. The optimistic concurrency option assumes that there are no issues with multiple users making changes simultaneously to the state of business objects, also known as *last change wins*. For some systems, this is perfectly reasonable behavior; however, when the state of your business objects needs to be consistent with the state when retrieved from the database, pessimistic concurrency is required.

Pessimistic concurrency can come in many flavors, from locking the data table when a record is retrieved to keeping a copy of the original contents of a business object and comparing that to the version in the datastore before an update is made to ensure there have been no changes to a record during a transaction. Many persistence frameworks use a version number to check whether a business entity has been amended since being retrieved from the database. Upon an update, the version number of the business entity is compared to the version number residing in the database before committing a change. This ensures that the business entity has not been modified since being retrieved.

As an example, NHibernate supports optimistic concurrency in several ways. One method that it employs is to use a version tag in the XML mapping file, as shown in Listing 21-25.

LISTING 21-25: An Entity with a Version Field

```
namespace DomainModel
{
  public class Product
  {
    private int _version;

    // ...
  }
}

<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    namespace="DomainModel" assembly="DomainModel">

  <class name="Product" table="Products" lazy="false" >

    <id name="Id" column="Id" type="int" unsaved-value="0">
      <generator class="native" />
    </id>

    // ....

       <version name="Version"  column="Version"/>
  </class>
</hibernate-mapping>
```

The `Version` property is set when the `Product` entity is retrieved from the datastore. If you feel uncomfortable with the `Version` property being on the `Product` entity because in the domain that you are modeling, a version isn't an attribute of a product, you could use an Entity Layer Supertype class, as shown in Listing 21-26 (see also Chapter 16) or return a proxy version of the `Product` entity and include the version ID within.

LISTING 21-26: A Verison Property Added to an Entity Base Class

```
namespace Infrastructure
{
    public abstract class Entity
    {
        public int VERSION {get; private set; };
    }
}

namespace DomainModel
{
    public class Product : Entity
   {

       // ...
    }
}
```

In Listing 21-27 from an application service class, you can see that when a `Product` entity is being saved to the database, the version of the changed entity is included in the `where` clause. NHibernate compares the values and throws a `StaleObjectStateException` because the `Person` object has been modified since you made your second change. The program should then alert the user that the `Product` entity has changed or been deleted since the original retrieval and the update have failed.

LISTING 21-27: Application Service Handling Concurrency Exceptions

```
namespace ApplicationLayer
{
    public class Pricing
    {
        private IProductRepository _productRepository;
        private ISession _session;

        public class Pricing(IProductRepository productRepo, ISession session)
        {
            _productRepository = productRepo;
            _session = session;
        }

        public void SetPrice(
                    UpdatedProductPriceInformation updatedProductInformation)
        {
            try
            {
```

```
                using(var transaction = _session.BeginTransaction())
                {
                    var product = _productRepo
                                  .FindBy(updatedProductInformation.ProductId);

                    product.SetPriceTo(updatedProductInformation.NewPrice);

                    transaction.Commit();
                }
            }
            catch (StaleObjectStateException ex)
            {
                // The product has changed in the time between retrieving it
                // and committing the transaction
            }
        }
    }
}
```

In Listing 21-27, an exception is thrown if the product was modified in the small amount of time between retrieving it from the persistence store to commit the changes on it. What is more likely is that a product changes while the user is viewing it on her desktop/browser. In this situation, the product can be updated while the user is reviewing the data, and the user can then overwrite changes unbeknownst to her. To prevent this from happening, the version flag should be sent to the UI with the view model, and the same version should be sent back with the `UpdatedProductPriceInformation` command. The updated code in Listing 21-28 checks to see if the version that the user was using is the same as the one retrieved from the repository. If it's not, an exception is thrown, and the user can request the fresh updated data and review it again before submitting a price update.

LISTING 21-28: Checking the Version Property of an Entitiy Based on the User Interface

```
namespace ApplicationLayer
{
    public class Pricing
    {
        private IProductRepository _productRepo;
        private ISession _session;

        public class Pricing(IProductRepository productRepo, ISession session)
        {
            _productRepo = productRepo;
            _session = session;
        }

        public void SetPrice(
                      UpdatedProductPriceInformation updatedProductInformation)
        {
            try
            {
                using(var transaction = _session.BeginTransaction())
                {
                    var product = _productRepo
```

continues

LISTING 21-28 *(continued)*

```
                                    .FindBy(updatedProductInformation.ProductId);

                        if (updatedProductInformation.Version != product.Version)
                            throw new ProductHasBeenUpdatedSinceViewingException();

                        product.SetPriceTo(updatedProductInformation.NewPrice);

                        transaction.Commit();
                    }
                }
                catch (StaleObjectStateException ex)
                {
                    Throw New StaleDataUpdateReportBeforeSendingCommmand();
                }
            }
        }
}
```

Audit Trails

If your data model requires metadata that does not make sense in your domain, you can utilize your repository to meet the requirements. Often changes to objects need to be marked with a last change date, which has no meaning to the domain. You can apply this metadata via the repository implementation.

Audit trails and logging can also be supplied via the repository. If you are deleting an aggregate, the repository can log the event for auditing and perhaps take a snapshot of the aggregate or a summary of it.

REPOSITORY ANTIPATTERNS

As with any pattern there are a number of recommended practices that should be avoided. These are known as antipatterns.

Antipatterns: Don't Support Ad Hoc Queries

A *repository* is an explicit contract between the domain model and the persistence mechanism. By exposing a generic catch-all method for querying in an ad hoc manner, you are weakening this layer of abstraction. Consider the contract in Listing 21-29.

LISTING 21-29: Exposing Generic FindBy Methods Instead of Being Explicit

```
public interface ICustomerRepository
{
   void Add(Customer customer);
   Customer FindBy(Guid Id);
   IEnumerable<Customer> FindBy(CustomerQuery query);
}
```

The `FindBy` method takes some form of customer query, perhaps in the form of a specification. The challenge here is that the client can write any type of query, meaning all fetch paths need to be optimized. A method like this weakens the repository's contract and completely removes the intent and meaning of querying the repository. The repository should not be open to extension; it is a boundary and procedural in nature. Instead, favor named methods so that you can write specific code that can be performance tuned and tailored to the underling persistence framework. New methods that are added can then be tuned, and the most appropriate fetching strategy can be created.

Exposing `IQueryable` on a repository is another flavor of supporting ad hoc query methods. The `IQueryable` interface is an extremely flexible pattern for data access; however, it can enable your data model to leak into your domain model. There is nothing wrong with using the `IQueryable` interface behind the boundary of your repository or directly accessing the data model for reporting needs, as shown in Listing 21-30.

LISTING 21-30: The Customer Repository Interface Contract

```
public interface ICustomerRepository
{
   void Add(Customer customer);
   Customer FindBy(Guid Id);
   IQueryable<Customer> Customers();
}
```

Antipatterns: Lazy Loading Is Design Smell

Aggregates should be built around invariants and contain all the properties necessary to enforce those invariants. Therefore, when loading an aggregate, you need to load all of it or none of it. If you have a relational database and are using an ORM for your data model, you may be able to lazy load some domain object properties. This enables you to defer loading parts of an aggregate you don't need. However, the problem with this is that if you can get away with only partially loading an aggregate, you probably have your aggregate boundaries wrong. Also, if data-fetching patterns are affecting your aggregates, you may have built aggregates around views and not business rules.

Antipatterns: Don't Use Repositories for Reporting Needs

The needs of your user-interface screens and the needs of business logic are very different. There will often be a mismatch between information required for a screen and the data contained within a single aggregate root. If only one or two properties are required for display purposes, an entire aggregate needs to be hydrated. Worse still, your domain objects may need extra information to support views that may make little sense to the domain objects they are attached to.

Further still, if you're creating a view model for a screen that spans multiple aggregate roots, you need to pull them all back and pick out the information you require.

Running reports using a transaction model can be slow and compromise the integrity of your model because of the additional properties for UI screens. It is better to use a framework to directly query a read store; this could be the same datastore you use for the transactional work, or it could be a denormalized store.

Don't abstract the persistence framework when querying your data model for reporting purposes. You'll read more about this in the next chapter.

REPOSITORY IMPLEMENTATIONS

In this section, you will work through some repository implementations based on popular .NET persistence frameworks, namely:

- NHibernate
- RavenDB
- Entity Framework
- Dapper Micro ORM

For the examples, you will use a small domain model based on the bidding logic of eBay. Figure 21-8 shows the domain model you will be using. You will not be following a test-first development methodology, because this is an exercise to show you an implementation of a repository rather than how to drive the design of your domain model. With this in mind, you will build up the solutions in a manner where you can follow along and create them yourself.

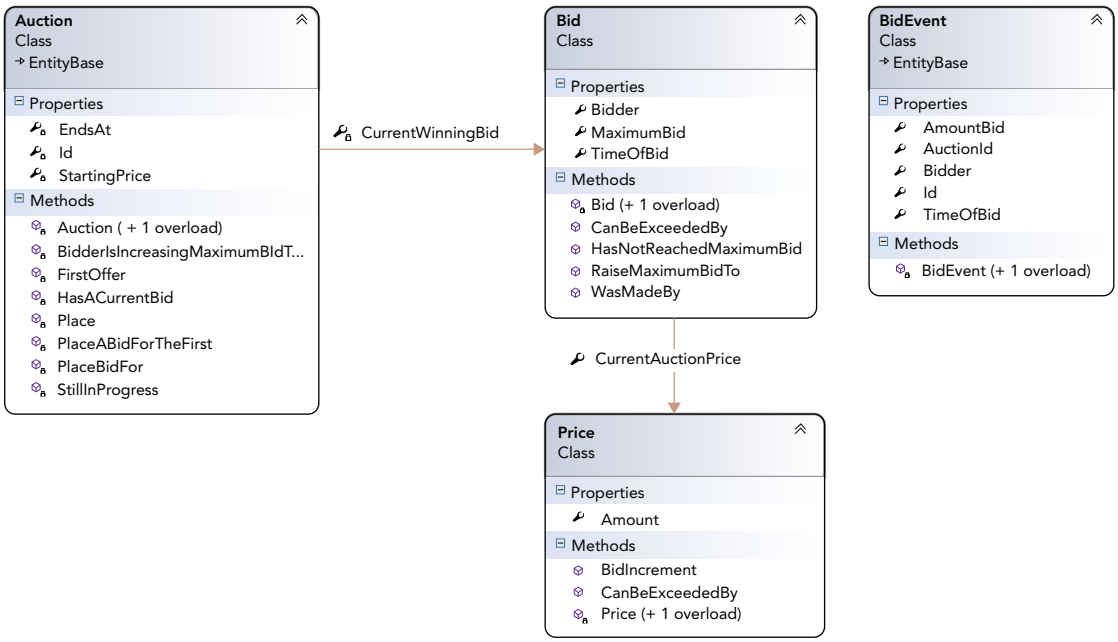


FIGURE 21-8: Solution object model.

The model represents the logic that handles eBay's automatic bidding mechanism:

- When a bidder makes a bid, he enters the maximum amount that he is willing to pay for the item. However, his actual bid is the minimum amount required to beat the previous bid or starting price.

- When a second bidder places a bid, an automatic bidder bids on behalf of the previous bidder up to his maximum amount or enough to beat the second bidder.
- If the second bidder exceeds the previous bidder's maximum bid, the previous bidder is notified that he has been outbid.
- If the second bidder matches the previous bidder's maximum bid, the previous bidder is still the winner of the auction because he was first to bid.

Each of the persistence frameworks that you will use will have a small impact on your domain model. The examples will demonstrate that no matter what persistence framework you are using, you can still support a domain model that is persistent ignorant.

Persistence Framework Can Map Domain Model to Data Model without Compromise

The first two examples of repository implementation use NHibernate and RavenDB. Both of these frameworks allow direct mapping of a domain model to a data model with little or no compromise.

NHibernate Example

NHibernate is a port of the popular open source Hibernate framework for Java. It's a framework that has been around for years, and it's a proven and robust piece of software. NHibernate is an ORM. One of the best features of NHibernate is the support for persistence ignorance; this means that your business objects don't have to inherit from base classes or implement framework interfaces. NHibernate uses an instance of an `ISession`, which is an implementation of the unit of work pattern. `ISession` is also the object you use to query your database with. It acts as your persistence manager and gateway into the database, allowing you to query against it, as well as saving, deleting, and adding entities. There are a number of ways to map business objects to database tables in NHibernate. One of the most popular is via an XML configuration file, but attributes and a fluent code mapping option are also available. This example uses the older XML mapping style.

The domain model in this example has no public properties and is totally encapsulated. You use NHibernate's ability to persist encapsulated models.

Solution Setup

Create a new blank Visual Studio solution named `DDDPPP.Chap21.NHibernateExample` and add to this a class library named `DDDPPP.Chap21.NHibernateExample.Application` and a console application named `DDDPPP.Chap21.NHibernateExample.Presentation`. Within the `Presentation` project, add a reference to the application. Next, create the following folders in the `DDDPPP.Chap21.NHibernateExample.Application` project:

- `Application`
- `Infrastructure`
- `Model`

You can delete `Class1.cs` that is automatically created because you won't be needing it. Your solution should now match Figure 21-9.

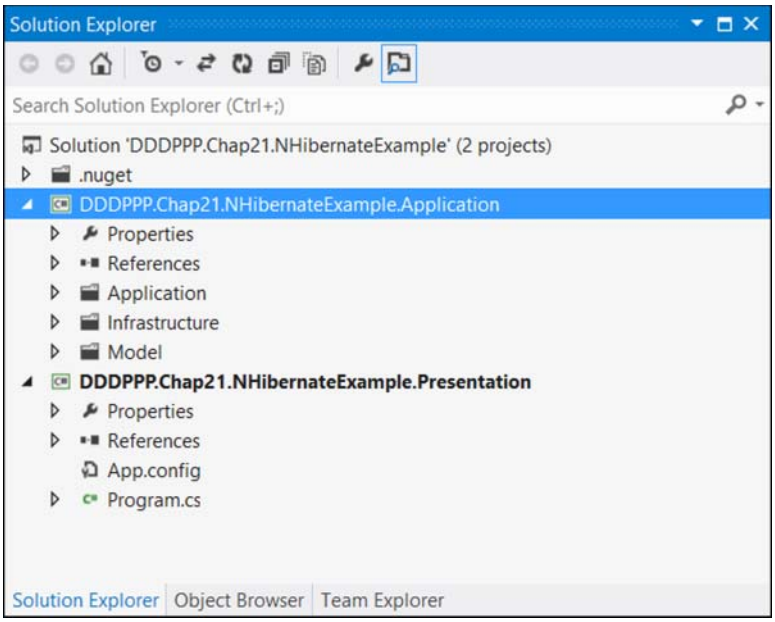


FIGURE 21-9: Visual Solution project structure

Before you build out the application, you need to reference the NHibernate assemblies. Open the NuGet package manager and install NHibernate, as shown in Figure 21-10.

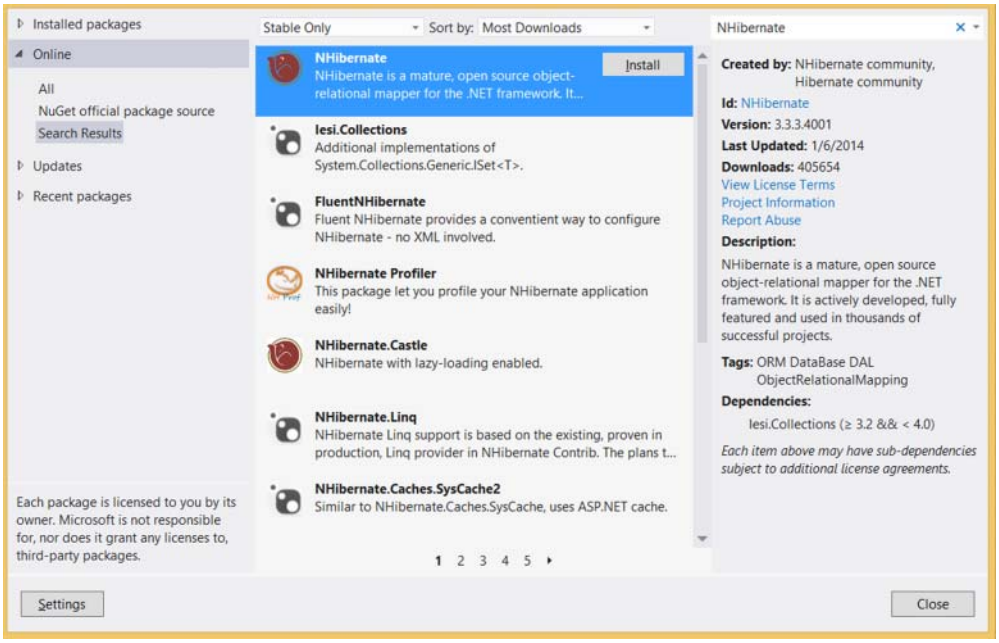


FIGURE 21-10: NuGet Package Manager

Creating the Model

The first thing you are going to do is to create the model. You will be utilizing domain events, so you need to create the infrastructure to support raising and handling events. You will use the framework that you built in Chapter 18, "Domain Events." Add a new class to the `Infrastructure` folder of the `Application` class library project called `DomainEvents` with Listing 21-31. You can find details about how this class works in Chapter 18.

LISTING 21-31: The Domain Event's Infrastructure Class

```
using System;
using System.Collections.Generic;

namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
      /// <summary>
      /// Domain Events class from
      ///  http://www.udidahan.com/2008/08/25/domain-events-take-2/
      /// </summary>
      public static class DomainEvents
      {
          [ThreadStatic]
          private static List<Delegate> _actions;
          private static List<Delegate> Actions
          {
              get
              {
                  if (_actions == null)
                  {
                      _actions = new List<Delegate>();
                  }
                  return _actions;
              }
          }

          public static IDisposable Register<T>(Action<T> callback)
          {
              Actions.Add(callback);

              return new DomainEventRegistrationRemover(() =>
                                          Actions.Remove(callback));
          }

          public static void Raise<T>(T eventArgs)
          {
              foreach (Delegate action in Actions)
              {
                  Action<T> typedAction = action as Action<T>;
                  if (typedAction != null)
                  {
                      typedAction(eventArgs);
                  }
```

continues

LISTING 21-31 *(continued)*

```
            }
        }

        private sealed class DomainEventRegistrationRemover : IDisposable
        {
            private readonly Action _callOnDispose;

            public DomainEventRegistrationRemover(Action toCall)
            {
                _callOnDispose = toCall;
            }

            public void Dispose()
            {
                _callOnDispose();
            }
        }
    }
}
```

You will also be using the value object base class that was covered in Chapter 15, "Value Objects." Create the `ValueObject` class, as defined in Listing 21-32, in the `Infrastructure` folder.

LISTING 21-32: The Value Object Base Class

```
using System.Collections.Generic;
using System.Linq;

namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    public abstract class ValueObject<T> where T : ValueObject<T>
    {
        protected abstract IEnumerable<object>

        GetAttributesToIncludeInEqualityCheck();

        public override bool Equals(object other)
        {
            return Equals(other as T);
        }

        public bool Equals(T other)
        {
            if (other == null)
            {
                return false;
            }
            return GetAttributesToIncludeInEqualityCheck().SequenceEqual(
                        other.GetAttributesToIncludeInEqualityCheck());
```

```
        }

        public static bool operator ==(ValueObject<T> left,
                                       ValueObject<T> right)
        {
            return Equals(left, right);
        }

        public static bool operator !=(ValueObject<T> left,
                                       ValueObject<T> right)
        {
            return !(left == right);
        }

        public override int GetHashCode()
        {
            int hash = 17;
            foreach (var obj in this.GetAttributesToIncludeInEqualityCheck())
                hash = hash * 31 + (obj == null ? 0 : obj.GetHashCode());

            return hash;
        }
    }
}
```

The last piece of infrastructure is to create a base class for the entities within the model, as shown in Listing 21-33.

LISTING 21-33: The Entity Base Class

```
namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    public abstract class Entity<TId>
    {
        public TId Id { get; protected set; }
    }
}
```

With the infrastructure sorted, you can build out the domain model. From within the `Model` folder of the `DDDPPP.Chap21.NHibernateExample.Application` project, create the following two folders that will contain the two aggregates that make up the domain model.

- `Auction`
- `BidHistory`

The first aggregate that you will build is the `Auction` aggregate. The `Auction` aggregate contains all the domain rules around placing bids. You need a value object to represent the money concept in the model. Using Listing 21-34, create a value object named `Money` in the `Auction` folder.

LISTING 21-34: The Money Value Object

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class Money : ValueObject<Money>, IComparable<Money>
    {
        protected decimal Value { get; set; }

        public Money() : this(0m)
        {
        }

        public Money(decimal value)
        {
            ThrowExceptionIfNotValid(value);

            Value = value;
        }

        private void ThrowExceptionIfNotValid(decimal value)
        {
            if (value % 0.01m != 0)
                throw new MoreThanTwoDecimalPlacesInMoneyValueException();

            if(value < 0)
                throw new MoneyCannotBeANegativeValueException();
        }

        public Money Add(Money money)
        {
            return new Money(Value + money.Value);
        }

        public bool IsGreaterThan(Money money)
        {
            return this.Value > money.Value;
        }

        public bool IsGreaterThanOrEqualTo(Money money)
        {
            return this.Value > money.Value || this.Equals(money);
        }

        public bool IsLessThanOrEqualTo(Money money)
        {
            return this.Value < money.Value || this.Equals(money);
        }

        public override string ToString()
        {
```

```
            return string.Format("{0}", Value);
        }

            public int CompareTo(Money other)
               {
                   return this.Value.CompareTo(other.Value);
               }

            protected override IEnumerable<object>
                                  GetAttributesToIncludeInEqualityCheck()
            {
                return new List<Object>() {Value};
            }
    }
}
```

As you read in Chapter 15, a value object is immutable; you cannot alter its state. This is why the `ADD` method returns a new `Money` object rather than changing the state of the object itself.

In the constructor, there is a small piece of logic that ensures you have a nonnegative money amount. If not, one of the following two exceptions in Listing 21-35 is thrown. Add these two classes to the `Auctions` folder.

LISTING 21-35: Domain Exceptions

```
namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class MoneyCannotBeANegativeValueException : Exception
    {
    }
}

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class MoreThanTwoDecimalPlacesInMoneyValueException : Exception
    {
    }
}
```

The next concept that you have in your auction domain is that of an offer. An *offer* is how much a bidder is willing to pay for an item in the auction. The reason you have the concept of an offer is that only offers that exceed the bid increment are turned into bids. The `Offer` class is also a value object. Create the `Offer` class within the `Auction` folder with the code shown in Listing 21-36.

LISTING 21-36: The Office Value Object

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
```

continues

LISTING 21-36 *(continued)*

```
    public class Offer : ValueObject<Offer>
    {
        public Offer(Guid bidderId, Money maximumBid, DateTime timeOfOffer)
        {
            if (bidderId == Guid.Empty)
                throw new ArgumentNullException("BidderId cannot be null");

            if (maximumBid == null)
                throw new ArgumentNullException("MaximumBid cannot be null");

            if (timeOfOffer == DateTime.MinValue)
                throw new ArgumentNullException(
                                "Time of Offer must have a value");

            Bidder = bidderId;
            MaximumBid = maximumBid;
            TimeOfOffer = timeOfOffer;
        }

        public Guid Bidder { get; private set; }
        public Money MaximumBid { get; private set; }
        public DateTime TimeOfOffer { get; private set; }

        protected override IEnumerable<object>
                                 GetAttributesToIncludeInEqualityCheck()
        {
            return new List<Object>()
            {
                Bidder, MaximumBid, TimeOfOffer
            };
        }
    }
}
```

The auction price represents the current price of the winning bid; the price has a method that calculates the bid increment. This is again a value object. Add the `Price` class to the `Model` folder with Listing 21-37. In the real eBay bidding system, there are far more bid increment levels. Here there are only three to keep the exercise simple.

LISTING 21-37: The Price Value Object

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class Price : ValueObject<Price>
    {
        public Price(Money amount)
```

```
        {
            if (amount == null)
                throw new ArgumentNullException("Amount cannot be null");

            Amount = amount;
        }

        public Money Amount { get; private set; }

        public Money BidIncrement()
        {
            if (Amount.IsGreaterThanOrEqualTo(new Money(0.01m)) &&
                            Amount.IsLessThanOrEqualTo(new Money(0.99m)))
                    return Amount.add(new Money(0.05m));

            if (Amount.IsGreaterThanOrEqualTo(new Money(1.00m)) &&
                            Amount.IsLessThanOrEqualTo(new Money(4.99m)))
                    return Amount.add(new Money(0.20m));

            if (Amount.IsGreaterThanOrEqualTo(new Money(5.00m)) &&
                            Amount.IsLessThanOrEqualTo(new Money(14.99m)))
                    return Amount.add(new Money(0.50m));

            return Amount.add(new Money(1.00m));
        }

        public bool CanBeExceededBy(Money offer)
        {
            return offer.IsGreaterThanOrEqualTo(BidIncrement());
        }

        protected override IEnumerable<object>
                                GetAttributesToIncludeInEqualityCheck()
        {
            return new List<Object>() {Amount};
        }
    }
}
```

The `WinningBid`, yet another value object, represents a bidder's accepted offer. Create the `WinningBid` class within the `Model` folder with Listing 21-38.

LISTING 21-38: The Winning Bid Value Object

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class WinningBid : ValueObject<WinningBid>
         public WinningBid(Guid bidder, Money maximumBid,
                           Money bid, DateTime timeOfBid)
```

continues

LISTING 21-38 *(continued)*

```
{
    if (bidder == Guid.Empty)
        throw new ArgumentNullException("Bidder cannot be null");

    if (maximumBid == null)
        throw new ArgumentNullException("MaximumBid cannot be null");

    if (timeOfBid == DateTime.MinValue)
        throw new ArgumentNullException("TimeOfBid must have a value");

    Bidder = bidder;
    MaximumBid = maximumBid;
    TimeOfBid = timeOfBid;
    CurrentAuctionPrice = new Price(bid);
}

public Guid Bidder { get; private set; }
public Money MaximumBid { get; private set; }
public DateTime TimeOfBid { get; private set; }
public Price CurrentAuctionPrice { get; private set; }

public WinningBid RaiseMaximumBidTo(Money newAmount)
{
    if (newAmount.IsGreaterThan(MaximumBid))
        return new WinningBid(Bidder, newAmount,
                              CurrentAuctionPrice.Amount,
                              DateTime.Now);
    else
        throw new ApplicationException(
     "Maximum bid increase must be larger than current maximum bid.");
}

public bool WasMadeBy(Guid bidder)
{
    return Bidder.Equals(bidder);
}

public bool CanBeExceededBy(Money offer)
{
    return CurrentAuctionPrice.CanBeExceededBy(offer);
}

public bool HasNotReachedMaximumBid()
{
    return MaximumBid.IsGreaterThan(CurrentAuctionPrice.Amount);
}

protected override IEnumerable<object>
                GetAttributesToIncludeInEqualityCheck()
{
    return new List<Object>() { Bidder, MaximumBid,
                                TimeOfBid, CurrentAuctionPrice };
```

```
        }
    }
}
```

The domain service `AutomaticBidder` bids on a member's behalf up to his maximum bid if he is outbid. It bids only as much as needed to remain the highest bidder, up to the bidder's maximum amount. Create the `AutomaticBidder` class in the `Model` folder with Listing 21-39.

LISTING 21-39: The Automatic Bidder Domain Service

```
using System.Collections.Generic;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class AutomaticBidder
    {
        public IEnumerable<WinningBid> GenerateNextSequenceOfBidsAfter(
                                Offer offer, WinningBid currentWinningBid)
        {
            var bids = new List<WinningBid>();

            if (currentWinningBid.MaximumBid
                        .IsGreaterThanOrEqualTo(offer.MaximumBid))
            {
                var bidFromOffer = new WinningBid(offer.Bidder,
                                                  offer.MaximumBid,
                                                  offer.MaximumBid,
                                                  offer.TimeOfOffer);
                bids.Add(bidFromOffer);

                bids.Add(CalculateNextBid(bidFromOffer,
                         new Offer(currentWinningBid.Bidder,
                                   currentWinningBid.MaximumBid,
                                   currentWinningBid.TimeOfBid)));
            }
            else
            {
                if (currentWinningBid.HasNotReachedMaximumBid())
                {
                    var currentBiddersLastBid =
                           new WinningBid(currentWinningBid.Bidder,
                                          currentWinningBid.MaximumBid,
                                          currentWinningBid.MaximumBid,
                                          currentWinningBid.TimeOfBid);
                    bids.Add(currentBiddersLastBid);

                    bids.Add(CalculateNextBid(currentBiddersLastBid, offer));
                }
                else
                    bids.Add(new WinningBid(offer.Bidder,
                     currentWinningBid.CurrentAuctionPrice.BidIncrement(),
                     offer.MaximumBid, offer.TimeOfOffer));
```

continues

LISTING 21-39 *(continued)*

```
            }

            return bids;
        }

        private WinningBid CalculateNextBid(WinningBid winningbid, Offer offer)
        {
            WinningBid bid;

            if (winningbid.CanBeExceededBy(offer.MaximumBid))
                bid = new WinningBid(offer.Bidder, offer.MaximumBid,
                                     winningbid.CurrentAuctionPrice.BidIncrement(),
                                     offer.TimeOfOffer);
            else
                bid = new WinningBid(offer.Bidder, offer.MaximumBid,
                                         offer.MaximumBid, offer.TimeOfOffer);

            return bid;
        }
    }
}
```

`BidPlaced` is a domain event signifying that a bid was placed. Add the class to the `Model` folder using Listing 21-40.

LISTING 21-40: The Bid Place Value Object

```
using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class BidPlaced
    {
        public BidPlaced(Guid auctionId, Guid bidderId,
                         Money amountBid, DateTime timeOfBid)
        {
            if (auctionId == Guid.Empty)
                throw new ArgumentNullException("Auction Id cannot be null");

            if (bidderId == Guid.Empty)
                throw new ArgumentNullException("Bidder Id cannot be null");

            if (amountBid == null)
                throw new ArgumentNullException("AmountBid cannot be null");

            if (timeOfBid == DateTime.MinValue)
                throw new ArgumentNullException("TimeOfBid must have a value");

            AuctionId = auctionId;
            Bidder = bidderId;
```

```
            AmountBid = amountBid;
            TimeOfMemberBid = timeOfBid;
        }

        public Guid AuctionId { get; private set; }
        public Guid Bidder { get; private set; }
        public Money AmountBid { get; private set; }
        public DateTime TimeOfMemberBid { get; private set; }
    }
}
```

The `OutBid` class is another domain event. It is raised if a bidder is outbid, as shown in Listing 21-41.

LISTING 21-41: The Outbid Value Object

```
using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class OutBid
    {
        public OutBid(Guid auctionId, Guid bidderId)
        {
            if (auctionId == Guid.Empty)
                throw new ArgumentNullException("Auction Id cannot be null");

            if (bidderId == Guid.Empty)
                throw new ArgumentNullException("Bidder Id cannot be null");

            AuctionId = auctionId;
            Bidder = bidderId;
        }

        public Guid AuctionId { get; private set; }
        public Guid Bidder { get; private set; }
    }
}
```

The `Auction` shown in Listing 21-42 is an entity and the aggregate root. The `Auction` class has no public properties and only exposes a single method to enable bids to be placed. The results of bids placed are raised as domain events.

LISTING 21-42: The Auction Entity

```
using System;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class Auction : Entity<Guid>
    {
```

continues

LISTING 21-42 *(continued)*

```
public Auction(Guid id, Money startingPrice, DateTime endsAt)
{
    if (id == Guid.Empty)
        throw new ArgumentNullException("Auction Id cannot be null");

    if (startingPrice == null)
        throw new ArgumentNullException(
               "Starting Price cannot be null");

    if (endsAt == DateTime.MinValue)
        throw new ArgumentNullException("EndsAt must have a value");

    Id = id;
    StartingPrice = startingPrice;
    EndsAt = endsAt;
}

private Money StartingPrice { get; set; }
private WinningBid WinningBid { get; set; }
private DateTime EndsAt { get; set; }

private bool StillInProgress(DateTime currentTime)
{
    return (EndsAt > currentTime);
}

public void PlaceBidFor(Offer offer, DateTime currentTime)
{
    if (StillInProgress(currentTime))
    {
        if (FirstOffer())
            PlaceABidForTheFirst(offer);
        else if (BidderIsIncreasingMaximumBidToNew(offer))
            WinningBid =
                WinningBid.RaiseMaximumBidTo(offer.MaximumBid);
        else if (WinningBid.CanBeExceededBy(offer.MaximumBid))
        {
            var newBids = new AutomaticBidder()
                .GenerateNextSequenceOfBidsAfter(offer, WinningBid);

            foreach (var bid in newBids)
                Place(bid);
        }
    }
}

private bool BidderIsIncreasingMaximumBidToNew(Offer offer)
{
    return WinningBid.WasMadeBy(offer.Bidder) && offer.MaximumBid
                                .IsGreaterThan(WinningBid.MaximumBid);
}

private bool FirstOffer()
```

```
        {
            return WinningBid == null;
        }

        private void PlaceABidForTheFirst(Offer offer)
        {
            if (offer.MaximumBid.IsGreaterThanOrEqualTo(StartingPrice))
                Place(new WinningBid(offer.Bidder, offer.MaximumBid,
                                     StartingPrice, offer.TimeOfOffer));
        }

        private void Place(WinningBid newBid)
        {
            if (!FirstOffer() && WinningBid.WasMadeBy(newBid.Bidder))
                DomainEvents.Raise(new OutBid(Id, WinningBid.Bidder));

            WinningBid = newBid;
            DomainEvents.Raise(new BidPlaced(Id, newBid.Bidder,
                               newBid.CurrentAuctionPrice.Amount,
                               newBid.TimeOfBid));
        }
    }
}
```

Notice that the `AutomaticBidder` class is instantiated directly within the `Auction` class as opposed to being injected as a dependency. The dependency injection pattern is useful when there is a selection of multiple implementations of a given dependency interface, but because there's only one implementation of the `AutomaticBidder` and there is no need to mock or stub the domain service, you can safely instantiate it directly.

The repository contract for the `Auction` aggregate has only two methods: one to find an auction by ID and one to add an auction. NHibernate can track changes, so the repositories can act like collection interfaces; there is no need to add an explicit save method to the repository. Add the contract definition in Listing 21-43 to the `Auction` folder. This is implemented within the `infrastructure` folder.

LISTING 21-43: The Auction Repository Interface

```
using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public interface IAuctionRepository
    {
        void Add(Auction auction);
        Auction FindBy(Guid Id);
    }
}
```

The second and final aggregate is really only a collection of value objects that represent the bidding history of an auction. Using Listing 21-44, create a new value object named `Bid` within the `BidHistory` folder.

LISTING 21-44: The Bid Value Object

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Model.Auction;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory
{
    public class Bid : ValueObject<Bid>
    {
        public Bid(Guid auctionId, Guid bidderId, Money amountBid,
                   DateTime timeOfBid)
        {
            if (auctionId == Guid.Empty)
                throw new ArgumentNullException("Auction Id cannot be null");

            if (bidderId == Guid.Empty)
                throw new ArgumentNullException("Bidder Id cannot be null");

            if (amountBid == null)
                throw new ArgumentNullException("AmountBid cannot be null");

            if (timeOfBid == DateTime.MinValue)
                throw new ArgumentNullException("TimeOfBid must have a value");

            AuctionId = auctionId;
            Bidder = bidderId;
            AmountBid = amountBid;
            TimeOfBid = timeOfBid;
        }

        public Guid AuctionId { get; private set; }
        public Guid Bidder { get; private set; }
        public Money AmountBid {get; private set;}
        public DateTime TimeOfBid { get; private set; }

        protected override IEnumerable<object>
                 GetAttributesToIncludeInEqualityCheck()
        {
            return new List<Object>() { Bidder, AuctionId, TimeOfBid, AmountBid };
        }
    }
}
```

The only other item for this aggregate is the repository contract, as shown in listing 21-45. The repository exposes a method to return the number of bids placed against an auction and has a method to persist new bids.

LISTING 21-45: The Bid History Repository Interface

```
using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory
```

```
{
    public interface IBidHistoryRepository
    {
        int NoOfBidsFor(Guid autionId);
        void Add(Bid bid);
    }
}
```

The domain model of the application is now complete. Your solution should look like Figure 21-11.

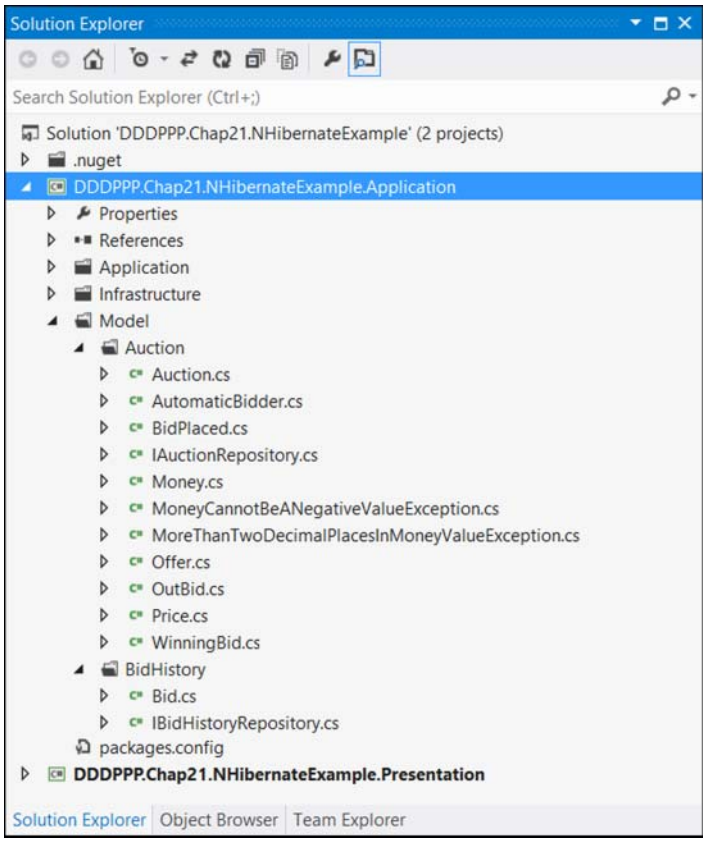


FIGURE 21-11: The Visual Studio solution structure

Application Service

With the model complete, you will now build the application service layer, which is the client of the domain model. Within the `Application` folder, add two folders:

- `BusinessUseCases`
- `Queries`

The `BusinessUseCases` folder contains all the features and use cases of the application, whereas the `Queries` folder contains all the queries required to report on the state of the domain model.

Creating an auction requires a starting price and an end date. Clients of the application service use a data transfer object (DTO) to carry the information to create an auction. Using Listing 21-46, create the DTO within the `BusinessUseCases` folder.

LISTING 21-46: The New Auction Request Command

```
using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Application.BusinessUseCases
{
    public class NewAuctionRequest
    {
        public decimal StartingPrice { get; set; }
        public DateTime EndsAt { get; set; }
    }
}
```

The application service that handles the request is detailed in Listing 21-47. The `CreateAuction` application service simply creates a new auction and adds it to the repository. The only thing of note is that you are wrapping the action within an NHibernate transaction. The application service is able to obtain NHibernate's unit of work pattern, the `ISession` variable, via the constructor. As you will see later, the `ISession` that is passed through the constructor is the same as what's used in the repository implementation this is the relationship between the unit of work and the repository within the application service. This ensures that the application service layer controls when changes are committed to domain objects.

LISTING 21-47: The Create an Auction Application Service

```
using System;
using DDDPPP.Chap21.NHibernateExample.Application.Model.Auction;
using NHibernate;

namespace DDDPPP.Chap21.NHibernateExample.Application.Application.BusinessUseCases
{
    public class CreateAuction
    {
        private IAuctionRepository _auctionRepository;
        private ISession _unitOfWork;

        public CreateAuction(IAuctionRepository auctionRepository,
                             ISession unitOfWork)
        {
            _auctionRepository = auctionRepository;
            _unitOfWork = unitOfWork;
        }

        public Guid Create(NewAuctionRequest command)
        {
            var auctionId = Guid.NewGuid();
```

```
            var startingPrice = new Money(command.StartingPrice);

            using (ITransaction transaction = _unitOfWork.BeginTransaction())
            {
                _auctionRepository.Add(new Auction(auctionId,
                                       startingPrice, command.EndsAt));

                transaction.Commit();
            }

            return auctionId;
        }
    }
}
```

The `ISession` is the main interface that persists and retrieves business entities and can be thought of as NHibernate's gateway to the database. The NHibernate site defines `ISession` as the "persistence manager." The `ISession` is NHibernate's unit of work implementation. Because the `ISession` interface implements the unit of work pattern discussed earlier in this chapter, no changes occur until a transaction is committed. Another pattern built into NHibernate is identity map, which maintains a single instance of a business entity in the `ISession` no matter how many times you retrieve it.

Time is an important concept in the auction domain, and bids should only be able to be placed while the auction is still active. As always, you never want to tie your domain objects to the system clock because it makes testing difficult. So with this in mind, you will abstract the concept of a clock by defining an interface named `IClock` within the `Infrastructure` folder, as shown in Listing 21-48.

LISTING 21-48: The Interface for the Clock Class

```
using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    public interface IClock
    {
        DateTime Time();
    }
}
```

For the application, you will use the system clock. Create an implementation of the `IClock` interface, again within the `Infrastructure` folder, using Listing 21-49.

LISTING 21-49: The Implementation of the Clock Interface

```
using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
```

continues

LISTING 21-49 *(continued)*

```
{
   public class SystemClock : IClock
    {
        public DateTime Time()
        {
            return DateTime.Now;
        }
    }
}
```

The second business use case that you will create an application service for is the task of bidding on an auction. Using Listing 21-50, create a class named `BidOnAuction`.

LISTING 21-50: The Bid On Auction Command

```
using System;
using DDDPPP.Chap21.NHibernateExample.Application.Model.Auction;
using DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory;
using NHibernate;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Application.BusinessUseCases
{
    public class BidOnAuction
    {
        private IAuctionRepository _auctionRepository;
        private IBidHistoryRepository _bidHistoryRepository;
        private ISession _unitOfWork;
        private IClock _clock;

        public BidOnAuction(IAuctionRepository auctionRepository,
                            IBidHistoryRepository bidHistoryRepository,
                            ISession unitOfWork, IClock clock)
        {
            _auctionRepository = auctionRepository;
            _bidHistoryRepository = bidHistoryRepository;
            _unitOfWork = unitOfWork;
            _clock = clock;
        }

        public void Bid(Guid auctionId, Guid memberId, decimal amount)
        {
            try
            {
                using (ITransaction transaction =
                              _unitOfWork.BeginTransaction())
                {
                    using (DomainEvents.Register(OutBid()))
                    using (DomainEvents.Register(BidPlaced()))
                    {
```

```
                    var auction = _auctionRepository.FindBy(auctionId);

                    var bidAmount = new Money(amount);

                    auction.PlaceBidFor(new Offer(memberId, bidAmount,
                                        _clock.Time()), _clock.Time());
                }

                transactionCommit();
            }
        }
        catch (StaleObjectStateException ex)
        {
            _unitOfWork.Clear();

            Bid(auctionId, memberId, amount);
        }
    }

    private Action<BidPlaced> BidPlaced()
    {
        return (BidPlaced e) =>
        {
            var bidEvent = new Bid(e.AuctionId, e.Bidder, e.AmountBid,
                                   e.TimeOfMemberBid);

            _bidHistoryRepository.Add(bidEvent);
        };
    }

    private Action<OutBid> OutBid()
    {
        return (OutBid e) =>
        {
            // E-mail customer to say that he has been outbid
        };
    }
  }
}
```

Again, you wrap the method call in an NHibernate transaction. This is important because a successful bid raises a domain event, which results in a bid being added to the `BidHistoryRepository`. The call to place a bid is also wrapped in a try catch with the `StaleObjectStateException`, resulting in the bid being placed again. The `StaleObjectStateException` is thrown if the auction is updated by another member between it being retrieved from the repository and the unit of work committing. In this exercise, you try again to place the bid with the refreshed auction to work against the latest version.

NHibernate Repository Implementation

With the model and application service layers complete, you can focus on the NHibernate repository implementation. The first thing you need to do is map the aggregates. Using Listing 21-51, add a new XML file to the `Infrastructure` folder named `Auction.hbm.xml`.

LISTING 21-51: NHibernate XML Mapping for the Auction Class

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    namespace="DDDPPP.Chap21.NHibernateExample.Application.Model.Auction"
        assembly="DDDPPP.Chap21.NHibernateExample.Application">

  <class name="Auction" table="Auctions" lazy="false" >

    <id name="Id" column="Id" type="Guid">
    </id>

    <version name="Version" column="Version" type="Int32" unsaved-value="0"/>

    <component name="StartingPrice" class="Money">
      <property name="Value" column="StartingPrice" not-null="true"/>
    </component>

    <property name="EndsAt" column="AuctionEnds" not-null="true"/>

    <component name="WinningBid" class="WinningBid">

      <property name="Bidder" column="BidderMemberId" not-null="false"/>

      <property name="TimeOfBid" column="TimeOfBid" not-null="false"/>

      <component name="MaximumBid" class="Money">
        <property name="Value" column="MaximumBid" not-null="false"/>
      </component>

      <component name="CurrentAuctionPrice" class="Price">
        <component name="Amount" class="Money">
          <property name="Value" column="CurrentPrice" not-null="false"/>
        </component>
      </component>
    </component>
  </class>
</hibernate-mapping>
```

This chapter doesn't go into detail about the syntax of these files because this is not a book on using NHibernate, but it should be easy to work out how NHibernate maps columns and tables to business entities and properties. For a deeper insight into the world of NHibernate, check out the many online resources or the book *NHibernate in Action*.

For NHibernate to pick up the mapping file, ensure that you change the Build Action property of the field to Embedded Resource, as shown in Figure 21-12.

To map a bid, add a second XML file to the `Infrastructure` folder named `Bid.hbm.xml` with Listing 21-52, again selecting Embedded Resource for the Build Action property of the file.

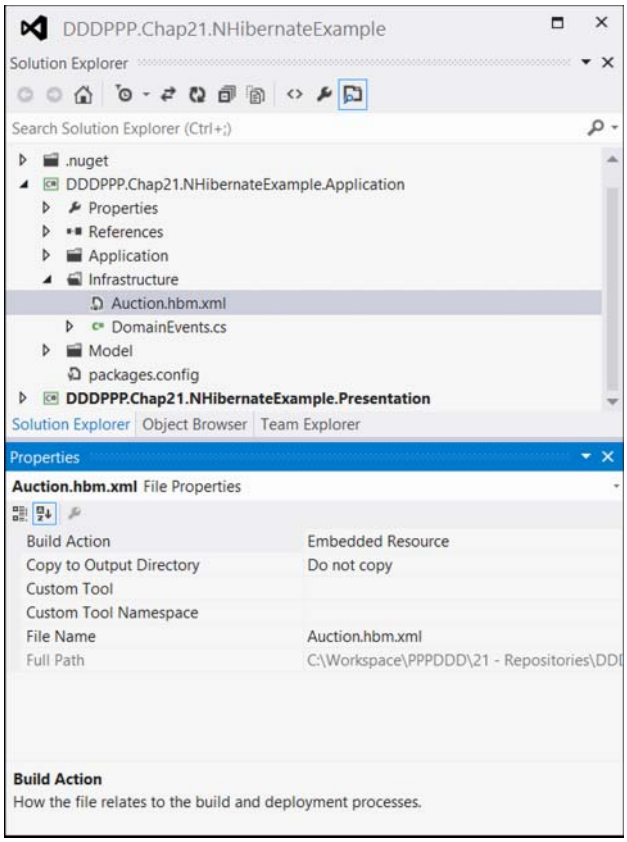


FIGURE 21-12: The build action property of the XML Mapping file

LISTING 21-52: NHibernate XML Mapping for the Bid History Class

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    namespace="DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory"
        assembly="DDDPPP.Chap21.NHibernateExample.Application">

  <class name="Bid" table="BidHistory" lazy="false" >

    <id name="Id" column="Id" type="Guid">
      <generator class="guid"/>
    </id>

    <property name="AuctionId" column="AuctionId" not-null="false"/>
    <property name="Bidder" column="BidderId" not-null="false"/>

    <component name="AmountBid"
```

continues

LISTING 21-52 *(continued)*

```
      class=
        "DDDPPP.Chap21.NHibernateExample.Application.Model.Auction.Money">
       <property name="Value" column="Bid" not-null="false"/>
     </component>

     <property name="TimeOfBid" column="TimeOfBid" not-null="false"/>
   </class>
</hibernate-mapping>
```

The repository implementations live under the infrastructure namespace. The first is the auction repository implementation, as defined in Listing 21-53.

LISTING 21-53: The Auction Repository Implementation

```
using System;
using DDDPPP.Chap21.NHibernateExample.Application.Model.Auction;
using NHibernate;
using NHibernate.Criterion;
using NHibernate.Transform;

namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    public class AuctionRepository : IAuctionRepository
    {
        private readonly ISession _session;

        public Auctions(ISession session)
        {
            _session = session;
        }

        public void Add(Auction auction)
        {
            _session.Save(auction);
        }

        public Auction FindBy(Guid Id)
        {
            return _session.Get<Auction>(Id);
        }
    }
}
```

As you can see, the `ISession` variable does all the work, which is why it's vital that the instance that is injected via the constructor is the same that is injected into the constructors of the application services. Otherwise, the application service can't control the unit of work.

The implementation for the `BidHistory` repository, shown in listing 21-54, is equally simple, with the only difference being the addition of a simple query to provide summary information on the number of bids for an auction.

LISTING 21-54: The Bid History Repository Implementation

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory;
using NHibernate;

namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    public class BidHistoryRepository : IBidHistoryRepository
    {
        private readonly ISession _session;

        public BidHistoryRepository(ISession session)
        {
            _session = session;
        }

        public int NoOfBidsFor(Guid autionId)
        {
            var sql = String.Format(
    "SELECT Count(*) FROM BidHistory Where AuctionId = '{0}'", autionId);
            var query = _session.CreateSQLQuery(sql);
            var result = query.UniqueResult();

            return Convert.ToInt32(result);
        }

        public void Add(BidEvent bid)
        {
            _session.Save(bid);
        }
    }
}
```

There are a few constraints to using NHibernate. One is that all objects being persisted need to have a parameterless constructor. Luckily, the constructor can be private, so this will have no effect on your model. This is a small price to pay for a persistence framework that allows your domain model to be POCO.

Update the `Auction` class and add a private constructor, as shown in Listing 21-55.

LISTING 21-55: The Auction Entity

```
namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class Auction : Entity<Guid>
    {
        private Auction() { }

        public Auction(Guid id, Money startingPrice, DateTime endsAt)
        {
```

continues

LISTING 21-55 *(continued)*

```
            Id = id;
            StartingPrice = startingPrice;
            EndsAt = endsAt;
        }

// .......
```

Update the `WinningBid` class and add a private constructor, as shown in Listing 21-56.

LISTING 21-56: The Winning Bid Value Object

```
namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class WinningBid : ValueObject<WinningBid>
    {
        private WinningBid() { }

        public WinningBid(Guid bidder, Money maximumBid, Money bid, DateTime
timeOfBid)
        {

//.......
```

Update the `Price` class and add a private constructor, as shown in Listing 21-57.

LISTING 21-57: The Price Value Object

```
namespace DDDPPP.Chap21.NHibernateExample.Application.Model.Auction
{
    public class Price : ValueObject<Price>
    {
        private Price() { }

        public Price(Money amount)
        {

//.......
```

You may have noticed in the mapping that you mapped an ID property for the `Bid` class. NHibernate needs to have an ID present for each object that it maps. Update the `Bid` class and add a private constructor and a new property to hold an ID that will be set by NHibernate, as shown in Listing 21-58.

LISTING 21-58: The Bid Value Object

```
namespace DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory
{
    public class Bid : ValueObject<Bid>
```

```
    {
        private Guid Id { get; set; }

        private Bid()
        { }

// .......
```

You may have also noticed that the mapping for the `Auction` entity had a version property. This property ensures that the data is not stale (it hasn't changed since pulling it from the database). When NHibernate persists the entity, it checks to ensure that the version is the same; if it's not, it throws a `StaleObjectStateException`. This exception is handled by the application service and in your auction domain; if this happens, try to apply the offer again.

Add a version property to the `Entity` class, as shown in Listing 21-59.

LISTING 21-59: The Entity Base Class

```
namespace DDDPPP.Chap21.NHibernateExample.Application.Infrastructure
{
    public abstract class Entity<TId>
    {
        public TId Id { get; protected set; }
        public int Version { get; private set; }
    }
}
```

Database Schema

For the database you can use the free MS SQL Express (`http://www.microsoft.com/web/platform/database.aspx`). You can probably work out the schema from the mapping; in fact, NHibernate can actually build the database from the mapping. However, here is the full database schema. Create a database named `AuctionExample` and run Listing 21-60 to set it up.

LISTING 21-60: The Database Schema

```
USE [AuctionExample]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[BidHistory](
[AuctionId] [uniqueidentifier] NOT NULL,
[BidderId] [uniqueidentifier] NOT NULL,
[Bid] [numeric](18, 2) NOT NULL,
[TimeOfBid] [datetime] NOT NULL,
```

continues

LISTING 21-60 *(continued)*

```
[Id] [uniqueidentifier] NOT NULL,
CONSTRAINT [PK_BidHistory] PRIMARY KEY CLUSTERED
(
[Id] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
       ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Auctions](
[Id] [uniqueidentifier] NOT NULL,
[StartingPrice] [decimal](18, 2) NOT NULL,
[BidderMemberId] [uniqueidentifier] NULL,
[TimeOfBid] [datetime] NULL,
[MaximumBid] [decimal](18, 2) NULL,
[CurrentPrice] [decimal](18, 2) NULL,
[AuctionEnds] [datetime] NOT NULL,
[Version] [int] NOT NULL,
CONSTRAINT [PK_Auctions] PRIMARY KEY CLUSTERED
(
[Id] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
                ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
GO
```

Your table schema will resemble Figure 21-13.

Querying

You will turn your attention to the query side of the application service. The query services report on the state of the auctions. The first view of the auction is a summary of its status. Instead of returning the domain object, you will use a simple view model. Using Listing 21-61, create the `AuctionStatus` class within the `Queries` folder.

LISTING 21-61: The Auction Status Class

```
using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Application.Queries
{
    public class AuctionStatus
    {
        public Guid Id { get; set; }
        public decimal CurrentPrice { get; set; }
        public DateTime AuctionEnds { get; set; }
```

```
        public Guid WinningBidderId { get; set; }
        public int NumberOfBids { get; set; }
        public TimeSpan TimeRemaining {get; set;}
    }
}
```

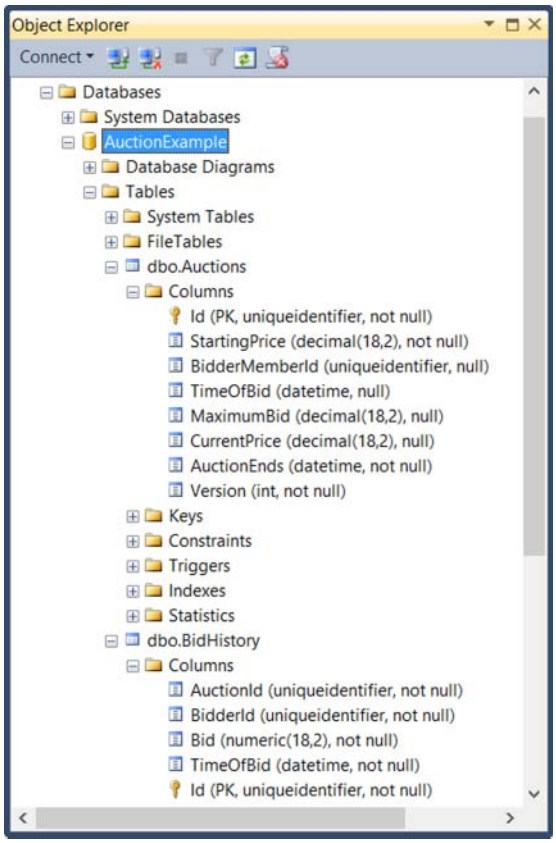


FIGURE 21-13: The database schema

The auction status query requires native SQL to be used because you have no public getters to transform the aggregate into an `AuctionStatus` DTO. However, this cleanly separates the domain model from the application's reporting needs and prevents the repository from having to deal with reporting concerns.

LISTING 21-62: The Auction Status Query

```
using System;
using NHibernate;
using NHibernate.Criterion;
using NHibernate.Transform;
```

continues

LISTING 21-62 *(continued)*

```
using DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;

namespace DDDPPP.Chap21.NHibernateExample.Application.Application.Queries
{
    public class AuctionStatusQuery
    {
        private readonly ISession _session;
        private readonly IBidHistoryRepository _bidHistory;
        private readonly IClock _clock;

        public AuctionStatusQuery(ISession session,
                                  IBidHistoryRepository bidHistory,
                                  IClock clock)
        {
            _session = session;
            _bidHistory = bidHistory;
            _clock = clock;
        }

        public AuctionStatus AuctionStatus(Guid auctionId)
        {
            var status = _session
                        .CreateSQLQuery(String.Format(
               "select Id, CurrentPrice, BidderMemberId as WinningBidderId, " +
               "AuctionEnds from Auctions Where Id = '{0}'", auctionId))
                        .SetResultTransformer(
                            Transformers.AliasToBean<AuctionStatus>())
.UniqueResult<AuctionStatus>();

            status.TimeRemaining = TimeRemaining(status.AuctionEnds);
            status.NumberOfBids = _bidHistory.NoOfBidsFor(auctionId);

            return status;
        }

        public TimeSpan TimeRemaining(DateTime AuctionEnds)
        {
            if (_clock.Time() < AuctionEnds)
                return AuctionEnds.Subtract(_clock.Time());
            else
                return new TimeSpan();
        }
    }
}
```

The other report that the application needs to give access to is the history of bids against the auction. Again, you don't want to expose your domain objects, so you will create a specific DTO, shown in Listing 21-63. Even though it closely resembles the real `Bid` domain object, the `BidInformation` class represents a completely different concern and will not be affected if the `Bid` value object evolves.

LISTING 21-63: Bid Information Data Transfer Object

```
using System;

namespace DDDPPP.Chap21.NHibernateExample.Application.Application.Queries
{
    public class BidInformation
    {
        public Guid Bidder { get;  set; }
        public decimal AmountBid { get;  set; }
        public string currency { get; set; }
        public DateTime TimeOfBid { get;  set; }
    }
}
```

Again, the query service, Listing 21-64, that pulls back information on the bids placed against an auction needs to go directly to the database to pull back a view because of the encapsulated domain model.

LISTING 21-64: The Bid History Query Service

```
using System;
using System.Collections.Generic;
using NHibernate;
using NHibernate.Criterion;
using NHibernate.Transform;

namespace DDDPPP.Chap21.NHibernateExample.Application.Application.Queries
{
    public class BidHistoryQuery
    {
        private readonly ISession _session;

        public BidHistoryQuery(ISession session)
        {
            _session = session;
        }

        public IEnumerable<BidInformation> BidHistoryFor(Guid auctionId)
        {
            var status = _session
                        .CreateSQLQuery(String.Format(
           "SELECT [BidderId] as Bidder,[Bid] as AmountBid ,TimeOfBid " +
           "FROM [BidHistory] " +
           "Where AuctionId = '{0}' "+
           "Order By Bid Desc, TimeOfBid Asc", auctionId))
            .SetResultTransformer(
                   Transformers.AliasToBean<BidInformation>());

            return status.List<BidInformation>();
        }
    }
}
```

Configuration

NHibernate needs a small piece of configuration to work. Update the `app.config` of the `Presentation` project so that it matches Listing 21-65. Then change the connection string depending on your database instance name.

LISTING 21-65: NHibernate Configuration File

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <configSections>
    <section name="hibernate-configuration"
         type="NHibernate.Cfg.ConfigurationSectionHandler, NHibernate" />
  </configSections>

  <startup>
      <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>

  <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
    <session-factory name="NHibernate.Test">
      <property name="connection.driver_class">
              NHibernate.Driver.SqlClientDriver</property>
      <property name="connection.connection_string">
        Data Source=.\SQLEXPRESS;Database=AuctionExample;Trusted_Connection=True;
      </property>
      <property name="adonet.batch_size">10</property>
      <property name="show_sql">false</property>
      <property name="dialect">NHibernate.Dialect.MsSql2005Dialect</property>
      <property name="command_timeout">60</property>
      <property name="query.substitutions">true 1, false 0, yes 'Y', no 'N'
      </property>
    </session-factory>
  </hibernate-configuration>
</configuration>
```

To tie all the dependencies together, you will utilize an inversion-of-control container. The container of choice will be StructureMap. Install StructureMap using the NuGet package manager just as you did with the NHibernate package, but this time ensure that StructureMap is referenced in both projects. Add a class named `Bootstrapper`, Listing 21-66, that wires the application service layer dependencies.

LISTING 21-66: Wiring Up Dependencies with Structure Map

```
using System;
using System.Collections.Generic;
using StructureMap;
using DDDPPP.Chap21.NHibernateExample.Application.Infrastructure;
using DDDPPP.Chap21.NHibernateExample.Application.Model.Auction;
using DDDPPP.Chap21.NHibernateExample.Application.Model.BidHistory;
using NHibernate;
```

```
using NHibernate.Cfg;

namespace DDDPPP.Chap21.NHibernateExample.Application
{
    public static class Bootstrapper
    {
        public static void Startup()
        {
            Configuration config = new Configuration();

            config.Configure();
            config.AddAssembly("DDDPPP.Chap21.NHibernateExample.Application");

            var sessionFactory = config.BuildSessionFactory();

            ObjectFactory.Initialize(structureMapConfig =>
            {

               structureMapConfig.For<IAuctionRepository>()
                                 .Use<AuctionRepository>();
                structureMapConfig.For<IBidHistoryRepository>()
                                 .Use<Infrastructure.BidHistoryRepository>();
                structureMapConfig.For<IClock>().Use<SystemClock>();
                structureMapConfig.For<ISessionFactory>().Use(sessionFactory);
                structureMapConfig.For<ISession>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use(x =>
                    {
                        var factory = x.GetInstance<ISessionFactory>();
                        return factory.OpenSession();
                    });
            });
        }
    }
}
```

The `ISessionFactory` is typically created as a singleton object because of the relatively expensive operation of creating it. One of the jobs of the `ISessionFactory` is to provide `ISession` instances. You use StructureMap to ensure that the same version of the `ISession` variable is used for all repositories and all application services during a process, meaning that the application service can control the unit of work.

Presentation

Finally, to show the application in action, you will simulate users bidding on an auction. Add Listing 21-67 to the `Program` file of the `Presentation` project.

LISTING 21-67: The Command Line Program

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.NHibernateExample.Application.Application.BusinessUseCases;
using DDDPPP.Chap21.NHibernateExample.Application.Application.Queries;
```

continues

LISTING 21-67 *(continued)*

```
using DDDPPP.Chap21.NHibernateExample.Application;
using StructureMap;

namespace DDDPPP.Chap21.NHibernateExample.Presentation
{
    public class Program
    {
       private static Dictionary<Guid, String> members
                            = new Dictionary<Guid, string>();

       public static void Main(string[] args)
        {
            Bootstrapper.Startup();

            var memberIdA = Guid.NewGuid();
            var memberIdB = Guid.NewGuid();

            members.Add(memberIdA, "Ted");
            members.Add(memberIdB, "Rob");

            var auctionId = CreateAuction();

            Bid(auctionId, memberIdA, 10m);
            Bid(auctionId, memberIdB, 1.49m);
            Bid(auctionId, memberIdB, 10.01m);
            Bid(auctionId, memberIdB, 12.00m);
            Bid(auctionId, memberIdA, 12.00m);
        }

       public static Guid CreateAuction()
       {
           var createAuctionService =
                        ObjectFactory.GetInstance<CreateAuction>();

           var newAuctionRequest = new NewAuctionRequest();

           newAuctionRequest.StartingPrice = 0.99m;
           newAuctionRequest.EndsAt = DateTime.Now.AddDays(1);

           var auctionId = createAuctionService.Create(newAuctionRequest);

           return auctionId;
       }

       public static void Bid(Guid auctionId, Guid memberId, decimal amount)
       {
           var bidOnAuctionService = ObjectFactory.GetInstance<BidOnAuction>();

           bidOnAuctionService.Bid(auctionId, memberId, amount);

           PrintStatusOfAuctionBy(auctionId);
           PrintBidHistoryOf(auctionId);
```

```
            Console.WriteLine("Hit any key to continue");
            Console.ReadLine();
        }

        public static void PrintStatusOfAuctionBy(Guid auctionId)
        {
            var auctionSummaryQuery =
                   ObjectFactory.GetInstance<AuctionStatusQuery>();
            var status = auctionSummaryQuery.AuctionStatus(auctionId);

            Console.WriteLine("No Of Bids: " + status.NumberOfBids);
            Console.WriteLine("Current Bid: " +
                              status.CurrentPrice.ToString("##.##"));
            Console.WriteLine("Winning Bidder: " +
                        FindNameOfBidderWith(status.WinningBidderId));
            Console.WriteLine("Time Remaining: " + status.TimeRemaining);
            Console.WriteLine();
        }

        public static void PrintBidHistoryOf(Guid auctionId)
        {
            var bidHistoryQuery = ObjectFactory.GetInstance<BidHistoryQuery>();
            var status = bidHistoryQuery.BidHistoryFor(auctionId);

            Console.WriteLine("Bids..");

            foreach (var bid in status)
                Console.WriteLine(FindNameOfBidderWith(bid.Bidder) +
                                  "\t - " +
                                  bid.AmountBid.ToString("G") +
                                  "\t at " + bid.TimeOfBid);
            Console.WriteLine("------------------------------");
            Console.WriteLine();
        }

        public static string FindNameOfBidderWith(Guid id)
        {
            if (members.ContainsKey(id))
                return members[id];
            else
                return string.Empty;
        }
    }
}
```

Figure 21-14 shows the program running.

RavenDB Example

RavenDB is a schemaless document database that stores domain objects as JSON (JavaScript Object Notation) documents. Therefore, there is no mismatch between the data and the domain model. In this example, you change the domain model so that the properties are public, but you keep the setters private.

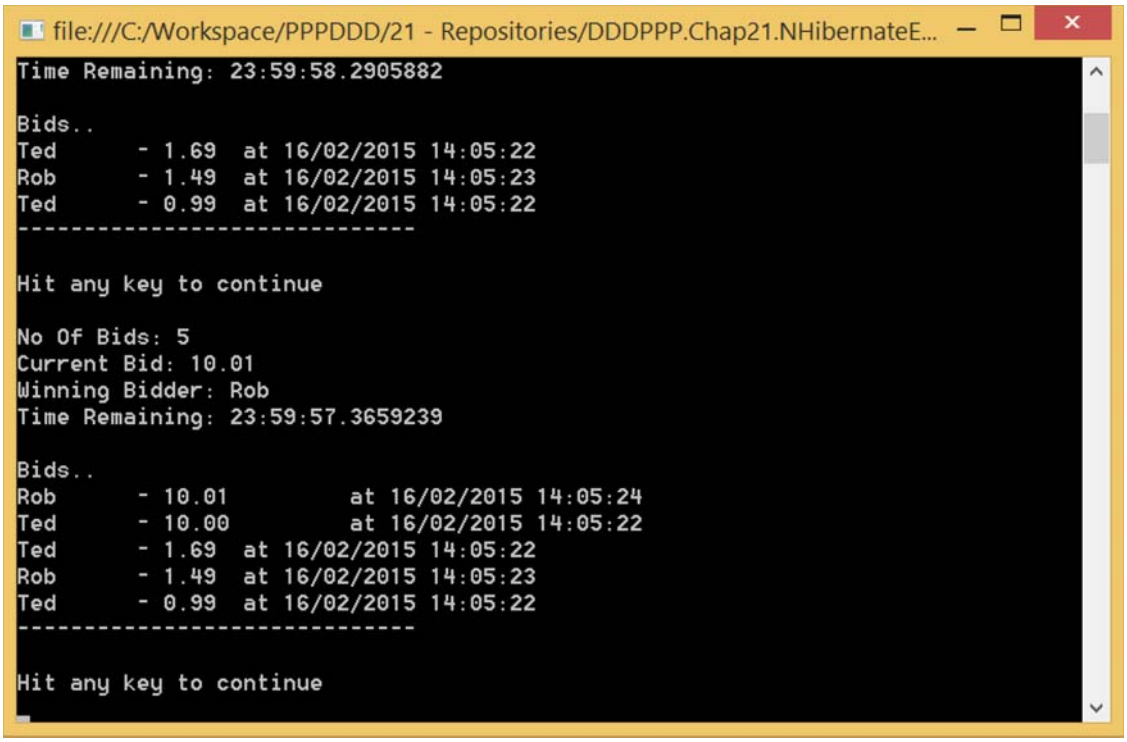


FIGURE 21-14: The running program

Solution Setup

Before you start with the Visual Studio solution, you need to install RavenDB. The easiest way to install RavenDB is to download the latest installer from RavenDB's website at `http://ravendb.net/download`. Every RavenDB server instance is manageable via a remotely accessible Silverlight application: the RavenDB Management Studio. After installing it, you can navigate to `http://localhost:8080/` to view the RavenDB Management Studio.

After installing RavenDB, create a new blank Visual Studio solution named `DDDPPP.Chap21.RavenDBExample` and add to this a class library named `DDDPPP.Chap21.RavenDBExample.Application` and a console application named `DDDPPP.Chap21.RavenDBExample.Presentation`. Within the `Presentation` project, add a reference to the application. Next, create the following folders in the `DDDPPP.Chap21.RavenDBExample.Application` project:

- `Application`
- `Infrastructure`
- `Model`

You can delete the `Class1.cs` that is automatically created because you won't be needing it. Use NuGet to install the RavenDB client libraries, as shown in Figure 21-15.

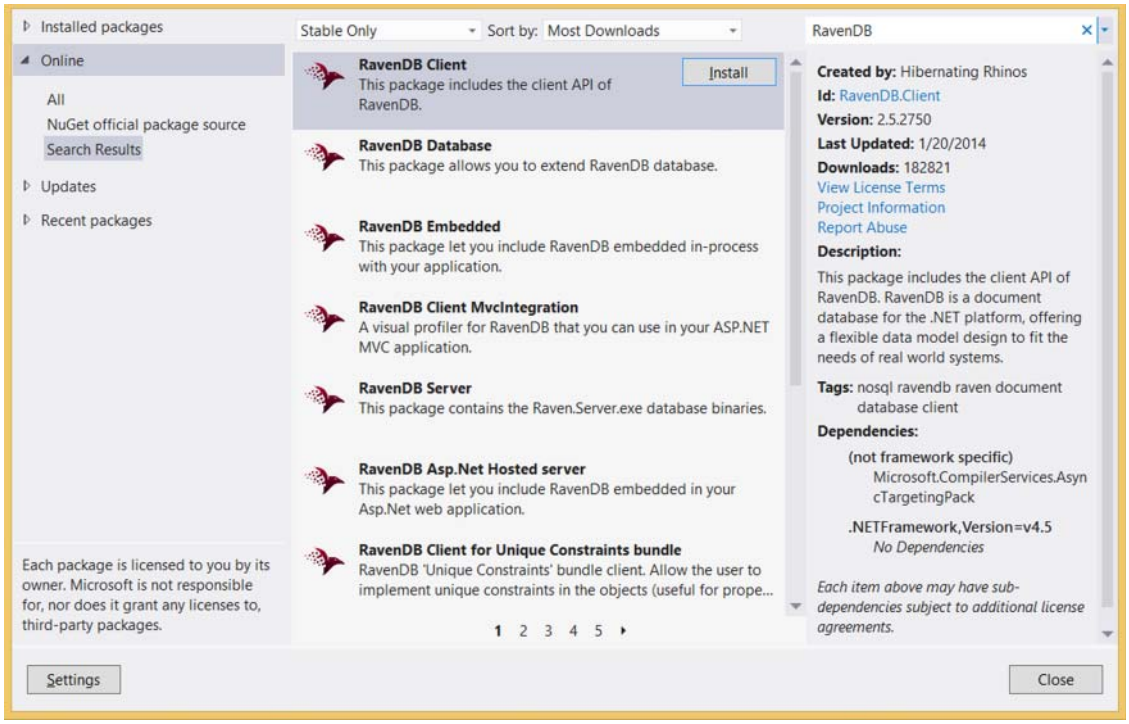


FIGURE 21-15: Install RavenDB client libraries via NuGet.

Build out the application in the same manner as you did for the NHibernate example, or simply copy the class files into this new solution, ensuring you update the namespaces. Remove the following classes that were specific to the NHibernate example:

- `BusinessUseCases\BidOnAuction.cs`
- `BusinessUseCases\CreateAuction.cs`
- `Queries\AuctionStatusQuery.cs`
- `Queries\BidHistoryQuery.cs`
- `Infrastructure\Auction.hbm.xml`
- `Infrastructure\Bid.hbm.xml`
- `Infrastructure\AuctionRepository.cs`
- `Infrastructure\BidHistoryRepository.cs`
- `Bootstrapper.cs`

You are left with a solution that resembles Figure 21-16.

The classes you removed are replaced with RavenDB's repository implementation.

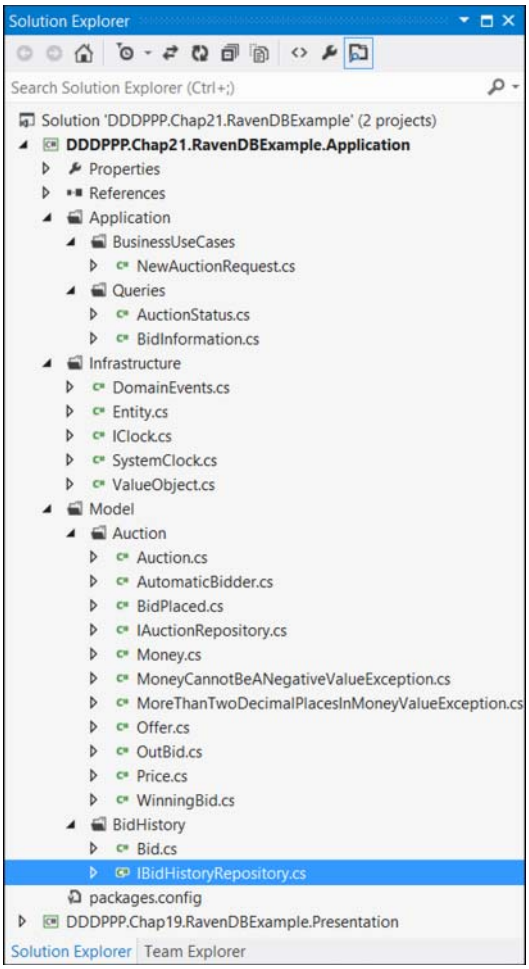


FIGURE 21-16: Solution explorer with an application based on the NHibernate sample.

Altering the Model

The model is nearly identical to the model that was built in the NHibernate example. It still requires a parameterless constructor in each of the domain objects. As you are modeling with public getters, you need to change the properties of the `Auction` class to match Listing 21-68.

LISTING 21-68: Modifiing the Properties of the Auction Class

```
using System;
using DDDPPP.Chap21.RavenDBExample.Application.Infrastructure;

namespace DDDPPP.Chap21.RavenDBExample.Application.Model.Auction
{
```

```
public class Auction : Entity<Guid>
{
    private Auction() { }

    ....

    public Money StartingPrice { get; private set; }
    public WinningBid WinningBid { get; private set; }
    public DateTime EndsAt { get; private set; }

    ...

    public bool HasBeenBidOn()
    {
        return WinningBid == null;
    }

    ....
}
```

There is also a public method on the `Auction` class named `HasBeenBidOn` that will be used within the application service query method to report on the state of the auction. The only other change is to make the getter `Value` property of the `Money` class public so you can report on it, as shown in Listing 21-69.

LISTING 21-69: Modifiying the Money Value Property

```
namespace DDDPPP.Chap21.RavenDBExample.Application.Model.Auction
{
    public class Money
    {
        public decimal Value { get; private set; }
```

An additional class that is required for the RavenDB example is the inclusion of the `BidHistory` domain object. This object contains all the bids placed against an auction in the order they were placed. Add this class to the `BidHistory` folder of the domain model with Listing 21-70.

LISTING 21-70: The Bid History Class

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory
{
    public class BidHistory
    {
        private IEnumerable<Bid> _bids;

        public BidHistory(IEnumerable<Bid> bids)
        {
              if (bids == null)
```

continues

LISTING 21-70 *(continued)*

```
                throw new ArgumentNullException("Bids cannot be null");

            _bids = bids;
        }

        public IEnumerable<Bid> ShowAllBids()
        {
            var bids = _bids.OrderByDescending(x => x.AmountBid)
                            .ThenBy(x => x.TimeOfBid);

            return bids;
        }
    }
}
```

To retrieve the `BidHistory` object, you need to add a new method to the `IBidHistoryRepository` contract, as shown in Listing 21-71.

LISTING 21-71: Adding a New Method to the Bid History Repository

```
using System;

namespace DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory
{
    public interface IBidHistoryRepository
    {
        int NoOfBidsFor(Guid autionId);
        void Add(Bid bid);
        BidHistory FindBy(Guid auctionId);
    }
}
```

Application Service

The application service only differs from the NHibernate solution because it uses RavenDB's `IDocumentSession` type as opposed to NHibernate's `ISession`. There is also a different exception that is thrown if concurrency is broken. RavenDB supports implicit transactions, which means they are built in to the `IDocumentSession`. You don't need to explicitly wrap the call in a transaction, as shown in Listing 21-72.

LISTING 21-72: Updating the Bid On Auction Application Service Class

```
using System;
using DDDPPP.Chap21.RavenDBExample.Application.Model.Auction;
using DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory;
using Raven.Client;
using Raven.Abstractions.Exceptions;
using DDDPPP.Chap21.RavenDBExample.Application.Infrastructure;

namespace DDDPPP.Chap21.RavenDBExample.Application.Application.BusinessUseCases
```

```
{
    public class BidOnAuction
    {
        private IAuctionRepository _auctions;
        private IBidHistoryRepository _bidHistory;
        private IDocumentSession _unitOfWork;
        private IClock _clock;

        public BidOnAuction(IAuctionRepository auctions,
                            IBidHistoryRepository bidHistory,
                            IDocumentSession unitOfWork,
                            IClock clock)
        {
            _auctions = auctions;
            _bidHistory = bidHistory;
            _unitOfWork = unitOfWork;
            _clock = clock;
        }

        public void Bid(Guid auctionId, Guid memberId, decimal amount)
        {
            try
            {
                using (DomainEvents.Register(OutBid()))
                using (DomainEvents.Register(BidPlaced()))
                {
                    var auction = _auctions.FindBy(auctionId);

                    var bidAmount = new Money(amount);

                    auction.PlaceBidFor(new Offer(memberId, bidAmount,
                                                  _clock.Time()), _clock.Time());
                }

                _unitOfWork.SaveChanges();
            }
            catch (ConcurrencyException ex)
            {
                _unitOfWork.Advanced.Clear();
                Bid(auctionId, memberId, amount);
            }
        }

        private Action<BidPlaced> BidPlaced()
        {
            return (BidPlaced e) =>
            {
                var bidEvent = new Bid(e.AuctionId, e.Bidder,
                                       e.AmountBid, e.TimeOfBid);

                _bidHistory.Add(bidEvent);
            };
        }

        private Action<OutBid> OutBid()
        {
```

continues

LISTING 21-72 *(continued)*

```
            return (OutBid e) =>
            {
                // E-mail customer to say that he has been outbid
            };
        }
    }
}
```

You don't use the `Version` property on the `Entity` base class with RavenDB because it already has a version tag built in by default. When loading a document from RavenDB, it caches the Etag that relates to it. The Etag is basically the version stamp. When the call to commit the session is made, RavenDB checks to see if the Etag has been updated since retrieving the document. If it has, the `ConcurrencyException` is thrown.

The `CreateAuction` application service is simple, and it's similar to the NHibernate implementation, as displayed in Listing 21-73.

LISTING 21-73: RavenDB Create Auction Application Service

```
using System;
using DDDPPP.Chap21.RavenDBExample.Application.Model.Auction;
using DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory;
using Raven.Client;
using DDDPPP.Chap21.RavenDBExample.Application.Infrastructure;

namespace DDDPPP.Chap21.RavenDBExample.Application.Application.BusinessUseCases
{
    public class CreateAuction
    {
        private IAuctionRepository _auctions;
        private IDocumentSession _unitOfWork;

        public CreateAuction(IAuctionRepository auctions,
                             IDocumentSession unitOfWork)
        {
            _auctions = auctions;
            _unitOfWork = unitOfWork;
        }

        public Guid Create(NewAuctionRequest command)
        {
            var auctionId = Guid.NewGuid();
            var startingPrice = new Money(command.StartingPrice);

            _auctions.Add(new Auction(auctionId, startingPrice,
                                      command.EndsAt));

            _unitOfWork.SaveChanges();

            return auctionId;
```

```
        }
    }
}
```

Querying

Because you now have public properties on the `Auction` entity, you can transform them into the DTOs that the query services return. First create the `AuctionStatusQuery` with Listing 21-74.

LISTING 21-74: The Auction Status Query Class

```
using System;
using System.Collections.Generic;
using Raven.Client;
using DDDPPP.Chap21.RavenDBExample.Application.Model.Auction;
using DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory;
using DDDPPP.Chap21.RavenDBExample.Application.Infrastructure;

namespace DDDPPP.Chap21.RavenDBExample.Application.Application.Queries
{
    public class AuctionStatusQuery
    {
        private readonly IAuctionRepository _auctions;
        private readonly IBidHistoryRepository _bidHistory;
        private readonly IClock _clock;

        public AuctionStatusQuery(IAuctionRepository auctions,
                                  IBidHistoryRepository bidHistory,
                                  IClock clock)
        {
            _auctions = auctions;
            _bidHistory = bidHistory;
            _clock = clock;
        }

        public AuctionStatus AuctionStatus(Guid auctionId)
        {
            var auction = _auctions.FindBy(auctionId);

            var status = new AuctionStatus();

            status.AuctionEnds = auction.EndsAt;
            status.Id = auction.Id;

            if (auction.HasBeenBidOn())
            {
                status.CurrentPrice =
                    auction.WinningBid.CurrentAuctionPrice.Amount.Value;
                status.WinningBidderId = auction.WinningBid.Bidder;
            }

            status.TimeRemaining = TimeRemaining(auction.EndsAt);
```

continues

LISTING 21-74 *(continued)*

```
            status.NumberOfBids = _bidHistory.NoOfBidsFor(auctionId);

            return status;
        }

        public TimeSpan TimeRemaining(DateTime AuctionEnds)
        {
            if (_clock.Time() < AuctionEnds)
                return AuctionEnds.Subtract(_clock.Time());
            else
                return new TimeSpan();
        }
    }
}
```

The query to return the bids for a given auction uses the `BidHistory` object to ensure the ordering of the bids is correct because this is domain logic. Add the `BidHistoryQuery` class to the solution with Listing 21-75.

LISTING 21-75: The Bid History Query Class

```
using System;
using System.Collections.Generic;
using Raven.Client;
using DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory;

namespace DDDPPP.Chap21.RavenDBExample.Application.Application.Queries
{
    public class BidHistoryQuery
    {
        private readonly IBidHistoryRepository _bidHistory;

        public BidHistoryQuery(IBidHistoryRepository bidHistory)
        {
            _bidHistory = bidHistory
        }

        public IEnumerable<BidInformation> BidHistoryFor(Guid auctionId)
        {
            var bidHistory = _bidHistory.FindBy(auctionId);

            return Convert(bidHistory.ShowAllBids());
        }

        public IEnumerable<BidInformation> Convert(IEnumerable<Bid> bids)
        {
            var bidInfo = new List<BidInformation>();

            foreach (var bid in bids)
            {
                bidInfo.Add(new BidInformation() { Bidder = bid.Bidder,
```

```
                                        AmountBid = bid.AmountBid.Value,
                                            TimeOfBid = bid.TimeOfBid });
            }

            return bidInfo;
        }
    }
}
```

RavenDB Repository Implementation

As you will remember, the `IBidHistoryRepository` contract exposes a count of the number of bids against an auction. To support this, you create an index to improve querying for this data. Add the index class `BidHistory _ NumberOfBids` with Listing 21-76 to the `Infrastructure` folder.

LISTING 21-76: Bid History Number of the Bid Index

```
using System;
using System.Collections.Generic;
using System.Linq;
using Raven.Client.Indexes;
using Raven.Client.Document;
using Raven.Abstractions.Indexing;
using DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory;
using DDDPPP.Chap21.RavenDBExample.Application.Application;

namespace DDDPPP.Chap21.RavenDBExample.Application.Infrastructure
{
    public class BidHistory_NumberOfBids : AbstractIndexCreationTask<Bid,
                                        BidHistory_NumberOfBids.ReduceResult>
    {
        public class ReduceResult
        {
            public Guid AuctionId { get; set; }
            public int Count { get; set; }
        }

        public BidHistory_NumberOfBids()
        {
            Map = bids => from bid in bids
                  select new {bid.AuctionId, Count = 1};

            Reduce = results => from result in results
                                group result by result.AuctionId into g
                                select new { AuctionId = g.Key,
                                        Count = g.Sum(x => x.Count) };
        }
    }
}
```

The `BidHistoryRepository` implementation, Listing 21-77, is similar to the NHibernate implementation apart from the use of the index created earlier to return the number of bids placed against an auction. Notice that in the query, you are calling the customization

`WaitForNonStaleResultsAsOfNow` because RavenDB is eventually consistent and you are blocking the thread to ensure the index is up to date.

LISTING 21-77: The RavenDB Bid History Repository Implementation

```
using System;
using System.Collections.Generic;
using System.Linq;
using DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory;
using Raven.Client;

namespace DDDPPP.Chap21.RavenDBExample.Application.Infrastructure
{
    public class BidHistoryRepository : IBidHistoryRepository
    {
        private readonly IDocumentSession _documentSession;

        public BidHistoryRepository(IDocumentSession documentSession)
        {
            _documentSession = documentSession;
        }

        public int NoOfBidsFor(Guid autionId)
        {

            var count = _documentSession.Query<BidHistory_NumberOfBids.ReduceResult,
                                              BidHistory_NumberOfBids>()
                                .Customize(x => x.WaitForNonStaleResultsAsOfNow())
                                .FirstOrDefault(x => x.AuctionId == autionId)
                                    ?? new BidHistory_NumberOfBids.ReduceResult();

            return count.Count;
        }

        public void Add(Bid bid)
        {
            _documentSession.Store(bid);
        }

        public BidHistory FindBy(Guid auctionId)
        {
            var bids = _documentSession.Query<Bid>()
                                .Customize(x =>
                                      x.WaitForNonStaleResultsAsOfNow())
                                .Where(x => x.AuctionId == auctionId)
                                .ToList();

            return new BidHistory(bids);
        }
    }
}
```

The implementation of the `IAuctionRepository` is straightforward and is shown in Listing 21-78.

LISTING 21-78: RavenDBs Auction Repository Implementation

```
using System;
using DDDPPP.Chap21.RavenDBExample.Application.Model.Auction;
using Raven.Client;
using DDDPPP.Chap21.RavenDBExample.Application.Application.Queries;

namespace DDDPPP.Chap21.RavenDBExample.Application.Infrastructure
{
    public class AuctionRepository : IAuctionRepository
    {
        private readonly IDocumentSession _documentSession;

        public AuctionRepository(IDocumentSession documentSession)
        {
            _documentSession = documentSession;
        }

        public void Add(Auction auction)
        {
            _documentSession.Store(auction);
        }

        public Auction FindBy(Guid Id)
        {
            return _documentSession.Load<Auction>("Auctions/" + Id);
        }
    }
}
```

Configuration

You will use StructureMap to wire up the dependencies of your application service. Install the StructureMap libraries, as you did for the NHibernate example, using NuGet. Then add the following `Bootstrappper` class, Listing 21-79, to the route of the application class library project.

LISTING 21-79: The StructureMap Configuration Bootstrapper Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using Raven.Client;
using Raven.Client.Document;
using Raven.Client.Extensions;
using Raven.Client.Indexes;
using StructureMap;
using DDDPPP.Chap21.RavenDBExample.Application.Infrastructure;
using DDDPPP.Chap21.RavenDBExample.Application.Model.Auction;
using DDDPPP.Chap21.RavenDBExample.Application.Model.BidHistory;

namespace DDDPPP.Chap21.RavenDBExample.Application
{
```

continues

LISTING 21-79 *(continued)*

```
    public static class Bootstrapper
    {
        public static void Startup()
        {
            var documentStore = new DocumentStore
            {
                ConnectionStringName = "RavenDB"
            }.Initialize();

            documentStore.DatabaseCommands
                        .EnsureDatabaseExists("RepositoryExample");

            ObjectFactory.Initialize(config =>
            {
                config.For<IAuctionRepository>().Use<AuctionRepository>();
                config.For<IBidHistoryRepository>()
                                                .Use<BidHistoryRepository>();
                config.For<IClock>().Use<SystemClock>();

                config.For<IDocumentStore>().Use(documentStore);
                config.For<IDocumentSession>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use(x =>
                    {
                        var store = x.GetInstance<IDocumentStore>();
                        var session = store.OpenSession();
                        session.Advanced.UseOptimisticConcurrency = true;
                        return session;
                    });

                IndexCreation.CreateIndexes(typeof(BidHistory_NumberOfBids)
                                                .Assembly, documentStore);
            });
        }
    }
}
```

Lastly, there is a small configuration that is required in the `app.config` file of the `Presentation` console project, as shown in Listing 21-80.

LISTING 21-80: RavenDB Configuration within the app.config File

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
  <connectionStrings>
    <add name="RavenDB"
     connectionString=
               "Url=http://localhost:8080;Database=RepositoryExample" />
```

```
  </connectionStrings>
</configuration>
```

The `Program` class file within the `Presentation` console project is the same as the NHibernate version. Once you run the sample program, you can launch RavenDB's management studio and inspect the auction document, as shown in Figure 21-17.

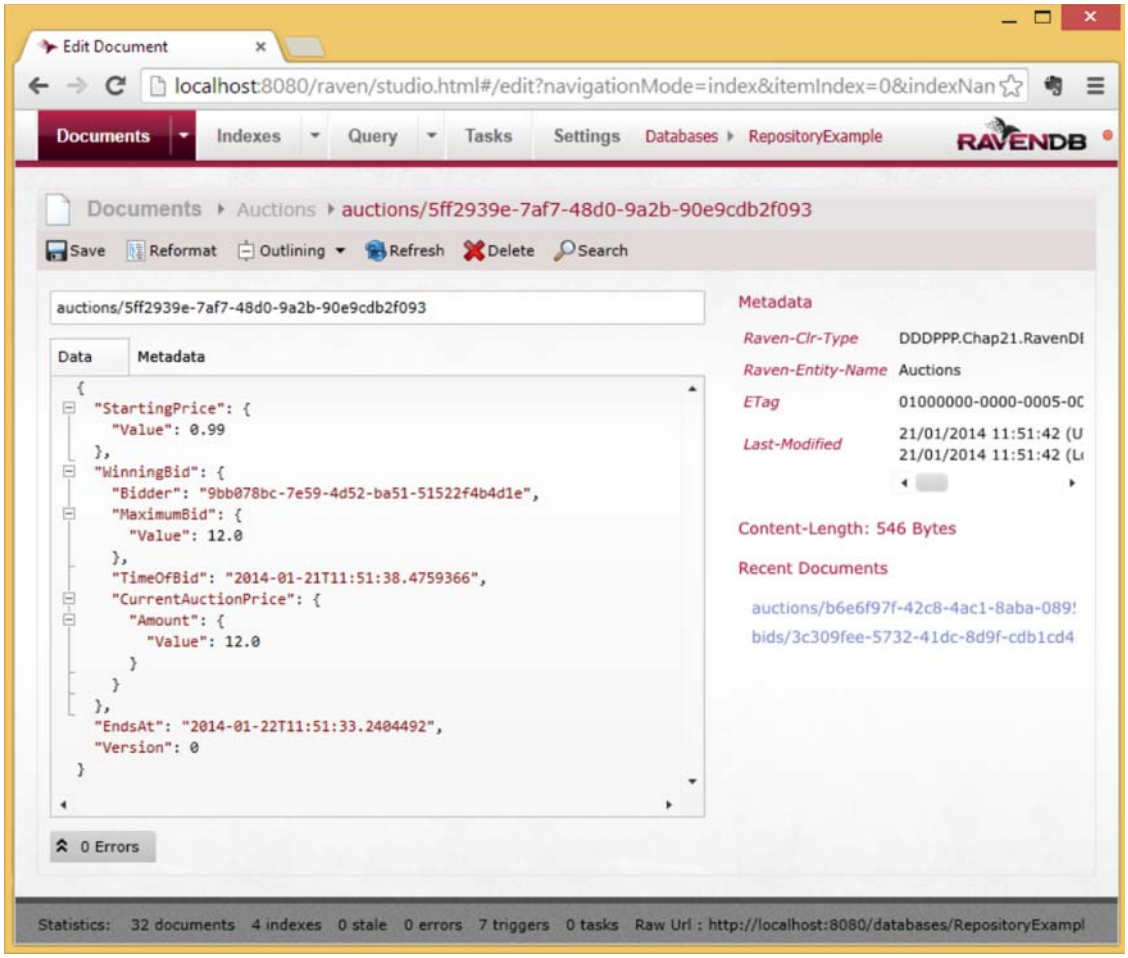


FIGURE 21-17: Inspecting the document inside RavenDB Management Studio.

Persistence Framework Cannot Map Domain Model Directly without Compromise

For the persistence frameworks that are unable to map a domain model directly to the data model, you need to take a different approach to persistence. It is still important to ensure that your data

model does not affect the structure of your domain model and that your domain model can evolve cleanly. You will run through two exercises: one using Entity Framework and the other raw ADO.NET with some help from Dapper, a micro ORM.

Entity Framework Example

The Entity Framework is Microsoft's enterprise-level ORM. It has similar capabilities to NHibernate, but it's fair to say that the NHibernate framework is a more mature project with more features for mapping a domain model directly. In this example, you use Entity Framework to map only the data model and utilize the memento pattern to provide a snapshot of the state of the domain model for storage.

Solution Setup

Create a new blank Visual Studio solution named `DDDPPP.Chap21.EFExample` and add to this a class library named `DDDPPP.Chap21.EFExample.Application` and a console application named `DDDPPP.Chap21.EFExample.Presentation`. Within the `Presentation` project, add a reference to the application. You can delete the `Class1.cs` that is automatically created because you won't be needing it. Use NuGet to install the Entity Framework client libraries, as shown in Figure 21.18.

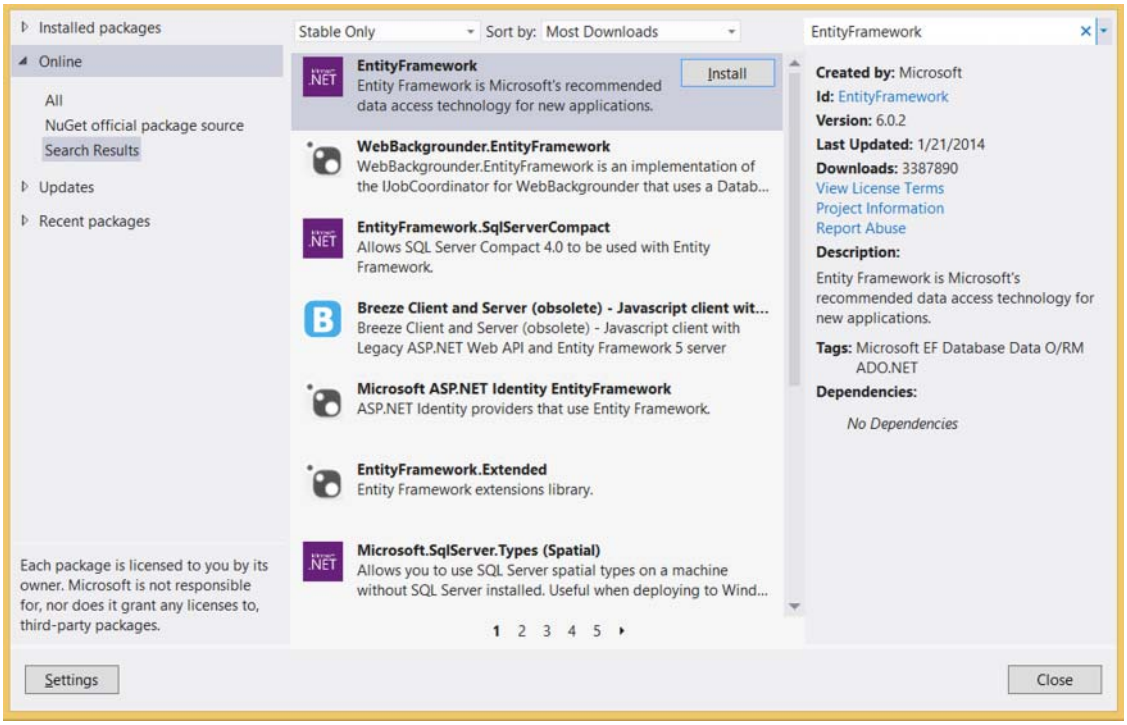


FIGURE 21-18: Install Entity Framework client libraries via NuGet.

Build out the application in the same manner as you did for the NHibernate example, or simply copy the class files into this new solution, ensuring that you update the namespaces. Remove the following classes that were specific to the NHibernate example:

- `BusinessUseCases\BidOnAuction.cs`
- `BusinessUseCases\CreateAuction.cs`
- `Queries\AuctionStatusQuery.cs`
- `Queries\BidHistoryQuery.cs`
- `Infrastructure\Auction.hbm.xml`
- `Infrastructure\Bid.hbm.xml`
- `Infrastructure\AuctionRepository.cs`
- `Infrastructure\BidHistoryRepository.cs`
- `Bootstrapper.cs`

You are left with a solution that resembles Figure 21-19.

Also, create the database tables using the same schema you used for the NHibernate example, if you haven't already created the database.

Altering the Model

Because you are going to implement the memento pattern, you need the aggregate to produce a snapshot of itself so you can map it to a data model. The first snapshot you will create is of the `WinningBid` value object. Using Listing 21-81, add a new class to the `Auction` folder within the `Model` folder named `WinningBidSnapshot`.

LISTING 21-81: The Bid Snapshot Class

```
using System;

namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public class WinningBidSnapshot
    {
        public Guid BiddersId { get; set; }
        public DateTime TimeOfBid { get; set; }
        public decimal BiddersMaximumBid { get; set; }
        public decimal CurrentPrice { get; set; }
    }
}
```

Next, create a snapshot for the auction itself, as shown in Listing 21-82. This holds the `WinningBidSnapshot` and provides a full snapshot of the `Auction` aggregate.

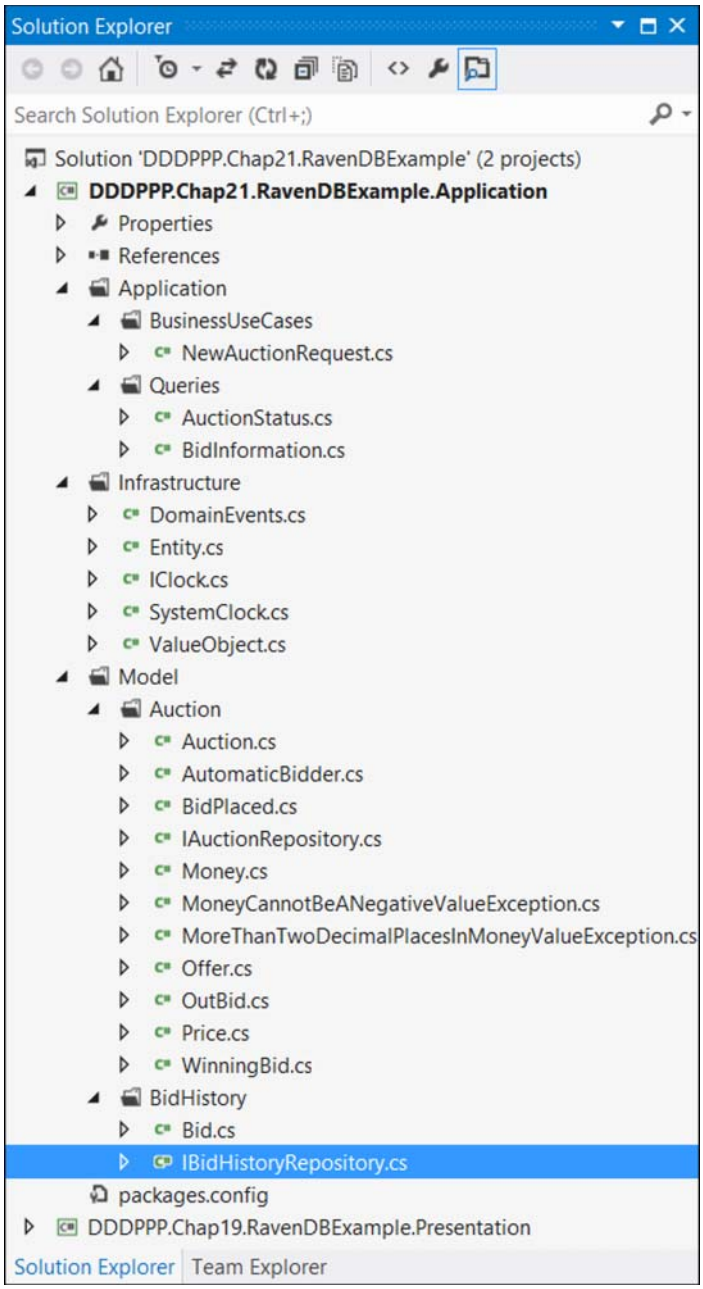


FIGURE 21-19: Solution explorer with an application based on the NHibernate sample.

LISTING 21-82: The Auction Snapshot Class

```
using System;

namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public class AuctionSnapshot
    {
        public Guid Id { get; set; }
        public decimal StartingPrice { get; set; }
        public DateTime EndsAt { get; set; }
        public WinningBidSnapshot WinningBid { get; set; }
        public int Version { get; set; }
    }
}
```

Finally, create a snapshot for the state of the money value object, as shown in Listing 21-83.

LISTING 21-83: The Money Snapshot Class

```
using System;

namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public class MoneySnapshot
    {
        public decimal Value { get; set; }
    }
}
```

Because you need access to setters and getters of the `Version` and `Id` properties, update the accessibility levels so that they have protected setters and public getters, as shown in Listing 21-84.

LISTING 21-84: The Entity Base Class

```
using System;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure
{
    public abstract class Entity<TId>
    {
        public TId Id { get; protected set; }
        public int Version { get; protected set; }
    }
}
```

To extract the state of the domain objects within the auction aggregate, you need to add a new public method that returns a snapshot. The first object to add this method to is the `Money` value object. Update the `Money` class to include the new `GetSnapshot` method, as shown in Listing 21-85.

LISTING 21-85: Adding a New Method to the Money Class

```
namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public class Money : ValueObject<Money>, IComparable<Money>
    {
        protected decimal Value { get; set; }

        public Money()
            : this(0m)
        {
        }

        // ....

        public MoneySnapshot GetSnapshot()
        {
            return new MoneySnapshot() { Value = this.Value };
        }
```

Do the same for the `WinningBid` object, but for this more complex type, add a new static factory method that enables a `WinningBid` to be created from a snapshot, as shown in Listing 21-86.

LISTING 21-86: Adding a New Method to the Winning Bid Class

```
namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public class WinningBid : ValueObject<WinningBid>
    {
        private WinningBid() { }

        public WinningBid(Guid bidder, Money maximumBid, Money bid,
                          DateTime timeOfBid)
        {
            if (bidder == Guid.Empty)
                throw new ArgumentNullException("Bidder cannot be null");

            if (maximumBid == null)
                throw new ArgumentNullException("MaximumBid cannot be null");

            if (timeOfBid == DateTime.MinValue)
                throw new ArgumentNullException("TimeOfBid must have a value");

            Bidder = bidder;
            MaximumBid = maximumBid;
            TimeOfBid = timeOfBid;
            CurrentAuctionPrice = new Price(bid);
        }

        // ....

        public WinningBidSnapshot GetSnapshot()
        {
```

```
            var snapshot = new WinningBidSnapshot();

            snapshot.BiddersId = this.Bidder;
            snapshot.BiddersMaximumBid = this.MaximumBid.GetSnapshot().Value;
            snapshot.CurrentPrice =
                  this.CurrentAuctionPrice.Amount.GetSnapshot().Value;
            snapshot.TimeOfBid = this.TimeOfBid;

            return snapshot;
        }

        public static WinningBid CreateFrom(WinningBidSnapshot bidSnapshot)
        {
            return new WinningBid(bidSnapshot.BiddersId,
                                  new Money(bidSnapshot.BiddersMaximumBid),
                                  new Money(bidSnapshot.CurrentPrice),
                                  bidSnapshot.TimeOfBid);
        }
```

Finally, update the `Auction` class by adding the static factory class to hydrate an auction from a snapshot and the method to obtain a snapshot, as shown in Listing 21-87.

LISTING 21-87: Updating the Auction Class

```
namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public class Auction : Entity<Guid>
    {
        public Auction(Guid id, Money startingPrice, DateTime endsAt)
        {

        if (id == Guid.Empty)
            throw new ArgumentNullException("Auction Id cannot be null");

        if (startingPrice == null)
            throw new ArgumentNullException("Starting Price cannot be null");

        if (endsAt == DateTime.MinValue)
            throw new ArgumentNullException("EndsAt must have a value");

            Id = id;
            StartingPrice = startingPrice;
            EndsAt = endsAt;
        }

        private Auction(AuctionSnapshot snapshot)
        {
            this.Id = snapshot.Id;
            this.StartingPrice = new Money(snapshot.StartingPrice);
            this.EndsAt = snapshot.EndsAt;
            this.Version = snapshot.Version;

            if (snapshot.WinningBid != null)
```

continues

LISTING 21-87 *(continued)*

```
            WinningBid = WinningBid.CreateFrom(snapshot.WinningBid);
        }

        public static Auction CreateFrom(AuctionSnapshot snapshot)
        {
            return new Auction(snapshot);
        }

        private Money StartingPrice { get; set; }
        private WinningBid CurrentWinningBid { get; set; }
        private DateTime EndsAt { get; set; }

        public AuctionSnapshot GetSnapshot()
        {
            var snapshot = new AuctionSnapshot();
            snapshot.Id = this.Id;
            snapshot.StartingPrice = this.StartingPrice.GetSnapshot().Value;
            snapshot.EndsAt = this.EndsAt;
            snapshot.Version = this.Version;

            if (HasACurrentBid())
                snapshot.WinningBid = WinningBid.GetSnapshot();

            return snapshot;
        }

        private bool HasACurrentBid()
        {
            return WinningBid != null;
        }

        // ....
```

You also need to add the `BidHistory` object, as you did in the RavenDB example. Listing 21-88 shows an example.

LISTING 21-88: The Bid History Class

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace DDDPPP.Chap21.EFExample.Application.Model.BidHistory
{
    public class BidHistory
    {
        private IEnumerable<Bid> _bids;

        public BidHistory(IEnumerable<Bid> bids)
        {
```

```
            if (bids == null)
              throw new ArgumentNullException("Bids cannot be null");

            _bids = bids;
        }

        public IEnumerable<Bid> ShowAllBids()
        {
            var bids = _bids.OrderByDescending(x => x.AmountBid)
                                                .ThenBy(x => x.TimeOfBid);

            return bids;
        }
    }
}
```

To retrieve the `BidHistory` object, you need to amend the `IBidHistoryRepository`, as shown in Listing 21-89.

LISTING 21-89: The Bid History Repository Interface

```
using System;

namespace DDDPPP.Chap21.EFExample.Application.Model.BidHistory
{
    public interface IBidHistoryRepository
    {
        int NoOfBidsFor(Guid autionId);
        void Add(Bid bid);
        BidHistory FindBy(Guid auctionId);
    }
}
```

Because you can't track changes implicitly with your domain objects, you need to explicitly save them, which means you must add a save method to the `IAuctionRepository`, as shown in Listing 21-90.

LISTING 21-90: The Auction Repository Interface

```
using System;

namespace DDDPPP.Chap21.EFExample.Application.Model.Auction
{
    public interface IAuctionRepository
    {
        void Add(Auction auction);
        void Save(Auction auction);
        Auction FindBy(Guid Id);
    }
}
```

Entity Framework Repository Implementation

Create a new folder within the `Infrastructure` folder named `DataModel`. Here you stall the data model objects that map to your database tables and rows. The first object to create represents a row within the `Auction` table, as shown in Listing 21-91.

LISTING 21-91: The Auction Data Transfer Object

```
using System;
using System.Collections.Generic;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel
{
    public partial class AuctionDTO
    {
        public System.Guid Id { get; set; }
        public decimal StartingPrice { get; set; }
        public DateTime AuctionEnds { get; set; }

        public Nullable<System.Guid> BidderMemberId { get; set; }
        public System.DateTime? TimeOfBid { get; set; }
        public Nullable<decimal> MaximumBid { get; set; }
        public Nullable<decimal> CurrentPrice { get; set; }

        public int Version { get; set; }
    }
}
```

The second data object, Listing 21-92, represents a row within the `BidHistory` table.

LISTING 21-92: The Bid Data Transfers Object

```
using System;
using System.Collections.Generic;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel
{
    public partial class BidDTO
    {
        public System.Guid Id { get; set; }
        public System.Guid AuctionId { get; set; }
        public System.Guid BidderId { get; set; }
        public decimal Bid { get; set; }
        public System.DateTime TimeOfBid { get; set; }
    }
}
```

You will use Entity Framework's code-first mapping to map the data model to the database. Add a new folder to the `Infrastructure` folder named `Mapping`. The `AuctionMap` class, Listing 21-93, maps the `AuctionDTO` to the `Auctions` database table.

LISTING 21-93: The Auction Map Class

```
using System.ComponentModel.DataAnnotations.Schema;
using System.Data.Entity.ModelConfiguration;
using DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure.Mapping
{
    public class AuctionMap : EntityTypeConfiguration<AuctionDTO>
    {
        public AuctionMap()
        {
            this.HasKey(t => t.Id);

            this.ToTable("Auctions");
            this.Property(t => t.Id).HasColumnName("Id");
            this.Property(t => t.StartingPrice).HasColumnName("StartingPrice");
            this.Property(t =>
                            t.BidderMemberId).HasColumnName("BidderMemberId");
            this.Property(t => t.TimeOfBid).HasColumnName("TimeOfBid");
            this.Property(t => t.MaximumBid).HasColumnName("MaximumBid");
            this.Property(t => t.CurrentPrice).HasColumnName("CurrentPrice");
            this.Property(t => t.AuctionEnds).HasColumnName("AuctionEnds");
            this.Property(t =>
                  t.Version).HasColumnName("Version").IsConcurrencyToken();
        }
    }
}
```

Similarly, the `BidMap`, Listing 21-94, maps the `BidDTO` to the `BidHistory` table.

LISTING 21-94: The Bid Map Class

```
using System.ComponentModel.DataAnnotations.Schema;
using System.Data.Entity.ModelConfiguration;
using DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure.Mapping
{
    public class BidMap : EntityTypeConfiguration<BidDTO>
    {
        public BidMap()
        {
            this.HasKey(t => t.Id);

            this.ToTable("BidHistory");
            this.Property(t => t.Id).HasColumnName("Id");
            this.Property(t => t.AuctionId).HasColumnName("AuctionId");
            this.Property(t => t.BidderId).HasColumnName("BidderId");
            this.Property(t => t.Bid).HasColumnName("Bid");
            this.Property(t => t.TimeOfBid).HasColumnName("TimeOfBid");
```

continues

LISTING 21-94 *(continued)*

```
        }
    }
}
```

To talk to the database, you need a context, as shown in Listing 21-95. The Entity Framework `DbContext` is similar to NHibernate's `ISession` and RavenDB's `IDocumentSession`. It is effectively Entity Framework's implementation of the unit-of-work pattern.

LISTING 21-95: Entity Framework's Auction Database Context Class

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel;
using DDDPPP.Chap21.EFExample.Application.Infrastructure.Mapping;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure
{
    public partial class AuctionDatabaseContext : DbContext
    {
        static AuctionDatabaseContext()
        {
            Database.SetInitializer<AuctionDatabaseContext>(null);
        }

        public AuctionDatabaseContext()
            : base("Name=AuctionDatabaseContext")
        {
        }

        public DbSet<AuctionDTO> Auctions { get; set; }
        public DbSet<BidDTO> Bids { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Configurations.Add(new AuctionMap());
            modelBuilder.Configurations.Add(new BidMap());
        }

        public void Clear()
        {
            var context = ((IObjectContextAdapter)this).ObjectContext;

            var addedObjects = context
                               .ObjectStateManager
                               .GetObjectStateEntries(EntityState.Added);

            foreach (var objectStateEntry in addedObjects)
            {
                context.Detach(objectStateEntry.Entity);
            }

            var modifiedObjects = context
```

```
                                .ObjectStateManager
                                .GetObjectStateEntries(EntityState.Modified);

            foreach (var objectStateEntry in modifiedObjects)
            {
                context.Detach(objectStateEntry.Entity);
            }
        }
    }
}
```

The implementation of the repository is different from what you have seen in the NHibernate and RavenDB examples. Because you are unable to map the domain model directly to the data model, the repository needs to extract the snapshot and map that instead. The `AuctionRepository` implementation is shown in Listing 21-96.

LISTING 21-96: Entity Framework's Auction Repository Implementation

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.EFExample.Application.Model.Auction;
using DDDPPP.Chap21.EFExample.Application.Application;
using DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure
{
    public class AuctionRepository : IAuctionRepository
    {
        private readonly AuctionDatabaseContext _auctionExampleContext;

        public AuctionRepository(AuctionDatabaseContext auctionExampleContext)
        {
            _auctionExampleContext = auctionExampleContext;
        }

        public void Add(Auction auction)
        {
            var auctionDTO = new AuctionDTO();

            Map(auctionDTO, auction.GetSnapshot());

           _auctionExampleContext.Auctions.Add(auctionDTO);
        }

        public void Save(Auction auction)
        {
            var auctionDTO = _auctionExampleContext.Auctions.Find(auction.Id);

            Map(auctionDTO, auction.GetSnapshot());
        }

        public Auction FindBy(Guid Id)
        {
            var auctionDTO = _auctionExampleContext.Auctions.Find(Id);
```

continues

LISTING 21-96 *(continued)*

```
            var auctionSnapshot = new AuctionSnapshot();

            auctionSnapshot.Id = auctionDTO.Id;
            auctionSnapshot.EndsAt = auctionDTO.AuctionEnds;
            auctionSnapshot.StartingPrice = auctionDTO.StartingPrice;
            auctionSnapshot.Version = auctionDTO.Version;

            if (auctionDTO.BidderMemberId.HasValue)
            {
                var bidSnapshot = new WinningBidSnapshot();

                bidSnapshot.BiddersMaximumBid = auctionDTO.MaximumBid.Value;
                bidSnapshot.CurrentPrice = auctionDTO.CurrentPrice.Value;
                bidSnapshot.BiddersId = auctionDTO.BidderMemberId.Value;
                bidSnapshot.TimeOfBid = auctionDTO.TimeOfBid.Value;
                auctionSnapshot.WinningBid = bidSnapshot;
            }

            return Auction.CreateFrom(auctionSnapshot);
        }

        public void Map(AuctionDTO auctionDTO, AuctionSnapshot snapshot)
        {
            auctionDTO.Id = snapshot.Id;
            auctionDTO.StartingPrice = snapshot.StartingPrice;
            auctionDTO.AuctionEnds = snapshot.EndsAt;
            auctionDTO.Version = snapshot.Version;

            if (snapshot.WinningBid != null)
            {
                auctionDTO.BidderMemberId = snapshot.WinningBid.BiddersId;
                auctionDTO.CurrentPrice = snapshot.WinningBid.CurrentPrice;
                auctionDTO.MaximumBid = snapshot.WinningBid.BiddersMaximumBid;
                auctionDTO.TimeOfBid = snapshot.WinningBid.TimeOfBid;
            }
        }
    }
}
```

The `BidHistoryRepository` works along the same lines as what's shown in Listing 21-97.

LISTING 21-97: The Bid History Repository

```
using System;
using System.Collections.Generic;
using System.Linq;
using DDDPPP.Chap21.EFExample.Application.Model.BidHistory;
using DDDPPP.Chap21.EFExample.Application.Model.Auction;
using DDDPPP.Chap21.EFExample.Application.Infrastructure.DataModel;

namespace DDDPPP.Chap21.EFExample.Application.Infrastructure
```

```
{
    public class BidHistoryRepository : IBidHistoryRepository
    {
        private readonly AuctionDatabaseContext _auctionExampleContext;

        public BidHistoryRepository(
                    AuctionDatabaseContext auctionExampleContext)
        {
            _auctionExampleContext = auctionExampleContext;
        }

        public int NoOfBidsFor(Guid autionId)
        {
            return _auctionExampleContext.Bids
                            .Count(x => x.AuctionId == autionId);
        }

        public void Add(Bid bid)
        {
            var bidDTO = new BidDTO();

            bidDTO.AuctionId = bid.AuctionId;
            bidDTO.Bid = bid.AmountBid.GetSnapshot().Value;
            bidDTO.BidderId = bid.Bidder;
            bidDTO.TimeOfBid = bid.TimeOfBid;

            bidDTO.Id = Guid.NewGuid();

            _auctionExampleContext.Bids.Add(bidDTO);
        }

        public BidHistory FindBy(Guid auctionId)
        {
            var bidDTOs = _auctionExampleContext.Bids.Where<BidDTO>(x =>
                                        x.AuctionId == auctionId).ToList();
            var bids = new List<Bid>();

            foreach (var bidDTO in bidDTOs)
            {
                bids.Add(new Bid(bidDTO.AuctionId, bidDTO.BidderId,
                                 new Money(bidDTO.Bid), bidDTO.TimeOfBid));
            }

            return new BidHistory(bids);
        }
    }
}
```

Application Service

There isn't much difference in either of the application services from what you have seen up to now except for using the `DbContext`, which is Entity Framework's unit of work implementation. The only thing of note is to say that Entity Framework implicitly wraps the call to `SaveChanges` in a transaction so you don't need to. You can see these changes in Listing 21-98.

LISTING 21-98: The Create Auction Application Service

```
using System;
using System.Collections.Generic;
using System.Linq;
using DDDPPP.Chap21.EFExample.Application.Model.Auction;
using DDDPPP.Chap21.EFExample.Application.Model.BidHistory;
using DDDPPP.Chap21.EFExample.Application.Infrastructure;

namespace DDDPPP.Chap21.EFExample.Application.Application.BusinessUseCases
{
    public class CreateAuction
    {
        private IAuctionRepository _auctions;
        private AuctionDatabaseContext _unitOfWork;

        public CreateAuction(IAuctionRepository auctions,
                             AuctionDatabaseContext unitOfWork)
        {
            _auctions = auctions;
            _unitOfWork = unitOfWork;
        }

        public Guid Create(NewAuctionRequest command)
        {
            var auctionId = Guid.NewGuid();
            var startingPrice = new Money(command.StartingPrice);

            _auctions.Add(new Auction(auctionId, startingPrice,
                                      command.EndsAt));

            _unitOfWork.SaveChanges();

            return auctionId;
        }
    }
}
```

In the `BidOnAuction` application service, Listing 21-99, you need to explicitly call the `Save` method on the repository. The only other difference is that the exception type that is thrown when there is a concurrency issue is different from both the NHibernate and RavenDB examples.

LISTING 21-99: The Bid On Auction Application Service

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.EFExample.Application.Model.Auction;
using DDDPPP.Chap21.EFExample.Application.Model.BidHistory;
using DDDPPP.Chap21.EFExample.Application.Infrastructure;
using System.Data.Entity.Infrastructure;

namespace DDDPPP.Chap21.EFExample.Application.Application.BusinessUseCases
```

```
{
    public class BidOnAuction
    {
        private IAuctionRepository _auctions;
        private IBidHistoryRepository _bidHistory;
        private AuctionDatabaseContext _unitOfWork;
        private IClock _clock;

        public BidOnAuction(IAuctionRepository auctions,
                            IBidHistoryRepository bidHistory,
                            AuctionDatabaseContext unitOfWork, IClock clock)
        {
            _auctions = auctions;
            _bidHistory = bidHistory;
            _unitOfWork = unitOfWork;
            _clock = clock;
        }

        public void Bid(Guid auctionId, Guid memberId, decimal amount)
        {
            try
            {
                using (DomainEvents.Register(OutBid()))
                using (DomainEvents.Register(BidPlaced()))
                {
                    var auction = _auctions.FindBy(auctionId);

                    var bidAmount = new Money(amount);

                    auction.PlaceBidFor(new Offer(memberId, bidAmount,
                                                  _clock.Time()), _clock.Time());

                    _auctions.Save(auction);
                }

                _unitOfWork.SaveChanges();
            }
            catch (DbUpdateConcurrencyException ex)
            {
                _unitOfWork.Clear();

                Bid(auctionId, memberId, amount);
            }
        }

        private Action<BidPlaced> BidPlaced()
        {
            return (BidPlaced e) =>
            {
                var bidEvent = new Bid(e.AuctionId, e.Bidder,
                                       e.AmountBid, e.TimeOfBid);

                _bidHistory.Add(bidEvent);
```

continues

LISTING 21-99 *(continued)*

```
            };
        }

        private Action<OutBid> OutBid()
        {
            return (OutBid e) =>
            {
                // E-mail customer to say that he has been outbid
            };
        }
    }
}
```

Querying

To implement the queries, you can utilize the snapshots to populate the view models, as shown in Listing 21-100 and Listing 21-101 for the `AuctionStatusQuery` and the `BidHistoryQuery`.

LISTING 21-100: The Auction Status Query

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.EFExample.Application.Model.Auction;
using DDDPPP.Chap21.EFExample.Application.Model.BidHistory;
using DDDPPP.Chap21.EFExample.Application.Infrastructure;

namespace DDDPPP.Chap21.EFExample.Application.Application.Queries
{
    public class AuctionStatusQuery
    {
        private readonly IAuctionRepository _auctions;
        private readonly IBidHistoryRepository _bidHistory;
        private readonly IClock _clock;

        public AuctionStatusQuery(IAuctionRepository auctions,
                                  IBidHistoryRepository bidHistory,
                                  IClock clock)
        {
            _auctions = auctions;
            _bidHistory = bidHistory;
            _clock = clock;
        }

        public AuctionStatus AuctionStatus(Guid auctionId)
        {
            var auction = _auctions.FindBy(auctionId);

            var snapshot = auction.GetSnapshot();

            return ConvertToStatus(snapshot);
```

```
        }

        public AuctionStatus ConvertToStatus(AuctionSnapshot snapshot)
        {
            var status = new AuctionStatus();

            status.AuctionEnds = snapshot.EndsAt;
            status.Id = snapshot.Id;
            status.TimeRemaining = TimeRemaining(snapshot.EndsAt);

            if (snapShot.WinningBid != null)
            {
                status.NumberOfBids = _bidHistory.NoOfBidsFor(snapshot.Id);
                status.WinningBidderId = snapshot.WinningBid.BiddersId;
                status.CurrentPrice = snapshot.WinningBid.CurrentPrice;
            }

            return status;
        }

        public TimeSpan TimeRemaining(DateTime AuctionEnds)
        {
            if (_clock.Time() < AuctionEnds)
                return AuctionEnds.Subtract(_clock.Time());
            else
                return new TimeSpan();
        }
    }
}
```

LISTING 21-101: The Bid Histroy Query

```
using System;
using System.Collections.Generic;
using DDDPPP.Chap21.EFExample.Application.Infrastructure;
using DDDPPP.Chap21.EFExample.Application.Model.BidHistory;

namespace DDDPPP.Chap21.EFExample.Application.Application.Queries
{
    public class BidHistoryQuery
    {
        private readonly IBidHistory _bidHistory;

        public BidHistoryQuery(IBidHistory bidHistory)
        {
            _bidHistory = bidHistory;
        }

        public IEnumerable<BidInformation> BidHistoryFor(Guid auctionId)
        {
            var bidHistory = _bidHistory.FindBy(auctionId);

            return Convert(bidHistory.ShowAllBids());
```

continues

LISTING 21-101 *(continued)*

```
        }

        public IEnumerable<BidInformation> Convert(IEnumerable<BidEvent> bids)
        {
            var bidInfo = new List<BidInformation>();

            foreach (var bid in bids)
            {
                bidInfo.Add(new BidInformation() { Bidder = bid.Bidder,
                                AmountBid = bid.AmountBid.GetSnapshot().Value,
                                TimeOfBid = bid.TimeOfBid });
            }

            return bidInfo;
        }
    }
}
```

Configuration

Again, you will be using StructureMap to wire up your dependencies, so install it as before from NuGet and create a bootstrapper class with Listing 21-102.

LISTING 21-102: The StructureMap Bootstrapper Class

```
using System;
using StructureMap;
using DDDPPP.Chap21.EFExample.Application.Infrastructure;
using DDDPPP.Chap21.EFExample.Application.Model.Auction;
using DDDPPP.Chap21.EFExample.Application.Model.BidHistory;

namespace DDDPPP.Chap21.EFExample.Application
{
    public static class Bootstrapper
    {
        public static void Startup()
        {
            ObjectFactory.Initialize(config =>
            {
                config.For<IAuctionRepository>().Use<AuctionRepository>();
                config.For<IBidHistoryRepository>()
                                                .Use<BidHistoryRepository>();
                config.For<IClock>().Use<SystemClock>();
            });
        }
    }
}
```

Lastly, add the XML snippet in Listing 21-103 to the `app.config` within the `Presentation` console application.

LISTING 21-103: The Entity Framework Configuration

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit
                          http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.
EntityFrameworkSection,
                        EntityFramework, Version=6.0.0.0, Culture=neutral,
                 PublicKeyToken=b77a5c561934e089" requirePermission="false" />
  <!-- For more information on Entity Framework configuration, visit
            http://go.microsoft.com/fwlink/?LinkID=237468 --></configSections>
  <connectionStrings>
    <add name="AuctionDatabaseContext" connectionString="Data
       Source=.\sqlexpress;Initial Catalog=AuctionExample;Integrated
                        Security=True;MultipleActiveResultSets=True"
                                 providerName="System.Data.SqlClient" />
  </connectionStrings>
  <entityFramework>
    <defaultConnectionFactory
                type="System.Data.Entity.Infrastructure.SqlConnectionFactory,
                                    EntityFramework" />
    <providers>
      <provider invariantName="System.Data.SqlClient"
               type="System.Data.Entity.SqlServer.SqlProviderServices,
                                                 EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
</configuration>
```

If you run the console application, you will see the same results as in the previous NHibernate and RavenDB examples.

Micro ORM Example

Dapper is a micro ORM, meaning that it helps you map database rows to objects but (by design) does little else. In this example, you need to build your own unit of work implementation and concurrency checks.

Solution Setup

To get started, create a new blank Visual Studio solution named `DDDPPP.Chap21.MicroORM` and add to this a class library named `DDDPPP.Chap21.MicroORM.Application` and a console application named `DDDPPP.Chap21.MicroORM.Presentation`. Within the `Presentation` project, add a reference to the application. You can delete the `Class1.cs` that is automatically created because you won't be needing it. Use NuGet to install the Dapper client libraries, as shown in Figure 21.20.

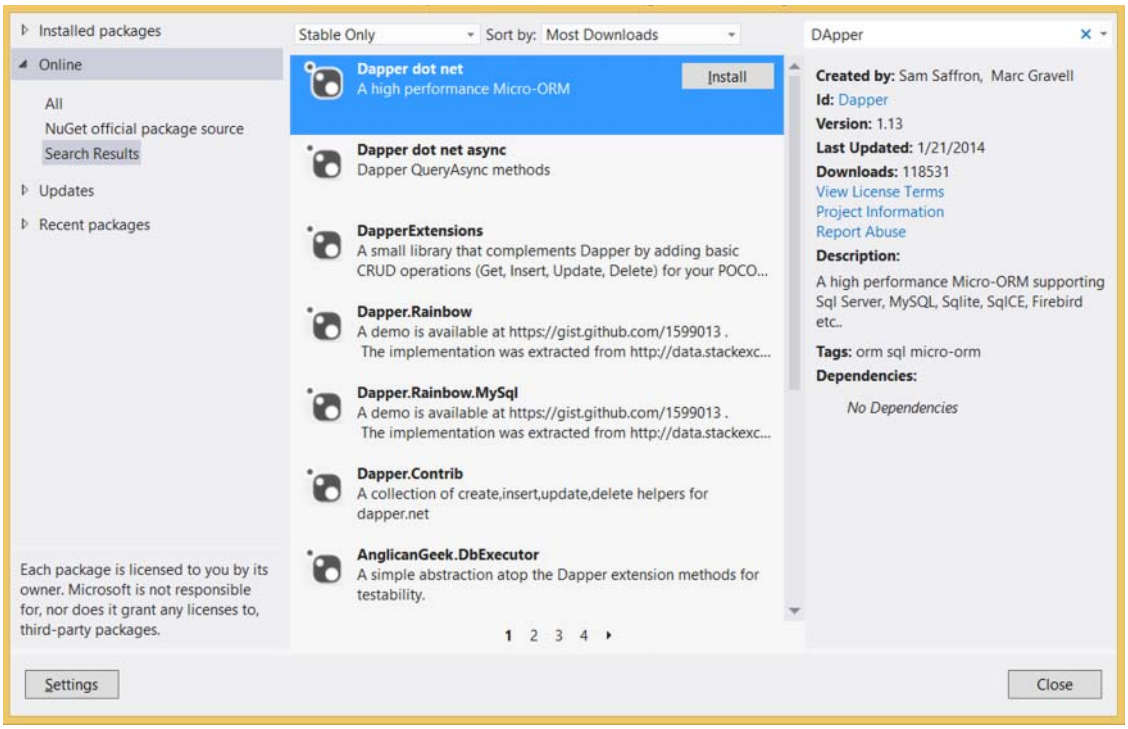


FIGURE 21-20: Install Dapper client libraries via NuGet.

The model is the same as the model you created for the Entity Framework example. It also has the database schema that you used in both the NHibernate and Entity Framework examples. Build out the application in the same manner as you did for the Entity Framework, but exclude the following files:

- `BusinessUseCases/BidOnAuction.cs`
- `BusinessUseCases/CreateAuction.cs`
- `Infrastructure/Mapping`
- `Infrastructure/AuctionDatabaseContext.cs`
- `Infrastructure/AuctionRepository.cs`
- `Infrastructure/BidHistoryRepository.cs`
- `Bootstrapper.cs`

You should now be left with a solution that resembles Figure 21-21.

Infrastructure

In this example we're using a micro ORM. Micro ORMs tend to focus on simplicity and performance, as such they have less features than their fully automated cousins Entity Framework and NHibernate. Consequently we will need to implement the unit of work pattern ourselves. The unit of work structure in this example is based on the framework that Tim McCarthy uses in his book *.NET Domain-Driven Design with C#: Problem-Design-Solution*.

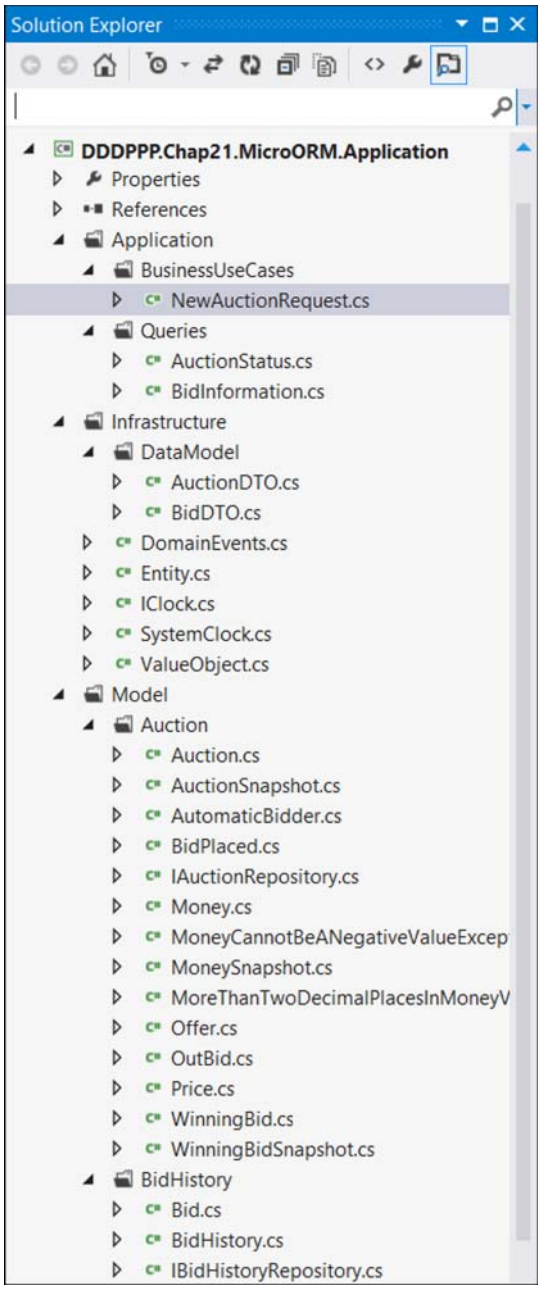


FIGURE 21-21: Skeleton solution based on the Entity Framework solution.

The `IAggregateDataModel` interface, Listing 21-104, is actually a pattern in itself called the marker interface pattern. The interface acts as metadata for a class, and methods that interact with instances of that class test for the existence of the interface before carrying out their work. You will see this pattern used later in this chapter when you build a repository layer that only persists business objects that implement the `IAggregateDataModel` interface.

LISTING 21-104: The IAggregateDataModel Interface

```
using System;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
    public interface IAggregateDataModel
    {
    }
}
```

The unit of work implementation uses the `IAggregateDataModel` interface to reference any business entity that is partaking in an atomic transaction. Add another interface to the `Infrastructure` project named `IUnitOfWorkRepository`, Listing 21-105, with the contract listing that follows.

LISTING 21-105: The IUnitofWorkRepository Interface

```
using System;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
    public interface IUnitOfWorkRepository
    {
        void PersistCreationOf(IAggregateDataModel entity);
        void PersistUpdateOf(IAggregateDataModel entity);
    }
}
```

The `IUnitOfWorkRepository` is a second interface that all repositories are required to implement if they intend to be used in a unit of work. You could have added this contract definition to the model `Repository` interface that you will add later, but the interfaces are addressing two different types of concerns. This is the definition of the Interface Segregation Principle. You are not going to be deleting anything in your application, so there is no need to add that method.

Finally, add a third interface to the `Infrastructure` project named `IUnitOfWork`, the definition of which you can find in Listing 21-106.

LISTING 21-106: The IUnitOfWork Interface

```
using System;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
    public interface IUnitOfWork
    {
        void RegisterAmended(IAggregateDataModel entity,
                             IUnitOfWorkRepository unitofWorkRepository);
        void RegisterNew(IAggregateDataModel entity,
                             IUnitOfWorkRepository unitofWorkRepository);
        void Commit();
```

```
        void Clear();
    }
}
```

The `IUnitOfWork` interface requires the `IUnitOfWorkRepository` when registering an amend/addition/deletion so that, on commitment, the unit of work can delegate the work of the actual persistence method to the appropriate concrete implementation. The logic behind the `IUnitOfWork` methods will become a lot clearer when you look at a default implementation of the `IUnitOfWork` interface and create the repositories.

The last class to create is the `ConcurrencyException`, Listing 21-107, which is thrown if the auction is updated between retrieving it, placing a bid, and persisting it.

LISTING 21-107: The Concurrency Exception Class

```
using System;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
    public class ConcurrencyException : ApplicationException
    {
    }
}
```

Application Service

Both of the application services should be familiar to you by now. Listing 21-108 and Listing 21-109 change because of your implementation of the unit of work and because of your concurrency exception type.

LISTING 21-108: The Create Auction Application Service

```
using System;
using DDDPPP.Chap21.MicroORM.Application.Model.Auction;
using DDDPPP.Chap21.MicroORM.Application.Infrastructure;

namespace DDDPPP.Chap21.MicroORM.Application.Application.BusinessUseCases
{
    public class CreateAuction
    {
        private IAuctionRepository _auctionRepository;
        private IUnitOfWork _unitOfWork;

        public CreateAuction(IAuctionRepository auctionRepository,
                             IUnitOfWork unitOfWork)
        {
            _auctionRepository = auctionRepository;
            _unitOfWork = unitOfWork;
        }

        public Guid Create(NewAuctionRequest command)
        {
```

continues

LISTING 21-108 *(continued)*

```
            var auctionId = Guid.NewGuid();
            var startingPrice = new Money(command.StartingPrice);

            _auctionRepository.Add(new Auction(auctionId,
                                    startingPrice, command.EndsAt));

            _unitOfWork.Commit();

            return auctionId;
        }
    }
}
```

LISTING 21-109: The Bid On Auction Application Service

```
using System;
using DDDPPP.Chap21.MicroORM.Application.Model.Auction;
using DDDPPP.Chap21.MicroORM.Application.Model.BidHistory;
using DDDPPP.Chap21.MicroORM.Application.Infrastructure;

namespace DDDPPP.Chap21.MicroORM.Application.Application.BusinessUseCases
{
    public class BidOnAuction
    {
        private IAuctionRepository _auctionRepository;
        private IBidHistoryRepository _bidHistoryRepository;
        private IUnitOfWork _unitOfWork;
        private IClock _clock;

        public BidOnAuction(IAuctionRepository auctionRepository,
                            IBidHistoryRepository bidHistoryRepository,
                            IUnitOfWork unitOfWork, IClock clock)
        {
            _auctionRepository = auctionRepository;
            _bidHistoryRepository = bidHistoryRepository;
            _unitOfWork = unitOfWork;
            _clock = clock;
        }

        public void Bid(Guid auctionId, Guid memberId, decimal amount)
        {
            try
            {
                using (DomainEvents.Register(OutBid()))
                using (DomainEvents.Register(BidPlaced()))
                {
                    var auction = _auctionRepository.FindBy(auctionId);

                    var bidAmount = new Money(amount);

                    auction.PlaceBidFor(new Offer(memberId, bidAmount,
```

```
                                        _clock.Time()), _clock.Time());

                    _auctionRepository.Save(auction);
                }

                _unitOfWork.Commit();
            }
            catch (ConcurrencyException ex)
            {
                _unitOfWork.Clear();

                Bid(auctionId, memberId, amount);
            }
        }

        private Action<BidPlaced> BidPlaced()
        {
            return (BidPlaced e) =>
            {
                var bidEvent = new Bid(e.AuctionId, e.Bidder, e.AmountBid,
                                       e.TimeOfBid);

                _bidHistoryRepository.Add(bidEvent);
            };
        }

        private Action<OutBid> OutBid()
        {
            return (OutBid e) =>
            {
                // E-mail customer to say that he has been outbid
            };
        }
    }
}
```

ADO.NET Repository Implementation

You need to make the `AuctionDTO`, Listing 21-110, and `BidDTO`, Listing 21-111, implement the `IAggregateDataModel` interface because these two classes are persisted using the unit of work implementation.

LISTING 21-110: Updating the AuctionDTO to Implementing the IAggregateDataModel

```
namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure.DataModel
{
    public partial class AuctionDTO : IAggregateDataModel
    {
        // ......
    }
}
```

LISTING 21-111: Updating the BidDTO to Implementing the IAggregateDataModel

```
namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure.DataModel
{
    public partial class BidDTO : IAggregateDataModel
    {
        // ......
    }
}
```

Both `AuctionRepository` and `BidHistoryRepository`, shown in Listing 21-112 and Listing 21-113, implement the domain model repository contracts `IAuctionRepository` and `IBidHistoryRepository`, as well as the `IUnitOfWorkRepository` interface. The implementations of both repository methods simply delegate work to the unit of work, passing the entity to be persisted along with a reference to the repository, which of course implements the `IUnitOfWorkRepository`. As seen previously when the unit of work's `Commit` method is called, the unit of work refers to the repository's implementation of the `IUnitOfWorkRepository` contract to perform the real persistence requirements. This is the same behavior as you saw in the enterprise-level frameworks. Because the repositories need to obtain a connection string from the `app.config` file, you need to add a reference to the `Systems.Configuration` assembly, as shown in Figure 21-22.

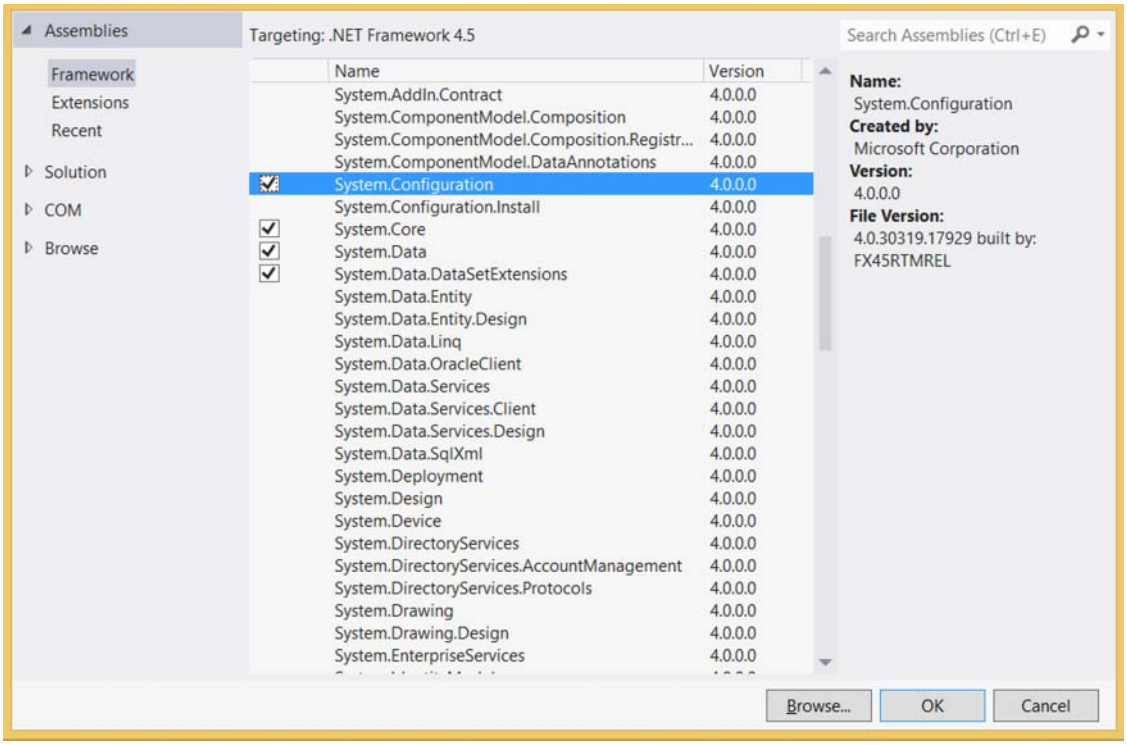


FIGURE 21-22: Add a reference to System.Configuration.

LISTING 21-112: The Auction Repository Implementation

```
using System;
using System.Collections.Generic;
using System.Linq;
using DDDPPP.Chap21.MicroORM.Application.Model.Auction;
using DDDPPP.Chap21.MicroORM.Application.Application;
using DDDPPP.Chap21.MicroORM.Application.Infrastructure.DataModel;
using System.Data.SqlClient;
using Dapper;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
    public class AuctionRepository : IAuctionRepository, IUnitOfWorkRepository
    {
        private IUnitOfWork _unitOfWork;

        public AuctionRepository(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }

        public void Add(Auction auction)
        {
            var snapshot = auction.GetSnapshot();
            var auctionDTO = new AuctionDTO();

            Map(auctionDTO, snapshot);

            _unitOfWork.RegisterNew(auctionDTO, this);
        }

        public void Save(Auction auction)
        {
            var snapshot = auction.GetSnapshot();
            var auctionDTO = new AuctionDTO();


            Map(auctionDTO, snapshot);

            _unitOfWork.RegisterAmended(auctionDTO, this);
        }

        public Auction FindBy(Guid Id)
        {
            AuctionDTO auctionDTO;

            using (var connection = new
                    SqlConnection(System.Configuration.ConfigurationManager
                          .ConnectionStrings["AuctionDB"].ConnectionString))
            {
                auctionDTO = connection.Query<AuctionDTO>("Select * From " +
                             "Auctions Where Id = CAST(@Id AS uniqueidentifier)"
                                , new { Id = Id }).FirstOrDefault();
```

continues

LISTING 21-112 *(continued)*

```
        }

        var auctionSnapshot = new AuctionSnapshot();

        auctionSnapshot.Id = auctionDTO.Id;
        auctionSnapshot.EndsAt = auctionDTO.AuctionEnds;
        auctionSnapshot.StartingPrice = auctionDTO.StartingPrice;
        auctionSnapshot.Version = auctionDTO.Version;

        if (auctionDTO.BidderMemberId.HasValue)
        {
            var bidSnapshot = new WinningBidSnapshot();

            bidSnapshot.BiddersMaximumBid = auctionDTO.MaximumBid.Value;
            bidSnapshot.CurrentPrice = auctionDTO.CurrentPrice.Value;
            bidSnapshot.BiddersId = auctionDTO.BidderMemberId.Value;
            bidSnapshot.TimeOfBid = auctionDTO.TimeOfBid.Value;
            auctionSnapshot.WinningBid = bidSnapshot;
        }

        return Auction.CreateFrom(auctionSnapshot);
    }

    public void PersistCreationOf(IAggregateDataModel entity)
    {
        var auctionDTO = (AuctionDTO)entity;

        using (var connection = new
             SqlConnection(System.Configuration.ConfigurationManager
           .ConnectionStrings["AuctionDB"].ConnectionString))
        {
            var recordsAdded = connection.Execute(@"
                INSERT INTO [AuctionExample].[dbo].[Auctions]
                       ([Id]
                       ,[StartingPrice]
                       ,[BidderMemberId]
                       ,[TimeOfBid]
                       ,[MaximumBid]
                       ,[CurrentPrice]
                       ,[AuctionEnds]
                       ,[Version])
                 VALUES
                       (@Id, @StartingPrice, @BidderMemberId, @TimeOfBid,
                      @MaximumBid, @CurrentPrice, @AuctionEnds, @Version)"
                , new { Id = auctionDTO.Id, StartingPrice =
                           auctionDTO.StartingPrice,
                           BidderMemberId = auctionDTO.BidderMemberId,
                         TimeOfBid = auctionDTO.TimeOfBid,
                         MaximumBid = auctionDTO.MaximumBid,
                        CurrentPrice = auctionDTO.CurrentPrice,
                         AuctionEnds = auctionDTO.AuctionEnds,
                         Version = auctionDTO.Version });
```

```
    }
}

public void PersistUpdateOf(IAggregateDataModel  entity)
{
    var auctionDTO = (AuctionDTO)entity;

    using (var connection = new
                SqlConnection(System.Configuration.ConfigurationManager
               .ConnectionStrings["AuctionDB"].ConnectionString))
    {
        var recordsUpdated = connection.Execute(@"
            UPDATE
                [AuctionExample].[dbo].[Auctions]
            SET
                [Id] = @Id
               ,[StartingPrice] = @StartingPrice
               ,[BidderMemberId] = @BidderMemberId
               ,[TimeOfBid] = @TimeOfBid
               ,[MaximumBid] = @MaximumBid
               ,[CurrentPrice] = @CurrentPrice
               ,[AuctionEnds] = @AuctionEnds
               ,[Version] = @Version
            WHERE
                Id = @Id AND Version = @PreviousVersion"
            , new
            {
                Id = auctionDTO.Id,
                StartingPrice = auctionDTO.StartingPrice,
                BidderMemberId = auctionDTO.BidderMemberId,
                TimeOfBid = auctionDTO.TimeOfBid,
                MaximumBid = auctionDTO.MaximumBid,
                CurrentPrice = auctionDTO.CurrentPrice,
                AuctionEnds = auctionDTO.AuctionEnds,
                Version = auctionDTO.Version + 1,
                PreviousVersion = auctionDTO.Version
            });

        if (!recordsUpdated.Equals(1))
        {
            throw new ConcurrencyException();
        }
    }
}

public void Map(AuctionDTO auctionDTO, AuctionSnapshot snapshot)
{
    auctionDTO.Id = snapshot.Id;
    auctionDTO.StartingPrice = snapshot.StartingPrice;
    auctionDTO.AuctionEnds = snapshot.EndsAt;
    auctionDTO.Version = snapshot.Version;

    if (snapshot.WinningBid != null)
    {
```

continues

LISTING 21-112 *(continued)*

```
                auctionDTO.BidderMemberId = snapshot.WinningBid.BiddersId;
                auctionDTO.CurrentPrice = snapshot.WinningBid.CurrentPrice;
                auctionDTO.MaximumBid = snapshot.WinningBid.BiddersMaximumBid;
                auctionDTO.TimeOfBid = snapshot.WinningBid.TimeOfBid;
            }
        }
    }
}
```

LISTING 21-113: The Bid History Repository Implementation

```
using System;
using System.Collections.Generic;
using System.Linq;
using DDDPPP.Chap21.MicroORM.Application.Model.BidHistory;
using DDDPPP.Chap21.MicroORM.Application.Model.Auction;
using DDDPPP.Chap21.MicroORM.Application.Infrastructure.DataModel;
using Dapper;
using System.Data.SqlClient;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
    public class BidHistoryRepository : IBidHistoryRepository,
                                        IUnitOfWorkRepository
    {
        private IUnitOfWork _unitOfWork;

        public BidHistoryRepository(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }

        public int NoOfBidsFor(Guid auctionId)
        {
            int count;

            using (var connection = new
               SqlConnection(System.Configuration.ConfigurationManager
              .ConnectionStrings["AuctionDB"].ConnectionString))
            {
                var count1 = connection.Query<int>(
            "Select Count(*) From BidHistory Where AuctionId = @Id",
                  new { Id = auctionId }).FirstOrDefault();

                count = count1 != null ? count1 : 1;
            }

            return count;
        }

        public void Add(Bid bid)
```

```
{
    var bidHistoryDTO = new BidDTO();

    bidHistoryDTO.AuctionId = bid.AuctionId;
    bidHistoryDTO.Bid = bid.AmountBid.GetSnapshot().Value;
    bidHistoryDTO.BidderId = bid.Bidder;
    bidHistoryDTO.TimeOfBid = bid.TimeOfBid;

    bidHistoryDTO.Id = Guid.NewGuid();

    _unitOfWork.RegisterNew(bidHistoryDTO, this);
}

public BidHistory FindBy(Guid auctionId)
{
    IEnumerable<BidDTO> bidDTOs;

    using (var connection = new
           SqlConnection(System.Configuration.ConfigurationManager
        .ConnectionStrings["AuctionDB"].ConnectionString))
    {
        bidDTOs = connection.Query<BidDTO>(
             "Select * From BidHistory Where AuctionId = @Id",
              new { Id = auctionId });
    }

    var bids = new List<Bid>();

    foreach (var bid in bidDTOs)
    {
        bids.Add(new Bid(bid.AuctionId, bid.BidderId,
                 new Money(bid.Bid), bid.TimeOfBid));
    }

    return new BidHistory(bids);
}

public void PersistCreationOf(IAggregateDataModel entity)
{
    var bidHistoryDTO = (BidDTO)entity;

    using (var connection = new
      SqlConnection(System.Configuration.ConfigurationManager
      .ConnectionStrings["AuctionDB"].ConnectionString))
    {
        connection.Execute(@"
            INSERT INTO [dbo].[BidHistory]
                   ([AuctionId]
                   ,[BidderId]
                   ,[Bid]
                   ,[TimeOfBid]
                   ,[Id])
             VALUES
                   (@AuctionId
                   ,@BidderId
```

continues

LISTING 21-113 *(continued)*

```
                        ,@Bid
                        ,@TimeOfBid
                        ,@Id)"
                    , new
                    {
                        Id = bidHistoryDTO.Id,
                        BidderId = bidHistoryDTO.BidderId,
                        Bid = bidHistoryDTO.Bid,
                        TimeOfBid = bidHistoryDTO.TimeOfBid,
                        AuctionId = bidHistoryDTO.AuctionId
                    });
            }
        }

        public void PersistUpdateOf(IAggregateDataModel entity)
        {
            throw new NotImplementedException();
        }
    }
}
```

To complete the repository implementation, you need to implement the unit-of-work interface. Add a new class to the `Infrastructure` folder named `UnitOfWork`, and use Listing 21-113 to update the newly created class.

LISTING 21-114: The Unit-of-Work Implementation

```
using System;
using System.Collections.Generic;
using System.Transactions;

namespace DDDPPP.Chap21.MicroORM.Application.Infrastructure
{
    public class UnitOfWork : IUnitOfWork
    {
        private Dictionary<IAggregateDataModel, IUnitOfWorkRepository>
                                addedEntities;
        private Dictionary<IAggregateDataModel, IUnitOfWorkRepository>
                                changedEntities;

        public UnitOfWork()
        {
            addedEntities = new Dictionary<IAggregateDataModel,
                                        IUnitOfWorkRepository>();
            changedEntities = new Dictionary<IAggregateDataModel,
                                        IUnitOfWorkRepository>();
        }

        public void RegisterAmended(IAggregateDataModel entity,
                            IUnitOfWorkRepository unitofWorkRepository)
```

```
        {
            if (!changedEntities.ContainsKey(entity))
            {
                changedEntities.Add(entity, unitofWorkRepository);
            }
        }

        public void RegisterNew(IAggregateDataModel entity,
                        IUnitOfWorkRepository unitofWorkRepository)
        {
            if (!addedEntities.ContainsKey(entity))
            {
                addedEntities.Add(entity, unitofWorkRepository);
            };
        }

        public void Clear()
        {
            addedEntities.Clear();
            changedEntities.Clear();
        }

        public void Commit()
        {
            using (TransactionScope scope = new TransactionScope())
            {
                foreach (IAggregateDataModel entity in this.addedEntities.Keys)
                {
                    this.addedEntities[entity].PersistCreationOf(entity);
                }

                foreach (IAggregateDataModel entity in
                                        this.changedEntities.Keys)
                {
                    this.changedEntities[entity].PersistUpdateOf(entity);
                }

                scope.Complete();

                Clear();
            }
        }

    }
}
```

You are required to add a reference to `System.Transactions` so you can use the `TransactionScope` class, as shown in Figure 21-23, which ensures the persistence will commit in an atomic transaction.

The `UnitOfWork` class uses three dictionaries to track pending changes to business entities. The first dictionary corresponds to entities to be added to the datastore. The second dictionary tracks entities to be updated, and the third deals with entity removal. A matching `IUnitOfWorkRepository` is stored against the entity key in the dictionary and is used in the `Commit` method to call the repository, which contains the code to actually persist an entity. The `Commit` method loops through

each dictionary and calls the appropriate `IUnitOfWorkRepository` method, passing a reference to the entity. The work in the `Commit` method is wrapped in a `TransactionScope` using a block; this ensures that no work is done until the `TransactionScope Complete` method is called. If an exception occurs while you are performing work within the `IUnitOfWorkRepository`, all work is rolled back, and the datastore is left in its original state.

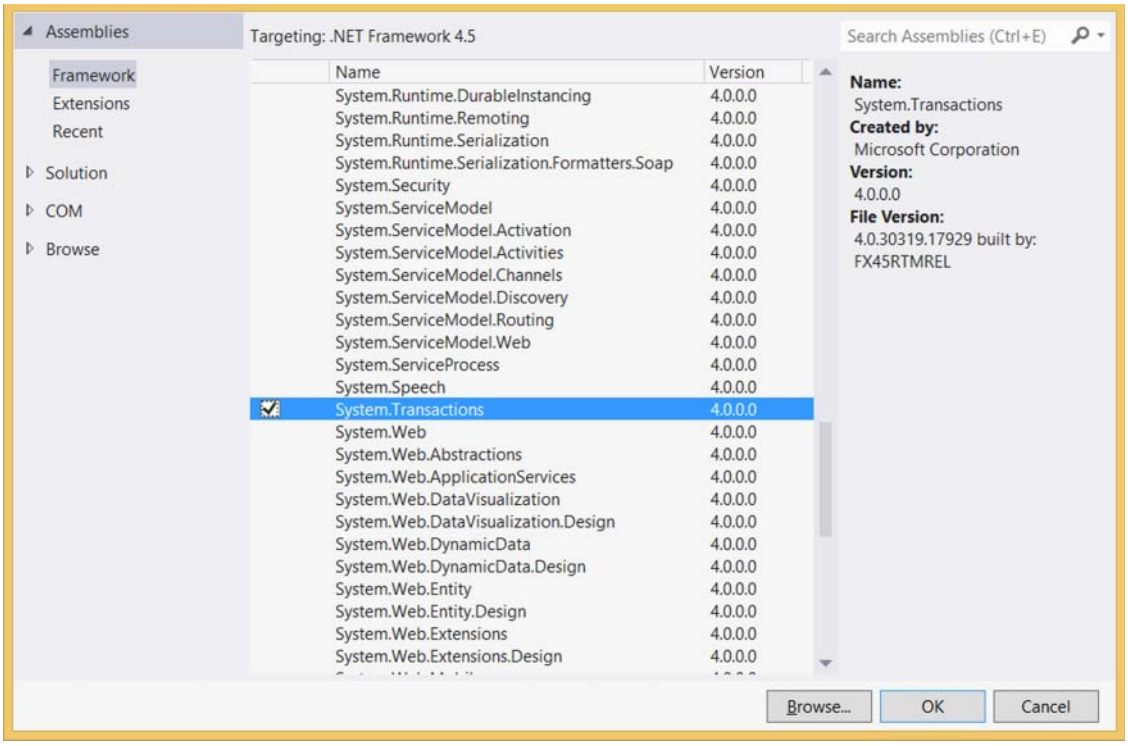


FIGURE 21-23: Add a reference to System.Transactions.

Configuration

Again, you will wire up the dependencies of the application services using StructureMap, which you can install via NuGet just as you did with NHibernate and RavenDB. The code is shown in Listing 21-115.

LISTING 21-115: The Bootstrapper Class for Wiring Up Dependencies

```
using System;
using StructureMap;
using DDDPPP.Chap21.MicroORM.Application.Infrastructure;
using DDDPPP.Chap21.MicroORM.Application.Model.Auction;
using DDDPPP.Chap21.MicroORM.Application.Model.BidHistory;

namespace DDDPPP.Chap21.MicroORM.Application
```

```
{
    public static class Bootstrapper
    {
        public static void Startup()
        {
            ObjectFactory.Initialize(config =>
            {
                config.For<IAuctionRepository>().Use<AuctionRepository>();
                config.For<IBidHistoryRepository>()
                                    .Use<BidHistoryRepository>();
                config.For<IClock>().Use<SystemClock>();

                config.For<IUnitOfWork>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use(x =>
                    {
                        var uow = new UnitOfWork();
                        return uow;
                    });
            });
        }
    }
}
```

To set the connection string to the database, update the `App.Config` within the presentation project to match the XML markup in Listing 21-116.

LISTING 21-116: The app.config File with Connection String Details

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>

  <connectionStrings>
    <add name="AuctionDB" connectionString="Data Source=.\sqlexpress;Initial
Catalog=AuctionExample;Integrated Security=True;MultipleActiveResultSets=True"
providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

You can now run the program and see the same results, as in all the examples.

THE SALIENT POINTS

- The repository mediates between the domain model and the data model; it maps the domain model to a persistence store.
- The repository pattern is a procedural boundary, keeping the domain model free of infrastructural concerns.

- The repository is a contract, not a data access layer. It explicitly states what operations are available for each aggregate—that is, which you can modify and which you can remove.
- The repository is for transactional behavior, not for ad hoc reporting. Using explicit query names provides more information.
- A repository hides the mechanisms for querying the data model and supports the Tell, Don't Ask principle.
- An ORM framework is not a repository. The repository is an architectural pattern, whereas an ORM is a means to represent the data model as an object model. A repository can use an ORM to assist in mediating between the domain model and the data model
- Your persistence framework affects the way your domain model is constructed. Be pragmatic, and don't fight your framework in the quest for needless purity.
- The repository is best used for bounded contexts with a rich domain model. For simpler bounded contexts without complex business logic, use the persistence framework directly.
- When managing transactions, use a unit of work. The unit of work tracks which objects have been removed, added, or updated. The repository is responsible for the actual persistence, performed within a transaction coordinated by the unit of work.
- A repository should be scoped to updating a single aggregate; it should not control a transaction. This should be the responsibility of a unit of work.

22

Event Sourcing

WHAT'S IN THIS CHAPTER?

- An introduction to event sourcing and the problems it solves
- Guidance and examples for building event-sourced domain models
- How to build an Event Store
- Examples of building Event Stores on top of RavenDB and SQL Server
- Examples of using Greg Young's Event Store
- A discussion of how CQRS synergizes with event sourcing
- A list of trade-offs to help you understand when event sourcing is a good choice and when you should avoid it

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 22 download and individually named according to the names throughout the chapter.

Bringing a competitive business advantage and exciting technical challenges, it's clear why a storage mechanism called event sourcing has gained a lot of popularity in recent years. Because a full history of activity is stored, event sourcing allows businesses to deeply understand many aspects of their data, including detailed behavior of their customers. With this historical information, new and novel queries can be asked that inform product development, marketing strategies, and other business decisions. Using event sourcing, you can

determine what the state of the system looked like at any given point in time and how it reached any of those states. For many domains, this is a game-changing capability.

Many systems today store only the current state of the domain model, thereby precluding the opportunity to analyze historical behavior. So, fundamental to event sourcing are new ways of storing data and new ways of building domain models with inherent challenges and advantages. This chapter takes you through concrete step-by-step examples of many such scenarios. You will even see how to build an Event Store and be introduced to a purpose-built Event Store.

Domain-Driven Design (DDD) practitioners often like to combine event sourcing with CQRS as the basis for enhanced scalability and performance. CQRS is fundamentally about denormalization, but such is the synergy with event sourcing that a whole community has formed around the combination. This makes understanding and disambiguating the concepts highly important. It's also important to be realistic in light of all the hype generated by the community. So this chapter helps you understand when event sourcing might not be an efficient choice.

In preparation for the technical examples, this chapter begins by accentuating the importance of event sourcing using comparisons with traditional current-state-only storage. This, and all the other examples in the chapter, are based on a pay-as-you-go service offered by a cell phone network operator. With this service, customers add credit to their account by topping-up their balance with vouchers or Top-Up cards. Credit allows them to make phone calls.

THE LIMITATIONS OF STORING STATE AS A SNAPSHOT

If you only store the current state of your domain model, you don't have a way of understanding how the system reached that state. Consequently, you cannot analyze past behavior to uncover new insights or to work out what has gone wrong. This can be demonstrated by examining a support case presented to the customer service department of a cell phone network operator. A customer reported her dissatisfaction at having an empty balance after she recently topped up and then made only a few short phone calls. She is convinced that she should still have remaining credit. By looking at the database, the customer service assistant pulls up the customer's account information that is shown in Table 22-1.

TABLE 22-1: Customer's Pay-as-You-Go Account Activity

CUSTOMER ID	ALLOWANCE (MINUTES)
123456789	0

Looking at the customer's account in Table 22-1, all the customer service assistant can do is agree that the account balance is empty. It's not possible to work out how the account reached this state by viewing recent call and top-up details. Using the logs from her phone and receipts for her top-ups, the customer supports her argument by providing the history of her account, as shown in Table 22-2.

TABLE 22-2: Customer's Actual Account Activity

PREVIOUS ALLOWANCE (MINUTES)	ACTION	CURRENT ALLOWANCE (MINUTES)
0	Top up $10	20
20	10-minute call	10
15	5-minute call	0

> **NOTE** *This example is intentionally simplistic to accentuate the problems that could arise from storing state as a snapshot. Cell phone network operators would actually keep a log of all calls and top-ups that their customer's made.*

If you observe Table 22-2, it follows that $10 buys 20 minutes of phone calls. Incorrectly, though, the balance is 0 after only a 10-minute and a 5-minute phone call. There is a bug in the system, but the business cannot easily prove it just by looking at its data because it only shows the current state. It does not keep a history of the customer's activity.

By storing state as a snapshot, the cell phone operator also lacks the ability to use the rich history of customer behavior to improve its services. It is unable to correlate user behavior with marketing promotions, and it is unable to look for patterns in user activity that can be capitalized on as new or enhanced streams of revenue.

In this scenario, there is a clear business case for the cell phone network operator to at least consider using event sourcing as a way to record account history. It might be able to gain a competitive advantage and a reduction in customer service complaints just by storing the state of its domain model as a stream of events.

GAINING COMPETITIVE ADVANTAGE BY STORING STATE AS A STREAM OF EVENTS

You can acquire the ability to analyze full historical data by storing each event of significance in chronological order with its timestamp. You then derive current state by replaying events. Significantly, though, you can do more than just work out the current state; you can replay any subsequence of events to work out the state, and activity that caused it, for any point in history. This is known as a *temporal query*—a game-changing capability inherent to event sourcing.

Temporal Queries

Temporal queries are like the ability to travel back in time, because essentially, you can rewind the state of your domain model to a previous point in history. Using a temporal query, the Customer Service department could accurately assess its customer's complaints about an incorrect balance. It

could repeatedly check the state at previous points in time until it found the event that incorrectly deducted double the amount from the allowance. Figure 22-1 illustrates how temporal queries could be applied to the event stream representing the customer's pay-as-you-go account activity detailed previously in Table 22-1.

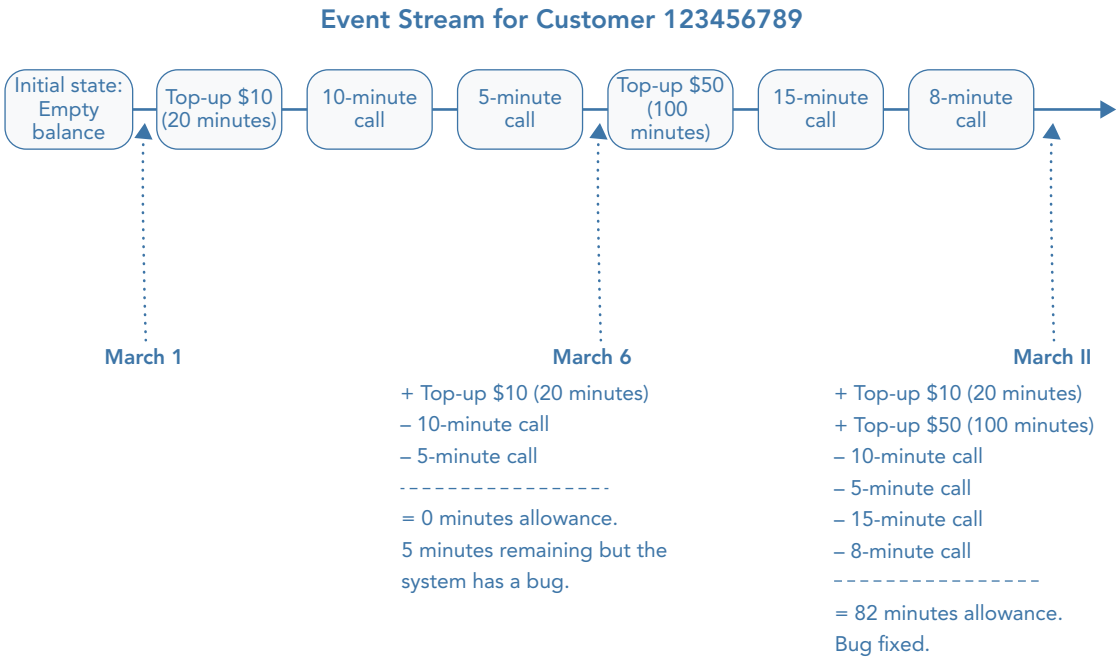


FIGURE 22-1: Calculating the state for any point in history by replaying events.

Replaying events is the mechanism that underlies temporal queries. Figure 22-1 illustrates how subsequences of events are used to calculate the state of the customer's account at two different points in history. Throughout your life and career as a developer, you have been exposed to temporal queries similar to these. Examples include bank accounts or version control systems (VCSs) like Git or Subversion. With each of these concepts, you can rewind state to any point in history by replaying all the events that occurred prior to it. Later in this chapter, you will see examples of performing temporal queries against a real Event Store.

By taking advantage of the event stream in Figure 22-1, the cell phone operator can now ask any question about the history of the account. But, even more interestingly, the operator can also combine information from many customers' accounts to work out behaviors of specific demographics. Such information can be used to make quantitative marketing and product development decisions or guide experimentation. Projections are the underlying feature that enables combining events from multiple streams to carry out these kinds of complex temporal queries.

Projections

A limitation of some event stores is a difficulty in carrying out ad hoc queries against multiple event streams, which in a SQL database would be achievable with straightforward joins. To get around this problem, event stores use a concept called *projections*, which are queries that map a set of input event streams onto one or more new output streams. For example, a cell phone operator can project the event streams for many of its customers to answer questions like these: "What was the total number of minutes used on a specific day by a specific demographic around the time of a certain sporting event" and "Did the total usage increase or decrease among a certain demographic when there was a special offer?" Figure 22-2 illustrates how arbitrary questions like these can be asked by combining events from multiple streams using projections.

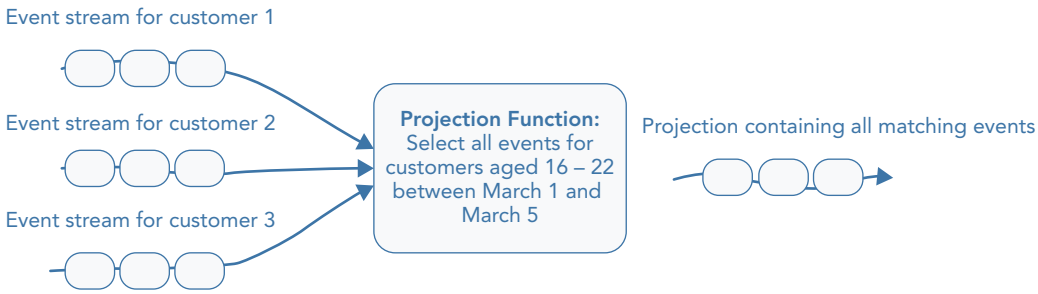


FIGURE 22-2: Creating projections from multiple event streams.

In Figure 22-2, each event stream that represents the account of a single customer is fed into the projection function. A projection function can do a number of things, including keeping state or emitting new events. The projection function in Figure 22-2 is going to emit all events that occurred between March 1 and March 5, and where the account holder is between 16 and 24 years of age, into a new event stream. This new stream is the projection. With all the required events in a single stream, it is then easy and efficient to query all of them to find out totals, averages, or any other piece of information that needs to be extracted.

You can project a set of input streams onto more than one output. As you'll see later in this chapter, with Greg Young's Event Store, you have a lot of power and flexibility. You can use the rich capabilities of JavaScript in projection functions.

> **NOTE** *You will also see examples of projections used to create reports from Event Store in Chapter 26: "Queries: Domain Reporting."*

Snapshots

A consequence of storing state as events is that event streams can grow very large, meaning that the time to replay events can continue to increase significantly. To avoid this performance hit, event

stores use snapshots. As Figure 22-3 illustrates, snapshots are intermediate steps in an event stream that represent the state after replaying all previous events.

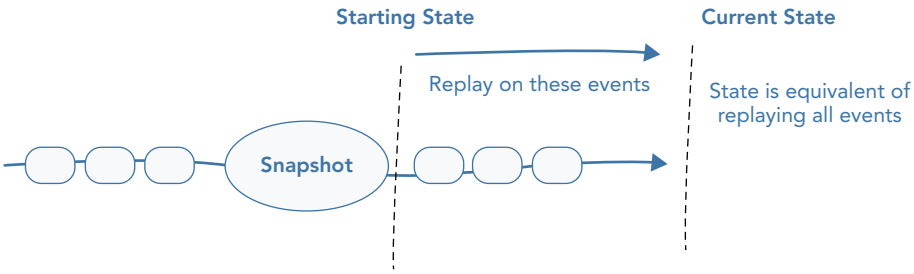


FIGURE 22-3: Efficiently restoring state by using snapshots.

When an application wants to load the current state of an aggregate from an event stream, all it has to do is find the latest snapshot. Using that snapshot as the initial state, it need then only replay all the subsequent events in the stream.

> **NOTE** *Throughout this chapter, the terms* event stream *and* stream *are used interchangeably, as are* domain event *and* event. *This is consistent with the way event sourcing practitioners use the terms.*

EVENT-SOURCED AGGREGATES

For compatibility with event sourcing, aggregates need to be event oriented. Importantly, they need to be able to calculate their state by applying a series of events. DDD practitioners find that a nice side effect of this is that their aggregates are more behavior oriented, providing heightened levels of domain event expressiveness. Another satisfying benefit is that persistence tends to be loosely coupled and less problematic.

Structuring

There are a few key details involved when creating aggregates for event-sourced domain models. Of these details, the most important is for an aggregate to be able to apply a domain event and update its state according to the appropriate business rule(s). Second, a list of uncommitted events needs to be maintained so that they can be persisted to an Event Store. Third, an aggregate needs to maintain a record of its version and provide the ability to create snapshots and restore from them.

Adding Event-Sourcing Capabilities

It's often useful to create a base class, as per Listing 22-1, which all your aggregates inherit from to encapsulate commonality.

LISTING 22-1: Aggregate Base Class Providing Event-Sourcing Commonality

```
public abstract class EventSourcedAggregate : Entity
{
    public List<DomainEvent> Changes { get; private set; }
    public int Version { get; protected set; }

    public EventSourcedAggregate()
    {
        Changes = new List<DomainEvent>();
    }

    public abstract void Apply(DomainEvent changes);
}
```

`Changes` is the collection of uncommitted events, and obviously `Version` keeps track of the aggregate's version, which will be explained in more detail shortly. `Apply()` takes a `DomainEvent` that aggregates must handle by applying business rules and updating state.

Listing 22-2 shows a concrete (partial) aggregate implementation that represents a customer's pay-as-you-go account. This illustrates the expressiveness that can be layered on top of `Apply()`.

LISTING 22-2: Event-Sourced Aggregates Can Be Almost Declarative of Business Rules

```
public class PayAsYouGoAccount : EventSourcedAggregate
{
    private FreeCallAllowance _freeCallAllowance;
    private Money _credit;
    private PayAsYouGoInclusiveMinutesOffer _inclusiveMinutesOffer =
        new PayAsYouGoInclusiveMinutesOffer();

    public PayAsYouGoAccount()
    { }

    public override void Apply(DomainEvent @event)
    {
        When((dynamic)@event);
        Version = Version ++;
    }

     private void When(CreditAdded creditAdded)
     {
        _credit = _credit.Add(creditAdded.Credit);
```

continues

LISTING 22-2 *(continued)*

```
        }

        private void When(PhoneCallCharged phoneCallCharged)
        {
            _credit = _credit.Subtract(phoneCallCharged.CostOfCall);

            if (_freeCallAllowance != null)
                _freeCallAllowance.Subtract(phoneCallCharged.CoveredByAllowance);
        }

        private void When(AccountCreated accountCreated)
        {
            Id = accountCreated.Id;
            _credit = accountCreated.Credit;
        }

         ...
    }
```

The implementation of `Apply()` in Listing 22-2 uses a tiny amount of dynamic magic that allows highly expressive event-handling methods. In Listing 22-2, these handlers are the `When()` methods. Each overload of `When()` is intended to read almost declaratively, expressing what business rules should apply and how the state should change when each type of event occurs. `Apply()` also updates the `Version` field each time an event is applied. This is helpful when working out whether an aggregate has changed before an operation is carried out on it. An aggregate's version is the sequence number, relative to the start of the aggregate's event stream, of the last event that has been passed into its `Apply()`.

Exposing Expressive Domain-Focused APIs

`Apply()` is mostly an implementation detail that won't be called outside the aggregate. The reason it is public is to enable the aggregate to be rehydrated (explained shortly). Outside of an aggregate, services still communicate through expressive APIs, being unaware of event-related features. This is exemplified by `TopUp()` in the updated `PayAsYouGoAccount` aggregate shown in Listing 22-3.

LISTING 22-3: Updated PasAsYouGoAccount Aggregate with Expressive Domain-Focused API

```
public class PayAsYouGoAccount : EventSourcedAggregate
{
    private FreeCallAllowance _freeCallAllowance;
    private Money _credit;
    private PayAsYouGoInclusiveMinutesOffer _inclusiveMinutesOffer =
        new PayAsYouGoInclusiveMinutesOffer();

    public PayAsYouGoAccount()
```

```
    { }

    ...

    public void TopUp(Money credit, IClock clock)
    {
        if (_InclusiveMinutesOffer.IsSatisfiedBy(credit))
            Causes(new CreditSatisfiesFreeCallAllowanceOffer(
                this.Id, clock.Time(), _inclusiveMinutesOffer.FreeMinutes)
            );

        Causes(new CreditAdded(this.Id, credit));
    }

    private void Causes(DomainEvent @event)
    {
        Changes.Add(@event);
        Apply(@event);
    }

    ...
}
```

Listing 22-3 shows the view of an aggregate that external domain and application services would see. They would not know that the aggregate used event sourcing; they would only see expressive high-level APIs like `TopUp()` that clearly express domain concepts. From this perspective, the similarity to non-event-sourced aggregates is similar.

To clarify that events are mostly an implementation detail and that interaction with an aggregate from the service layer will be similar, Listing 22-4 presents the `TopUpCredit` application service. Try to focus on how there is no trace of event sourcing and how this would look almost identical to interaction with a non-event-sourced aggregate.

LISTING 22-4: Interacting with Aggregates from within Application Services via Expressive High-Level APIs

```
// application service
public class TopUpCredit
{
    private IPayAsYouGoAccountRepository _payAsYouGoAccountRepository;
    private IDocumentSession _unitOfWork;
    private IClock _clock;

    public TopUpCredit(IPayAsYouGoAccountRepository payAsYouGoAccountRepository,
                       IClock clock)
    {
        _payAsYouGoAccountRepository = payAsYouGoAccountRepository;
        _clock = clock;
```

continues

LISTING 22-4 *(continued)*

```
    }

    public void Execute(Guid id, decimal amount)
    {
        ...

        var account = _payAsYouGoAccountRepository.FindBy(id);
        var credit = new Money(amount);

        // .. expressive domain-focused API - no hint of event sourcing
        account.TopUp(credit, _clock);
        _payAsYouGoAccountRepository.Save(account);

        ...
     }
}
```

When `TopUp()` is invoked, either the `CreditSatisfiedFreeCallAllowanceOffer` or the `CreditAdded` domain event will occur based on the `InclusiveMinutesOfferBusiness` policy. Importantly, the outcome of these actions will result in a new event being raised. This new event will need to cause an update of the aggregate's state, after being passed into `Apply()`. But the event will also need to be persisted so that it can be replayed anytime the aggregate is loaded from its event stream. `Causes()` takes care of these concerns by adding the event to `Changes`—the collection of uncommitted events—and then feeding the event into `Apply()`.

Adding Snapshot Support

One remaining aggregate detail that needs to be addressed before persistence can be tackled is the ability to create snapshots. The aggregates themselves need to create the snapshots, because they contain the domain logic that builds state by replaying all previous events. Accordingly, it's convenient to add a method onto aggregates that builds a snapshot and one that restores the aggregate. An example of each is shown in Listing 22-5.

LISTING 22-5: Adding Snapshot-Creation Support to Aggregates

```
public class PayAsYouGoAccount : EventSourcedAggregate
{
    private FreeCallAllowance _freeCallAllowance;
    private Money _credit;
    private PayAsYouGoInclusiveMinutesOffer _inclusiveMinutesOffer =
        new PayAsYouGoInclusiveMinutesOffer();

    ...

    // constructor overload - restore aggregate from snapshot
    public PayAsYouGoAccount(PayAsYouGoAccountSnapshot snapshot)
    {
        Version = snapshot.Version;
        _credit = new Money(snapshot.Credit);
```

```
        }

        public PayAsYouGoAccountSnapshot GetPayAsYouGoAccountSnapshot()
        {
            return new PayAsYouGoAccountSnapshot
            {
                Version = Version,
                Credit = _credit.Amount
            };
        }

        ...
    }
```

In Listing 22-5, the constructor overload that takes a `PayAsYouGoSnapshot` restores the state of the aggregate to match the state of the `PayAsYouGoAccountSnapshot`. As previously discussed, this is a performance-improving shortcut that is equivalent to applying all the events that occurred before the snapshot was created. For this to work, though, snapshots need to accurately represent an aggregate's state. You can see in Listing 22-5 that `GetPayAsYouGoAccountSnapshot()` ensures this happens by creating a snapshot and using the correct values to populate it. The `PayAsYouGoAccountSnapshot` is shown in Listing 22-6.

LISTING 22-6: Snapshot Containing All of an Aggregate's State

```
public class PayAsYouGoAccountSnapshot
{
    public int Version { get; set; }

    public decimal Credit { get; set; }
}
```

Persisting and Rehydrating

Persisting event-sourced aggregates is just a case of storing the uncommitted changes in an event store. Similarly, loading the aggregate, also known as *rehydrating*, requires you to load and replay all previously stored events, with the option to use snapshots as a shortcut. You saw in the previous examples that `Changes` and `Apply()` are used for these purposes, respectively.

Creating an Event-Sourcing Repository

Listing 22-7 contains an example repository implementation that handles loading and saving `PayAsYouGoAccount` aggregates (initially ignoring snapshots).

LISTING 22-7: Repository for Persisting and Rehydrating Event-Sourced Aggregates

```
public class PayAsYouGoAccountRepository : IPayAsYouGoAccountRepository
{
    private readonly IEventStore _eventStore;

    public PayAsYouGoAccountRepository(IEventStore eventStore)
```

continues

LISTING 22-7 *(continued)*

```
        {
            _eventStore = eventStore;
        }

        public void Add(PayAsYouGoAccount payAsYouGoAccount)
        {
            var streamName = StreamNameFor(payAsYouGoAccount.Id);

            _eventStore.CreateNewStream(streamName, payAsYouGoAccount.Changes);
        }

        public void Save(PayAsYouGoAccount payAsYouGoAccount)
        {
            var streamName = StreamNameFor(payAsYouGoAccount.Id);

            _eventStore.AppendEventsToStream(streamName, payAsYouGoAccount.Changes);
        }

        public PayAsYouGoAccount FindBy(Guid id)
        {
            var streamName = StreamNameFor(id);
            var fromEventNumber = 0;
            var toEventNumber = int.MaxValue ;

            var stream = _eventStore.GetStream(
                streamName, fromEventNumber, toEventNumber
            );

            var payAsYouGoAccount = new PayAsYouGoAccount();
            foreach(var @event in stream)
            {
                payAsYouGoAccount.Apply(@event);
            }

            return payAsYouGoAccount;
        }

        private string StreamNameFor(Guid id)
        {
            // stream per-aggregate: {AggregateType}-{AggregateId}
            return string.Format("{0}-{1}", typeof(PayAsYouGoAccount).Name, id);
        }
    }
```

In Listing 22-7, the three major operations that repositories need to support are shown: creating streams, appending to streams, and loading streams. Creating an event stream involves creating a new stream with an initial set of events, as `Add()` shows. In a similar fashion, `Save()` appends the uncommitted events of a `PayAsYouGoAccount` aggregate to an existing stream. Lastly, `FindBy()` loads a `PayAsYouGoAccount` given its ID.

Loading aggregates is a little more complex than creating or updating because you have to decide which events should be loaded from the stream by specifying the highest and

lowest version number (useful for reloading an aggregate to a previous state). Once all the events are pulled back from the Event Store, they are passed into the aggregate's `Apply()` so that the aggregate's state will be built. This is the only time when an external service should call `Apply()`.

A detail that's important to discern from Listing 22-7 is the name of the event stream that has the format `{AggregateType}-{AggregateId}`, and for good reason. Having a stream per aggregate means you only have to replay events for a single aggregate when loading it—not events for every aggregate of that type. This has significant performance advantages and provides the platform for better scalability. It can also make life easier when snapshots are involved.

Adding Snapshot Persistence and Reloading

Snapshots are a performance enhancement to avoid loading the entire history of events in a stream, as previously mentioned. Snapshots are usually created by a background process such as a Windows service. A basic example of a background job that creates snapshots for the `PayAsYouGoAccount` aggregate, using a RavenDB-based event store, is shown in Listing 22-8.

LISTING 22-8: An Example Background Process That Periodically Creates Snapshots

```
public class PasAsYouGoAccountSnapshotJob
{
    private IDocumentStore _documentStore;

    public PasAsYouGoAccountSnapshotJob(IDocumentStore documentStore)
    {
        this._documentStore = documentStore;
    }

    public void Run()
    {
        while(true)
        {
            foreach (var id in GetIds())
            {
                using (var session = _documentStore.OpenSession())
                {
                    var repository = new PayAsYouGoAccountRepository(
                        new EventStore(session)
                    );
                    var account = repository.FindBy(Guid.Parse(id));
                    var snapshot = account.GetPayAsYouGoAccountSnapshot();
                    repository.SaveSnapshot(snapshot, account);
                }
            }

             // create a new snapshot for each aggregate every 12 hours
            Thread.Sleep(TimeSpan.FromHours(12));
        }
```

continues

LISTING 22-8 *(continued)*

```
    }

    private IEnumerable<string> GetIds()
    {
        using (var session = _documentStore.OpenSession())
        {
            return session.Query<EventStream>()
                          .Select(x => x.Id)
                          .ToList();
        }
    }
}
```

Essentially, the background job in Listing 22-8 is going to loop through the IDs of all aggregates and create a snapshot for each of them every 12 hours. In a production-quality version, you may want to iterate through smaller sets of IDs, you may want to add logging, and you will likely want to adjust the 12-hour period to something more optimal for your environment. It's also likely that you may want to create snapshots based on some condition, such as the number of events that have occurred since the last snapshot.

Once you are generating and storing snapshots, you can optimize your loading strategy by using the latest snapshot as the initial state. Listing 22-9 shows an updated `PayAsYouGoRepository.FindBy()` implementation that takes advantage of snapshots.

LISTING 22-9: Snapshot-Aware Aggregate Loading

```
public PayAsYouGoAccount FindBy(Guid id)
{
    var streamName = StreamNameFor(id);

    var fromEventNumber = 0;
    var toEventNumber = int.MaxValue ;

    var snapshot = _eventStore.GetLatestSnapshot<PayAsYouGoAccountSnapshot>(
        streamName
    );

    if (snapshot != null)
    {
        fromEventNumber = snapshot.Version + 1; // load only events after snapshot
    }

    var stream = _eventStore.GetStream(streamName, fromEventNumber, toEventNumber);

    PayAsYouGoAccount payAsYouGoAccount = null;
    if (snapshot != null)
    {
        payAsYouGoAccount = new PayAsYouGoAccount(snapshot);
    }
    else
```

```
        {
            payAsYouGoAccount = new PayAsYouGoAccount();
        }

        foreach(var @event in stream)
        {
            payAsYouGoAccount.Apply(@event);
        }

        return payAsYouGoAccount;
    }
```

You can see in Listing 22-9 that `FindBy()` has been updated with a call to `_eventStore.GetLatestSnapshot()`. If this method returns a snapshot, the `fromEventNumber` is bumped up to the version after the snapshot. This ensures that subsequent calls to `_eventStore.GetStream()` will only retrieve events that occurred after that snapshot. You can also see the other change; if a snapshot is found, the `PayAsYouGoAccount` aggregate is created with the snapshot as the initial state.

Handling Concurrency

Sometimes you want the operation of adding an event to a stream to fail. This is often the case when multiple users are updating the state of an aggregate at the same time, and you want to avoid a change being committed if the aggregate has updated after the new change was requested but not committed. You might already be familiar with this concept: optimist concurrency (`http://technet.microsoft.com/en-us/library/aa213031(v=sql.80).aspx`).

In an event-sourced application, you can implement optimistic concurrency with two additional steps. First, when an aggregate is loaded, it keeps a record of the version number it had at the start of the transaction. Second, just before any new events are appended to the stream, a check is carried out to ensure that the last persisted version number matches the version number the aggregate had at the start of the transaction. Listing 22-10 and Listing 22-11 illustrate this process.

LISTING 22-10: Aggregate Updated with an Initial Version Number

```
public class PayAsYouGoAccount : EventSourcedAggregate
{
    ...

    // once set, does not change
    public int InitialVersion { get; private set; }

    public PayAsYouGoAccount(PayAsYouGoAccountSnapshot snapshot)
    {
        Version = snapshot.Version;
        InitialVersion = snapshot.Version;
        _credit = new Money(snapshot.Credit);
    }

    ...
}
```

LISTING 22-11: Checking if the Aggregate Has Recently Been Updated by Telling the Event Store Which Version Number Should Be the Most Recently Persisted

```
public class PayAsYouGoAccountRepository : IPayAsYouGoAccountRepository
{
    ...

    public void Save(PayAsYouGoAccount payAsYouGoAccount)
    {
        var streamName = StreamNameFor(payAsYouGoAccount.Id);

        var expectedVersion = GetExpectedVersion(payAsYouGoAccount.InitialVersion);
        _eventStore.AppendEventsToStream(streamName, payAsYouGoAccount.Changes,
                expectedVersion);
    }

    private int? GetExpectedVersion(int expectedVersion)
    {
        if (expectedVersion == 0)
        {
            // first time the aggregate is stored, there is no expected version
            return null;
        }
        else
        {
            return expectedVersion;
        }
    }

    ...
}
```

Listing 22-10 contains an updated `PayAsYouGoAccount` aggregate with an `InitialVersion` property, representing the version of the last event that was persisted prior to the aggregate being loaded. This value is then passed into the Event Store's `AppendEventsToStream()`, as shown in Listing 22-11, where the Event Store implementation can then check to ensure this version is still the most up to date. Later in the chapter, you will see an event store implementation that shows how to fully implement this feature.

Testing

Unit testing event-sourced aggregates involves asserting that events have occurred. This is done by checking the collection of uncommitted events belonging to the aggregate. An example demonstrating this for `PayAsYouGoAccount.TopUp()` is shown in Listing 22-12.

LISTING 22-12: Unit Testing Event-Sourced Aggregates by Asserting That Events Occurred

```
[TestClass]
public class PayAsYouGoAccount_Tests
{
    static PayAsYouGoAccount _account;
```

```
        static Money _fiveDollars = new Money(5);
        static PayAsYouGoInclusiveMinutesOffer _free90MinsWith10DollarTopUp =
            new PayAsYouGoInclusiveMinutesOffer(
                new Money(10), new Minutes(90)
            );

        [ClassInitialize] // runs first
        public static void When_applying_a_top_up_that_does_not_qualify_for_inclusive
    _minutes(TestContext ctx)
        {
            _account = new PayAsYouGoAccount(Guid.NewGuid(), new Money(0));

            // remove the AccountCreated event that is not relevant to this test
            _account.Changes.Clear();

            _account.AddInclusiveMinutesOffer(_free90MinsWith10DollarTopUp);

            // $5 top up does not meet $10 threshold for free minutes
            _account.TopUp(_fiveDollars, new SystemClock());
        }

        [TestMethod]
        public void The_account_will_be_credited_with_the_top_up_amount_but_no_free
    _minutes()
        {
            var lastEvent = _account.Changes.SingleOrDefault() as CreditAdded;
            Assert.IsNotNull(lastEvent);
            Assert.AreEqual(_fiveDollars, lastEvent.Credit);
            Assert.AreEqual(5, _account.GetPayAsYouGoAccountSnapshot().Credit);
        }
    }
```

After a `PayAsYouGoAccount` has an inclusive minutes offer applied to it, if the customer tops up by the inclusive minute offer's threshold, the customer is given free credits in addition to her top-up amount. The test in Listing 22-12 verifies that when a top up does not meet the criteria, no free minutes are applied. It does this by asserting that only a single `CreditAdded` event exists within the aggregate's uncommitted events, and the state of the aggregate has been increased only by the amount of the top up. You can see this with the calls to `Assert.Equal()` at the bottom of Listing 22-12.

Listing 22-12 is a common case when testing event-sourced aggregates; you will interact with them through expressive high-level application programming interfaces (APIs), like `TopUp()`. You will then assert that events were added to the uncommitted list, and state changed accordingly. This will often apply to both high-level integration tests and low-level unit tests. The major difference is whether you mock collaborators and external services or pass in real implementations.

BUILDING AN EVENT STORE

To build applications that use event sourcing, you can use a purpose-built event store or create one of your own. Creating one of your own really just means using an existing tool in a novel way. So in this section, you will see how you can use RavenDB and SQL Server as the basis for event

stores. Creating your own event store is also a hands-on way to increase your understanding of the core concepts.

You saw the `IEventStore` interface being used previously by the `PayAsYouGoAccoutRepository` in Listing 22-11. This is the interface that the Event Store being created in this section will implement. `IEventStore` is shown in its entirety in Listing 22-13.

LISTING 22-13: IEventStore Interface

```
public interface IEventStore
{
    void CreateNewStream(string streamName, IEnumerable<DomainEvent> domainEvents);

    void AppendEventsToStream(string streamName, IEnumerable<DomainEvent>
domainEvents, int? expectedVersion);

    IEnumerable<DomainEvent> GetStream(string streamName, int fromVersion, int
toVersion);

    void AddSnapshot<T>(string streamName, T snapshot);

    T GetLatestSnapshot<T>(string streamName) where T: class;
}
```

WARNING *Abstracting your data access comes with known trade-offs. An abstraction gives you the opportunity to switch technologies by providing an alternative implementation. Conversely, creating an abstraction specifies a strict interface; it may block the ability to use some advanced features of each data access technology. Being aware of these costs, and how likely you are to want to change the underlying technology, helps guide your decision about whether to abstract. Later in this chapter is a discussion of why event sourcing can be less of a barrier to persistence ignorance.*

Designing a Storage Format

Designing or choosing a storage format is one of the big challenges when building event store functionality. Using your chosen technology, at a minimum you need to provide the ability to create event streams, append events to them, and pull those events back out again in the same order. It's also likely that you will want to support snapshotting.

In this example, the storage format is based on three document types: `EventStream`, `EventWrapper`, and `SnapshotWrapper`. `EventStream` represents an event stream but only contains meta data such as its `Id` and `Version`. `EventWrapper` wraps and represents an individual domain event, containing all the event's data plus meta data about the stream it belongs to. `SnapshotWrapper` represents a single snapshot. An alternative strategy would have been to have a single document that contained

the meta data, events, and snapshots for each aggregate. You are free to design and refine your own format as you see fit.

RavenDB is a document database in which each document is a JavaScript Object Notation (JSON) object. Handily, Raven's C# client library automatically converts Plain Old C# Object (POCO) classes into the required JSON for you. This means that the three document types to be used as the basis for event store functionality need only be declared in code. `EventStream`, `EventWrapper`, and `SnapshotWrapper` are shown in Listings 22-14 through 22-16, respectively.

LISTING 22-14: EventStream

```
public class EventStream
{
    public string Id { get; private set; } //aggregate type + id
    public int Version {get; private set;}

    private EventStream() { }

    public EventStream(string id)
    {
        Id = id;
        Version = 0;
    }

    public EventWrapper RegisterEvent(DomainEvent @event)
    {
        Version ++;

        return new EventWrapper(@event, Version, Id);
    }
}
```

LISTING 22-15: EventWrapper

```
public class EventWrapper
{
    public string Id { get; private set; }
    public DomainEvent Event { get; private set; }
    public string EventStreamId { get; private set; }
    public int EventNumber { get; private set; }

    public EventWrapper(DomainEvent @event, int eventNumber, string streamStateId)
    {
        Event = @event;
        EventNumber = eventNumber;
        EventStreamId = streamStateId;
        Id = string.Format("{0}-{1}", streamStateId, EventNumber);
    }
}
```

LISTING 22-16: SnapshotWrapper

```
public class SnapshotWrapper
{
    public string StreamName { get; set; }

    public Object Snapshot { get; set; }

    public DateTime Created { get; set; }
}
```

Creating Event Streams

Now that you've chosen a document type to represent an event stream, creating an instance of an event stream involves creating a document of that type. This is demonstrated with the initial implementation of an event store that supports the creation of streams, as shown in Listing 22-17.

LISTING 22-17: An Event Store Implementation That Creates Streams

```
public class EventStore : IEventStore
{
    private readonly IDocumentSession _documentSession;

    public EventStore(IDocumentSession documentSession)
    {
        _documentSession = documentSession;
    }

    public void CreateNewStream(string streamName,
            IEnumerable<DomainEvent> domainEvents)
    {
        var eventStream = new EventStream(streamName);
        _documentSession.Store(eventStream);

        AppendEventsToStream(streamName, domainEvents);  // see next listing
    }
}
```

RavenDB is the underlying storage technology represented by its `IDocumentSession` interface. You can see the implementation of `CreateNewStream()` in Listing 22-17 is using an `IDocumentSession` to store an `EventStream` object. As mentioned, RavenDB converts this `EventStream` object to JSON and creates a new document for it. Conceptually, at the point RavenDB creates the `EventStream` document, the new event stream has been created. However, it doesn't contain any events.

Appending to Event Streams

Once event streams have been created, the next challenge is to provide the ability to append events to them. In this example, an event is represented by an `EventWrapper`, a class that wraps a domain event with meta data so that when it is persisted, it can be associated with the correct event stream.

An updated implementation of `EventStore` that supports appending `EventWrappers` is shown in Listing 22-18.

LISTING 22-18: Adding the Ability to Append Events to Streams

```
public class EventStore : IEventStore
{
    ...

    public void AppendEventsToStream(string streamName,
        IEnumerable<DomainEvent> domainEvents, int? expectedVersion = null)
    {
        var stream = _documentSession.Load<EventStream>(streamName);

        foreach (var @event in domainEvents)
        {
            _documentSession.Store(stream.RegisterEvent(@event));
        }
}
```

`AppendEventsToStream()` stores each `EventWrapper` as a unique document. But this is an implementation detail; all callers of `AppendEventsToStream()` know is that the event was appended to the appropriate stream. However, because each event is a unique document, there needs to be some way of retrieving all events for a given stream. `EventWrapper` contains the meta data that enables this: the ID of the stream and an event version number. These values are populated by `EventStream.RegisterEvent()`, as shown previously in Listing 22-14. As you'll see in the next example, querying for events in a stream is then mainly a case of searching for events whose `EventStreamId` matches the name of the stream whose events are being queried.

Querying Event Streams

When building event store functionality, as a minimum you need to include the ability to query for all the events in a stream. You will also need to provide the ability to query by the version or ID of an event to support snapshot loading. An updated implementation of `EventStore` that provides these capabilities is shown in Listing 22-19.

LISTING 22-19: Adding the Ability to Load Events from Streams

```
public class EventStore : IEventStore
{

    ...

    public IEnumerable<DomainEvent> GetStream(string streamName,
        int fromVersion, int toVersion)
    {
        // get events from a specific version
```

continues

LISTING 22-19 *(continued)*

```
        var eventWrappers = (from stream in _documentSession.Query<EventWrapper>()
                             .Customize(x => x.WaitForNonStaleResultsAsOfNow())
                             where stream.EventStreamId.Equals(streamName)
                             && stream.EventNumber <= toVersion
                             && stream.EventNumber >= fromVersion
                             orderby stream.EventNumber
                             select stream).ToList();

            if (eventWrappers.Count() == 0) return null;

            var events = new List<DomainEvent>();

            foreach (var @event in eventWrappers)
            {
                events.Add(@event.Event);
            }

            return events;
        }

        ...
}
```

As previously discussed, the convention for representing event streams in this example is to store each event as a unique `EventWrapper` document containing the ID of the stream it belongs to. You can see in Listing 22-19 that the `where` clause adheres to this convention by specifying that only `EventWrappers` with the passed-in stream name should be selected.

In addition to loading all events for a stream, the `fromVersion` and `toVersion` parameters are used as part of the query. They specify the earliest and latest event that should be pulled back from the stream, respectively. As you will see in the next example, this is an important capability when snapshots are involved.

Adding Snapshot Support

To enable snapshots, you must provide the facility to store them. In this example, `SnapshotWrapper` is used to wrap snapshots with additional meta data—the name of the stream it belongs to and the date it was added. Listing 22-20 demonstrates an updated version of `EventStore` with the ability to save snapshots.

LISTING 22-20: Adding the Ability to Save Snapshots

```
public class EventStore : IEventStore
{
    ...

    public void AddSnapshot<T>(string streamName, T snapshot)
    {
        var wrapper = new SnapshotWrapper
        {
```

```
            StreamName = streamName,
            Snapshot = snapshot,
            Created = DateTime.Now
        };

        _documentSession.Store(snapshot);
    }

    ...
}
```

`AddSnapshot<T>()` allows consumers to store a snapshot for a given stream by wrapping the snapshot with a `SnapshotWrapper` that contains the name of the stream it belongs to and the time it was saved. As Listing 22-21 shows, these two pieces of meta data allow consumers of `EventStore` to get the latest snapshot for any stream simply by passing in the name of the stream.

LISTING 22-21: Finding the Latest Snapshot by Supplying Just the Name of the Stream

```
public class EventStore : IEventStore
{
    ...

    public void AddSnapshot<T>(string streamName, T snapshot)
    {
        var wrapper = new SnapshotWrapper
        {
            StreamName = streamName,
            Snapshot = snapshot,
            Created = DateTime.Now
        };

        _documentSession.Store(snapshot);
    }

    public T GetLatestSnapshot<T>(string streamName) where T: class
    {
        var latestSnapshot = _documentSession.Query<SnapshotWrapper>()
                        .Customize(x => x.WaitForNonStaleResultsAsOfNow())
                        .Where(x => x.StreamName == streamName)
                        .OrderByDescending(x => x.Created)
                        .FirstOrDefault();

        if (latestSnapshot == null)
        {
            return null;
        }
        else
        {
            // unwrap snapshot - give user what he passed to AddSnapshot
            return (T)latestSnapshot.Snapshot;
        }
    }

}
```

Hopefully, you can see the benefits of using `SnapshotWrapper` by observing the code in Listing 22-21. Its `StreamName` and `Created` fields are used to build a query that finds the latest saved snapshot for the name of the passed-in stream. In combination with `AddSnapshot<T>()`, this is a clean API for storing a snapshot of any type and pulling it out later. But this is just one possible solution; you may find alternative solutions that fit your needs better. Creative thinking is encouraged.

Managing Concurrency

Many modern applications are multiuser, meaning that different users view and update the same piece of data at the same time. If one user attempts to update some data but another has changed it without the other user realizing, it's sometimes essential that the update fails. As mentioned, this is optimistic concurrency. A generic way of handling optimistic concurrency in event stores is to verify that the version number of the last stored event matches the version number that the new update is based on.

Keeping track of the expected version at the application level was shown previously in Listing 22-11, where the `InitialVersion` property was added to the aggregate and then passed into the event store each time new events were appended to the stream. Listing 22-22 shows how an updated version of `EventStore.AppendEventsToStream()` uses this version number to abort saving any events if the expected version number does not match the version number of the last stored event—meaning a new event(s) has been added since this transaction began, and thus the user was probably not aware of it.

LISTING 22-22: Add Optimistic Concurrency Checks When Storing Events

```
public void AppendEventsToStream(string streamName,
   IEnumerable<DomainEvent> domainEvents, int? expectedVersion = null)
{
    var stream = _documentSession.Load<EventStream>(streamName);

    if (expectedVersion != null)
    {
        CheckForConcurrencyError(expectedVersion, stream);
    }

    foreach (var @event in domainEvents)
    {
        _documentSession.Store(stream.RegisterEvent(@event));
    }
}

private static void CheckForConcurrencyError(int? expectedVersion,
    EventStream stream)
{
    var lastUpdatedVersion = stream.Version;
    if (lastUpdatedVersion != expectedVersion)
    {
        var error = string.Format("Expected: {0}. Found: {1}", expectedVersion,
lastUpdatedVersion);
        throw new OptimsticConcurrencyException(error);
    }
}
```

If an expected version number is passed into `AppendEventsToStream`, the `EventStore` ensures that before appending new events, the passed-in version number matches the version number of the last stored event. This is shown in Listing 22-22. In particular, the private `CheckForConcurrencyError()` carries out the comparison logic and throws an `OptimisticConcurrencyException` if the version numbers aren't the same.

Unfortunately, the solution in Listing 22-22 is not robust enough when used with RavenDB because of the possibility of a race condition. If you look carefully, you'll notice there will be a short period between the completion of `CheckForConcurrencyError()` and `_documentSession .Store()`. What would happen if a new event was appended to the stream in this period? Both events would be appended to the stream, even though `CheckForConcurrenyError()` aims to prevent this. Fortunately, you can configure RavenDB for optimistic concurrency, so it does not store an event if the stream has been updated since the event occurred.

Whichever technologies you use to build an event store, race conditions will likely be a factor you need to deal with if you're supporting optimistic concurrency. Locking the stream and checking the ID of the last committed event is likely to be the process you will rely on for this. To lock an event stream stored in RavenDB, the first step is to enable optimistic concurrency, as shown in Listing 22-23.

LISTING 22-23: Enabling Optimistic Concurrency with RavenDB

```
public static class Bootstrapper
{
    public static void Startup()
    {
        var documentStore = new DocumentStore
        {
            ConnectionStringName = "RavenDB"
        }.Initialize();

        documentStore
            .DatabaseCommands
            .EnsureDatabaseExists("EventSourcingExample");

        ObjectFactory.Initialize(config =>
        {
           // register other dependencies here, too
            config.For<IDocumentStore>().Use(documentStore);
            config.For<IDocumentSession>()
                  .HybridHttpOrThreadLocalScoped()
                  .Use(x =>
                  {
                      var store = x.GetInstance<IDocumentStore>();
                      var session = store.OpenSession();
                      session.Advanced.UseOptimisticConcurrency = true;
                      return session;
                  });
         });
    }
}
```

RavenDB provides support for optimistic concurrency. Listing 22-23 shows how it is enabled by setting `session.Advanced.UseOptimisticConcurreny` to `true`. As Listing 22-23 also shows, this can be configured in the service layer (explained more in Chapter 25: "Commands: Application Service Patterns for Processing Business Use Cases"), where the lifecycle of objects is managed, often with a container. (`ObjectFactory` is the IoC container in Listing 22-23.) Essentially, the code in Listing 22-23 ensures that each new web request, or each thread, gets its own document session with optimistic concurrency enabled. It's important for each thread or web request to get its own session because all changes made in a session can be rolled back together.

Once enabled, RavenDB enforces optimistic concurrency by aborting transactions and throwing `ConcurrencyExceptions` when an event being stored has been modified in another `IDocumentSession` since the current `IDocumentSession` was created. Listing 22-24 illustrates how this applies to `EventStore` by showing a full business use case that is carried out by the `TopUpCredit` application service.

LISTING 22-24: Handling RavenDB's Optimistic Concurrency Errors

```
public class TopUpCredit
{
    private IPayAsYouGoAccountRepository _payAsYouGoAccountRepository;
    private IDocumentSession _unitOfWork;
    private IClock _clock;

    public TopUpCredit(IPayAsYouGoAccountRepository payAsYouGoAccountRepository,
                       IDocumentSession unitOfWork, IClock clock)
    {
        _payAsYouGoAccountRepository = payAsYouGoAccountRepository;
        _unitOfWork = unitOfWork;
        _clock = clock;
    }

    public void Execute(Guid id, decimal amount)
    {
        try
        {
            var account = _payAsYouGoAccountRepository.FindBy(id);

            var credit = new Money(amount);

            account.TopUp(credit, _clock);

            _payAsYouGoAccountRepository.Save(account);

            _unitOfWork.SaveChanges();
        }
        catch (ConcurrencyException ex)
        {
            _unitOfWork.Advanced.Clear();

             // any additional error handling - retry messages, error queue, etc.

            throw ex;
```

```
            }
        }
    }
```

An instance of `IDocumentSession` is created at the start of a web request and passed into the `TopUpCredit` application service shown in Listing 22-24. `TopUpCredit` then finds the relevant account using a repository and tops up its credit, causing domain events to be added to the aggregate's list of uncommitted events. These uncommitted events are queued up for storage in RavenDB when `Save()` is called on the repository. Finally, when `SaveChanges()` is called on the `IDocumentSession` (`_unitOfWork`), RavenDB commits the changes and persists them to disk. However, if RavenDB notices that the event stream these events are being appended to has been modified by another `IDocumentSession` since the `IDocumentSession` passed into `TopUpCredit` was started, it does not commit the changes. Instead, it throws the `ConcurrencyException` you can see being handled in Listing 22-24.

> **NOTE** *You can learn more about RavenDB's optimistic concurrency support by reading the official documentation (*`https://ravendb.net/kb/16/using-optimistic-concurrency-in-real-world-scenarios`*).*

A SQL Server-Based Event Store

SQL Server is another common storage option for building event stores. There are several open source projects that provide guidance and reference implementations to help you along. Ncqrs (`https://github.com/ncqrs/ncqrs/blob/master/Framework/src/Ncqrs/Eventing/Storage/SQL/MsSqlServerEventStore.cs`) and NEventStore (`https://github.com/NEventStore/NEventStore/blob/master/src/NEventStore/Persistence/Sql/SqlPersistenceEngine.cs`) are among the most popular. Ncqrs is used as a case study in the following short section.

Choosing a Schema

When you're choosing to use SQL to store events and streams, an important consideration is schema. Those who have built event stores on top of SQL suggest being minimalistic with schema while storing event data as blobs of XML or JSON. The recommendation is a generic, domain-agnostic schema like the one shown in Listing 22-25.

LISTING 22-25: Generic Schema for a SQL Server-Based Event Store

```
Table Events:
    Id [uniqueidentifier] NOT NULL,
    TimeStamp [datetime] NOT NULL,

    Name [varchar](max) NOT NULL,
```

continues

LISTING 22-25 *(continued)*

```
    Version [varchar](max) NOT NULL,

    EventSourceId [uniqueidentifier] NOT NULL,
    Sequence [bigint],

    Data [nvarchar](max) NOT NULL

Table EventSources:
    Id [uniqueidentifier] NOT NULL,
    Type [nvarchar](255) NOT NULL,
    Version [int] NOT NULL

Table Snapshots:
    EventSourceId [uniqueidentifier] NOT NULL,
    Version [bigint] NULL,
    TimeStamp [datetime] NOT NULL,
    Type [varchar](255) NOT NULL,
    Data [varbinary](max) NOT NULL
```

One difference to the RavenDB event store you previously saw is that this schema uses `[uniqueidentifiers]`, also known as `GUIDs`, as the ID for an event stream (referred to in Listing 22-25 as an event source). In your application, you can get away with whichever type you prefer. Apart from that difference, you can see a similarity to the RavenDB-based approach, where each event is a unique record, and the concept of a stream is a logical one that relies on associating events by the ID of the stream they belong to.

Creating a Stream

Most of the implementation of the Ncqrs event store is obviated by the schema. For example, creating a new stream is simply a case of adding a new record to the `EventSources` table. This is exemplified in Listing 22-26. Do note that Ncqrs uses the term *event source*, which is what has been referred to as an *event stream* elsewhere in the chapter.

LISTING 22-26: Ncqrs Logic for Creating a Stream

```
private static void AddEventSource(Guid eventSourceId, Type eventSourceType,
    long initialVersion, SqlTransaction transaction)
{
    using (var command =
           new SqlCommand(Queries.InsertNewProviderQuery,
          transaction.Connection))
    {
        command.Transaction = transaction;
        command.Parameters.AddWithValue("Id", eventSourceId);
        command.Parameters.AddWithValue("Type", eventSourceType.ToString());
        command.Parameters.AddWithValue("Version", initialVersion);
        command.ExecuteNonQuery();
    }
```

```
}

internal static class Queries
{
    ...

    public const String InsertNewProviderQuery =
    "INSERT INTO [EventSources](Id, Type, Version) VALUES (@Id, @Type, @Version)";

    ...
}
```

Saving Events

The schema also obviates saving events. Listing 22-27 demonstrates this by showing how each event is added as a new row in the Events table, with a reference to the event stream it belongs to.

LISTING 22-27: Ncqrs Logic for Storing Events

```
private void SaveEvents(IEnumerable<UncommittedEvent> uncommittedEvents,
    SqlTransaction transaction)
{
    foreach (var sourcedEvent in uncommittedEvents)
    {
        SaveEvent(sourcedEvent, transaction);
    }
}

private void SaveEvent(UncommittedEvent uncommittedEvent, SqlTransaction transaction)
{
    string eventName;
    var document = _formatter.Serialize(uncommittedEvent.Payload, out eventName);
    var storedEvent = new StoredEvent<JObject>(
        uncommittedEvent.EventIdentifier,
        uncommittedEvent.EventTimeStamp, eventName,
        uncommittedEvent.EventVersion,
        uncommittedEvent.EventSourceId,
        uncommittedEvent.EventSequence, document
    );
    var raw = _translator.TranslateToRaw(storedEvent);

    using (var command = new SqlCommand(
        Queries.InsertNewEventQuery, transaction.Connection))
    {
        command.Transaction = transaction;
        command.Parameters.AddWithValue("EventId", raw.EventIdentifier);
        command.Parameters.AddWithValue("TimeStamp", raw.EventTimeStamp);
        command.Parameters.AddWithValue("EventSourceId", raw.EventSourceId);
        command.Parameters.AddWithValue("Name", raw.EventName);
        command.Parameters.AddWithValue("Version", raw.EventVersion.ToString());
        command.Parameters.AddWithValue("Sequence", raw.EventSequence);
```

continues

LISTING 22-27 *(continued)*

```
            command.Parameters.AddWithValue("Data", raw.Data);
            command.ExecuteNonQuery();
        }
    }

    internal static class Queries
        {
            ...

            public const String InsertNewEventQuery = "INSERT INTO [Events]([Id],
    [EventSourceId], [Name], [Version], [Data], [Sequence], [TimeStamp]) VALUES
    (@EventId, @EventSourceId, @Name, @Version, @Data, @Sequence, @TimeStamp)";

            ...
    }
```

Loading Events from a Stream

Listing 22-28 uses a sample from Ncqrs that reads events from a stream. This is akin to `IEventStore.GetStream()` shown earlier in the chapter, and with similar logic. All events that reference the passed-in stream name (`id`) are returned if they fall within the range of the supplied `minVersion` and `maxVersion`.

LISTING 22-28: Ncqrs Retrieving Events from a SQL Server-Based Event Store

```
public CommittedEventStream ReadFrom(Guid id, long minVersion, long maxVersion)
{
    var events = new List<CommittedEvent>();

    using (var connection = new SqlConnection(_connectionString))
    {
        using (var command =
            new SqlCommand(Queries.SelectAllEventsQuery, connection))
        {
            command.Parameters.AddWithValue("EventSourceId", id);
            command.Parameters.AddWithValue("EventSourceMinVersion", minVersion);
            command.Parameters.AddWithValue("EventSourceMaxVersion", maxVersion);
            connection.Open();

            using (SqlDataReader reader = command.ExecuteReader())
            {
                while (reader.Read())
                {
                    var evnt = ReadEventFromDbReader(reader);
                    events.Add(evnt);
                }
            }
        }
    }

    return new CommittedEventStream(id, events);
```

```
}

internal static class Queries
{
    ...

    public const String SelectAllEventsQuery =
    "SELECT [Id], [EventSourceId],[Name], [Version], [TimeStamp], [Data], Sequence]
FROM [Events] WHERE [EventSourceId] = @EventSourceId
AND [Sequence] >= @EventSourceMinVersion AND
    [Sequence] <= @EventSourceMaxVersion
ORDER BY [Sequence]";

     ...
}
```

You can see the full implementation of `ReadEventFromDbReader()` referenced in Listing 22-28 on GitHub (`https://github.com/ncqrs/ncqrs/blob/master/Framework/src/Ncqrs/Eventing/Storage/SQL/MsSqlServerEventStore.cs`).

Snapshots

A dedicated table is used to store snapshots in Ncqrs. Using the ID of event stream (also known as an event source), Listing 22-29 shows how snapshots are saved, and Listing 22-30 shows how they are loaded.

LISTING 22-29: Ncqrs Logic for Saving Snapshots

```
public void SaveSnapshot(Snapshot snapshot)
{
    using (var connection = new SqlConnection(_connectionString))
    {
        connection.Open();
        using (SqlTransaction transaction = connection.BeginTransaction())
        {
            try
            {
                using (var dataStream = new MemoryStream())
                {
                    var formatter = new BinaryFormatter();
                    formatter.Serialize(dataStream, snapshot.Payload);
                    byte[] data = dataStream.ToArray();

                    using (var command =
                        new SqlCommand(Queries.InsertSnapshot,
                            transaction.Connection))
                    {
                        command.Transaction = transaction;
                        command.Parameters.AddWithValue(
                            "EventSourceId", snapshot.EventSourceId
                        );
                        command.Parameters.AddWithValue(
                            "Version", snapshot.Version
```

continues

LISTING 22-29 *(continued)*

```
                        );
                        command.Parameters.AddWithValue(
                            "Type", snapshot.GetType().AssemblyQualifiedName
                        );
                        command.Parameters.AddWithValue("Data", data);
                        command.ExecuteNonQuery();
                    }
                }

                transaction.Commit();
            }
            catch
            {
                transaction.Rollback();
                throw;
            }
        }
    }
}

internal static class Queries
{
    ...

    public const String InsertSnapshot =
"DELETE FROM [Snapshots] WHERE [EventSourceId]=@EventSourceId;
INSERT INTO [Snapshots]([EventSourceId], [Timestamp], [Version], [Type], [Data])
VALUES (@EventSourceId, GETDATE(), @Version, @Type, @Data)";

    ...
}
```

LISTING 22-30: Ncqrs Logic for Loading Snapshots

```
public Snapshot GetSnapshot(Guid eventSourceId, long maxVersion)
{
    using (var connection = new SqlConnection(_connectionString))
    {
        connection.Open();

        using (var command =
            new SqlCommand(Queries.SelectLatestSnapshot, connection))
        {
            command.Parameters.AddWithValue("@EventSourceId", eventSourceId);

            using (var reader = command.ExecuteReader())
            {
                if (reader.Read())
                {
                    var snapshotData = (byte[])reader["Data"];

                    using (var buffer = new MemoryStream(snapshotData))
```

```
                    {
                        var formatter = new BinaryFormatter();
                        var payload = formatter.Deserialize(buffer);
                        var theSnapshot = new Snapshot(
                            eventSourceId, (long)reader["Version"], payload
                        );

                        return theSnapshot.Version > maxVersion
                                    ? null
                                    : theSnapshot;
                    }
                }

                return null;
            }
        }
    }
}

internal static class Queries
{
    ...

    public const String SelectLatestSnapshot =
"SELECT TOP 1 * FROM [Snapshots] WHERE [EventSourceId]=@EventSourceId
ORDER BY Version DESC";

    ...
}
```

Is Building Your Own Event Store a Good Idea?

During the early days of event sourcing, there were no commercial tools. Instead, developers had to build event-sourcing capabilities on top of existing technologies like SQL databases or document databases, as shown in this section. This means that it's definitely achievable. But if you move on to more advanced scenarios like projections, complex temporal queries, and enhancing scalability, you may find you are spending a lot of time away from adding business value. This is why you may want to consider a purpose-built technology like Greg Young's Event Store (also known as the Event Store) that provides many advanced features out of the box.

USING THE PURPOSE-BUILT EVENT STORE

Choosing to use an existing event store reduces the amount of work you have to do up front to get started with event sourcing. Also, by choosing an event store like Greg Young's Event Store, you get many additional features out of the box, including advanced projections and multinode clustering for highly scalable environments. To demonstrate using event store, the examples in this section remain oriented around the pay-as-you-go service offered by a fictitious cell phone network operator. You see how to build an alternative `IEventStore` implementation using the Event Store C# client library, as well as running queries and projections in the admin web UI. To work through these examples, you need Event Store installed on your machine.

Installing Greg Young's Event Store

To run projections, Event Store version 3.0.0 rc2 is required (`http://download.geteventstore.com/binaries/eventstore-net-v3.0.0rc2.zip`). Once you've downloaded the zip file, you need to extract it to a folder of your choice. From within the directory you extract to, you can start Event Store with the following PowerShell command. You need to start PowerShell with Administrator privileges.

```
.\EventStore.SingleNode.exe --db .\ESData --run-projections=all
```

A few tweaks are necessary to enable projections. By navigating to the admin web UI (`http://localhost:2113/projections`), you can enable projections by accessing the Projections tab and starting the following projections: `$by_category` and `$stream_by_category`. (Click on the link and then click the Start button.)

> **WARNING** *The steps to install Event Store are almost identical to those shown in Chapter 13: "Integrating via HTTP with RPC and REST." However, please note that this is an unstable version (v3 rc2) that provides the projection capabilities required in this chapter's examples. It is not production ready, although a newer version that is production ready may be available when you are reading this. If you are using a newer version, it's possible that the projection's API may have changed.*

Using the C# Client Library

You can take advantage of Event Store in your applications by writing a persistence layer that uses the official Event Store C# client library. Alternatively, Event Store has a Hypertext Transport Protocol (HTTP) API that you will see in Chapter 26: "Queries: Domain Reporting." In this example, you see a new implementation of `IEventStore` that can be switched with the RavenDB version you created earlier in the chapter. This implementation relies on the following NuGet packages:

```
Install-Package EventStore.Client -version 2.0.2.0
Install-Package Newtonsoft.Json -version 6.0.3
```

`EventStore.Client` is the official client library, created by the Event Store team. It uses TCP to communicate with Event Store. `Newtonsoft.Json` is necessary because events are stored as JSON; therefore, they need to be serialized to and from C# classes. Listing 22-31 shows the `GetEventStore` implementation of `IEventStore` that provides concrete examples of these details.

LISTING 22-31: An IEventStore Implementation That Uses Event Store as Storage

```
public class GetEventStore : IEventStore
{
    private IEventStoreConnection esConn;

    private const string EventClrTypeHeader = "EventClrTypeName";

    public GetEventStore(IEventStoreConnection esConn)
```

```
{
    this.esConn = esConn;
}

public void CreateNewStream(string streamName,
    IEnumerable<DomainEvent> domainEvents)
{
    // automatically creates a stream when events are added to it
    AppendEventsToStream(streamName, domainEvents, null);
}

public void AppendEventsToStream(string streamName,
    IEnumerable<DomainEvent> domainEvents, int? expectedVersion)
{
    var commitId = Guid.NewGuid();

    var eventsInStorageFormat = domainEvents.Select(
        e => MapToEventStoreStorageFormat(e, commitId, e.Id)
    );

    esConn.AppendToStream(
        StreamName(streamName),
        expectedVersion ?? ExpectedVersion.Any,
        eventsInStorageFormat
    );
}

private EventData MapToEventStoreStorageFormat(object evnt,
    Guid commitId, Guid eventId)
{
    var headers = new Dictionary<string, object>
    {
        // each event in this operation will be associated with the same commit
        {"CommitId", commitId},

        // store type of class so event can be rebuilt when the event is loaded
        {EventClrTypeHeader, evnt.GetType().AssemblyQualifiedName}
    };

    // events and headers are stored at binary-encoded JSON
    var data = Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(evnt));
    var metadata = Encoding.UTF8.GetBytes(
        JsonConvert.SerializeObject(headers)
    );

    // enahnced support in the admin web UI if Event Store knows events are JSON
    var isJson = true;

    return new EventData(eventId, evnt.GetType().Name, isJson, data, metadata);
}

public IEnumerable<DomainEvent> GetStream(string streamName,
    int fromVersion, int toVersion)
{
    // Event Store wants number of events to retrieve
```

continues

LISTING 22-31 *(continued)*

```
            // not highest version
            var amount = (toVersion - fromVersion) + 1;
            var events = esConn.ReadStreamEventsForward(
                StreamName(streamName), fromVersion, amount, false
            );

            // map events back from JSON to DomainEvent. Header indicates the type
            return events.Events.Select(e => (DomainEvent)RebuildEvent(e));
        }

        private object RebuildEvent(ResolvedEvent eventStoreEvent)
        {
            var metadata = eventStoreEvent.OriginalEvent.Metadata;
            var data = eventStoreEvent.OriginalEvent.Data;
            var typeOfDomainEvent =
                JObject.Parse(Encoding.UTF8.GetString(metadata))
                       .Property(EventClrTypeHeader).Value;

            var rebuiltEvent = JsonConvert.DeserializeObject(
                Encoding.UTF8.GetString(data),
                Type.GetType((string)typeOfDomainEvent)
            );

            return rebuiltEvent;
        }

        // snapshots in Event Store are just events in dedicated snapshot streams
        // explained: http://stackoverflow.com/questions/16359330/is-snapshot-
    supported-from-greg-young-eventstore
        public void AddSnapshot<T>(string streamName, T snapshot)
        {
            var stream = SnapshotStreamNameFor(streamName);
            var snapshotAsEvent = MapToEventStoreStorageFormat(
                snapshot, Guid.NewGuid(), Guid.NewGuid()
            );

            esConn.AppendToStream(stream, ExpectedVersion.Any, snapshotAsEvent);
        }

        public T GetLatestSnapshot<T>(string streamName) where T : class
        {
            var stream = SnapshotStreamNameFor(streamName);
            var amountToFetch = 1; // just the latest one
            var ev = esConn.ReadStreamEventsBackward(
                stream, StreamPosition.End, amountToFetch, false
            );

            if (ev.Events.Any())
                return (T)RebuildEvent(ev.Events.Single());
            else
                return null;
```

```
        }

        private string SnapshotStreamNameFor(string streamName)
        {
            // snapshots are just events in separate streams
            return StreamName(streamName) + "-snapshots";
        }

        private string StreamName(string streamName)
        {
            // Event Store projections require only a single hypen ("-")
            // see: https://groups.google.com/forum/#!msg/event-
    store/D477bKLcdI8/62iFGhHdMMIJ
            var sp = streamName.Split(new []{ '-' }, 2);
            // remove all hyphens except the first
            return sp[0] + "-" + sp[1].Replace("-", "");
        }
    }
```

It's worth spending a few moments looking at Listing 22-31. Notice how using a purpose-built event store requires less work than the RavenDB or SQL Server-based approaches shown earlier in the chapter. Two noticeable examples of this are querying for events and optimistic concurrency support. When querying for events in `GetEventStream()`, you simply pass in the name of the stream with a starting version and number of events to retrieve. In the RavenDB solution, there was a moderately complex query with a number of clauses. This is because Event Store natively supports the concept of a stream, rather than logically simulating one.

Native stream support is also the reason optimistic concurrency is less work to implement. As you can see with Event Store's `IEventStoreConnection.AppendToStream()`, you only have to pass in the expected version number of the last stored event; Event Store takes care of all the optimistic concurrency checking and management for you.

One advantage that RavenDB does have is that it deals with the serialization to JSON and back. In Listing 22-31, you can see that the type of the event—the name of the C# class—is stored as a header in `MapToEventStoreStorageFormat()`. This header is then used on the way back out when events are reconstructed from JSON, as shown in `RebuildEvent()`. In both cases, there is a need to coordinate the serialization and deserialization logic yourself.

Snapshot support is implemented by creating a new event stream for each aggregate's snapshots. Using a purpose-built Event Store shines for this use case as well. You can see in `GetLatestSnapshot()` that you can make a single query to retrieve the latest snapshot by really leaning on `IEventStore.ReadStreamEventsBackwards()`, passing in 1 as the number of events to return.

One important detail that isn't shown is creating the initial connection to Event Store. Listing 22-32 shows how you can use the client library's `EventStoreConnection.Create()` to do this.

LISTING 22-32: Creating a Connection to Event Store with the Client Library

```
IPEndPoint endpoint = new IPEndPoint(IPAddress.Loopback, 1113);
IEventStoreConnection con = EventStoreConnection.Create(endpoint)
con.Connect();
```

Listing 22-32 shows how to create a TCP connection to an Event Store instance running locally on the default port of 1113. However, you should change the port and address to those that you configured Event Store to run on.

> **WARNING** *The examples in this chapter use the blocking methods provided by Event Store's C# client. For better performance and scalability, you may prefer to look at the nonblocking, asynchronous API calls that are suffixed with* `Async` *(for example:* `AppendEventsToStreamAsync()`*).*

Running Temporal Queries

Greg Young and the Event Store team have decided that JavaScript is the best tool for making queries and creating projections. Shortly, you will see examples of creating temporal queries in Event Store's user-friendly web UI. First, though, you need to import some data into your locally running version of Event Store. This chapter's sample code contains a class called `ImportTestData` in the `EventStoreDemo` project. This has a `TestMethod` that you can run from within Visual Studio, and it populates Event Store with a number of events for a collection of `PayAsYouGoAccount` aggregates. Once you have run the test, you can see the newly created streams and their events by navigating to the Stream tab in the web UI (`http://localhost:2113/web/streams.htm`). You should see activity similar to Figure 22-4, indicating that the streams were created.

Recently Changed Streams

- PayAsYouGoAccount-6f6ac515-9e3d-414f-b1f3-950e39256646
- PayAsYouGoAccount-f727da8b-2a0c-4a8a-bae9-545285959ff5
- PayAsYouGoAccount-7c296380-897b-43c2-b712-6b40041aa65b
- PayAsYouGoAccount-21e6936b-afeb-4dd0-a9df-b695e8bf51d4
- PayAsYouGoAccount-18722870-e0b9-493f-932f-5510e70f1721

FIGURE 22-4: Event Store's stream tab indicating the test data was successfully inserted.

Querying a Single Stream

Despite sounding a bit fancy, temporal queries don't have to be big and complicated. A short query that might be useful is showing a customer how many minutes he used on a given day. It could be the current day, or it could be a date in the past. This helps customers understand if they are using their phone too much. Listing 22-33 shows the JavaScript needed for this query.

LISTING 22-33: An Event Store Query to Get Total Minutes Used on a Given Date for a Single Customer

```
fromStream('PayAsYouGoAccount-5b3415c8d58a4dcf955eed9978bcd8b1')
.when({
     // initialize the state
    $init : function(s,e) {return {minutes : 0}},

    "PhoneCallCharged" : function(s,e) {
        var dateOfCall = e.data.PhoneCall.StartTime;
        var june4th = "2014-06-04";
        if (dateOfCall.substring(0, 10) == june4th) {
```

```
                s.minutes += e.data.PhoneCall.Minutes.Number;
            }
        }
    });
```

Queries in Event Store begin when you specify which events you want to query. In Listing 22-33, `fromStream()` is specifying that the query should apply to all events in a single stream representing a single `PayAsYouGoAccount`. You can get the ID of an event stream from the Streams tab in the web UI. Each event in the stream, starting with the oldest, is then passed into `when()`. Inside `when()` is a pattern match that determines which events are handled and how. In Listing 22-33, you can see that only events of type `PhoneCallCharged` are handled.

The result of an Event Store query is a JavaScript object known as the state of the query. This object is passed into `when()` with each event and is updated repeatedly as per your query logic. You can see that the state in Listing 22-33 is initialized inside the `$init` handler as a JavaScript object with a single property called `minutes`. It's up to you to decide the structure of this state in each of your queries. It just has to be valid JavaScript. Listing 22-33 uses the state by augmenting its `minutes` property each time a `PhoneCallCharged` event, which represents a phone call on June 4, is handled.

After running the query in Listing 22-33 on the Query tab of Event Store's web UI (`http://localhost:2113/web/query.htm`), you will see output similar to Figure 22-5.



FIGURE 22-5: Result of running a query in the Event Store web UI.

Figure 22-5 shows how Event Store displays the results of a query as its final state in the right State pane. As mentioned, you determine the structure of the state yourself and can have whatever properties are necessary. For example, you might want to enhance the query in Listing 22-33 to keep a total for the number of minutes used on multiple dates, where each date is a separate property on the state. This query is shown in Listing 22-34.

LISTING 22-34: A More Complex Query That Builds Up Multivalued State

```
fromStream('PayAsYouGoAccount-86e057b961624c6f92c9ab3c56e6e232')
.when({
    // initialize the state
    $init : function(s,e) {return { june3rd: 0, june4th: 0, june5th: 0}},

   "PhoneCallCharged" : function(s,e) {
        var dateOfCall = e.data.PhoneCall.StartTime;
        var june3rd = "2014-06-03";
        var june4th = "2014-06-04";
        var june5th = "2014-06-05";
        if (dateOfCall.substring(0, 10) == june3rd) {
            s.june3rd += e.data.PhoneCall.Minutes.Number;
        }
        if (dateOfCall.substring(0, 10) == june4th) {
            s.june4th += e.data.PhoneCall.Minutes.Number;
        }
        if (dateOfCall.substring(0, 10) == june5th) {
            s.june5th += e.data.PhoneCall.Minutes.Number;
        }
    }

    // handle other types of event and update the state accordingly
 });
```

In Listing 22-34, the state contains three properties, corresponding to each date. Each of these properties is increased by the number of minutes used up during phone calls on the relevant date.

Querying Multiple Streams

Event Store is not limited to querying a single stream. For many use cases, it's important to combine data from multiple streams, akin to joins in SQL. Listing 22-35 shows how the combined total minutes used on each date can be calculated for all pay-as-you-go accounts.

LISTING 22-35: A Query Combining Events from Multiple Streams

```
fromCategory('PayAsYouGoAccount')
.when({
    // initialize the state
    $init : function(s,e) {return { june3rd: 0, june4th: 0, june5th: 0}},

   "PhoneCallCharged" : function(s,e) {
        var dateOfCall = e.data.PhoneCall.StartTime;
        var june3rd = "2014-06-03";
```

```
        var june4th = "2014-06-04";
        var june5th = "2014-06-05";
        if (dateOfCall.substring(0, 10) == june3rd) {
            s.june3rd += e.data.PhoneCall.Minutes.Number;
        }
        if (dateOfCall.substring(0, 10) == june4th) {
            s.june4th += e.data.PhoneCall.Minutes.Number;
        }
        if (dateOfCall.substring(0, 10) == june5th) {
            s.june5th += e.data.PhoneCall.Minutes.Number;
        }
    }

    // handle other types of event and update the state accordingly
});
```

Event Store's `fromCategory()` works by combining all streams whose name begins with the passed-in name, suffixed with a hyphen. For example, the query in Listing 22-35 matches all events in all streams whose name begins with `PayAsYouGoAccount-`. However, a small gotcha is that Event Store looks for the last hyphen. This is the reason for the private `StreamName()` method in the `GetEventStore` class shown in Listing 22-35, which removes all but the first hyphen.

Creating Projections

Instead of just calculating state, sometimes you want the ability to take subsets of events and create an entirely new stream from them. This can be crucial in a large system with millions or billions of events. Take the cell phone network operator, for example; if there a millions of customers each with lots of account activity, there could easily be billions of events. It would be inefficient to run queries across all of them. Projections solve this problem by letting you select only events that you are interested in by putting them in a new stream and then running your query against the newly created stream. Listing 22-36 shows a projection that groups all the top ups made by all customers into a new stream called `AllTopUps`.

LISTING 22-36: A Projection Selectively Copying Events from Many Streams into a Single One

```
fromCategory('PayAsYouGoAccount')
.when({
    "CreditAdded": function(s, event) {
        linkTo('AllTopUps', event);
    }
});
```

Running the projection in Listing 22-36 causes all `CreditAdded` events that exist in any `PayAsYouGoAccount` stream to be added to a new stream called `AllTopUps`. This means it is possible to run queries against all the `CreditAdded` events without having to load all the other events in the streams like `PhoneCallCharged`. Event Store's key tool for providing this behavior is `linkTo()`, which adds a link in the stream whose name matches the first argument (`AllTopUps` in this example)

to the passed-in event. Projections are run by navigating to the Projections tab, choosing New Projection, and filling out the form, as per Figure 22-6.

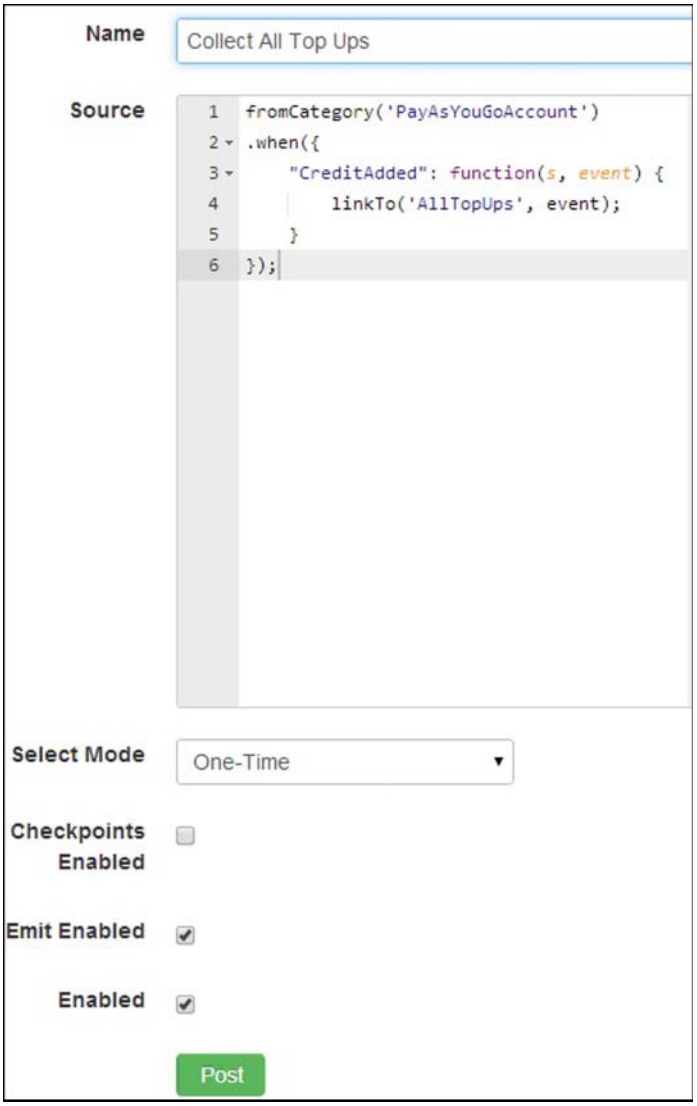


FIGURE 22-6: Creating a projection in the web UI.

Figure 22-6 shows a one-time projection being created. However, the Select Mode field can instead be set to Continuous. This is useful when you want the projection to be applied to new events as they are stored in near-real time. CQRS is a common example of where you might want to do this, as you will see shortly.

You can learn a lot more about the projection capabilities of Event Store on the official blog (`http://geteventstore.com/blog/`). There is a multipart series devoted to projections (`http://geteventstore.com/blog/20130309/projections-8-internal-indexing/index.html`).

> **NOTE** *Chapter 26: "Queries: Domain Reporting," contains examples of creating reports from Event Store projections.*

CQRS WITH EVENT SOURCING

As a way to improve performance and scalability, you may want to create materialized views of your events. Doing so prevents the need for many queries to continually be run against a single event stream. Instead, precomputed values will be available for each specific need. A scenario that demonstrates this problem occurs when a number of web pages all run different queries against a single event stream, as shown in Figure 22-7.

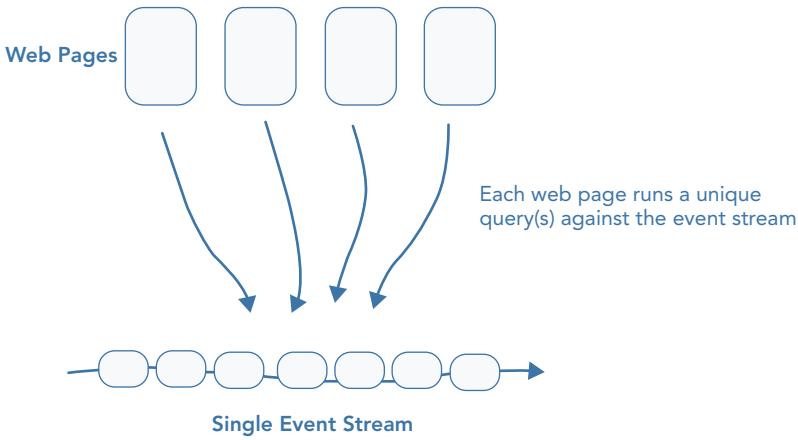


FIGURE 22-7: One event stream used to support many different queries and use cases.

As load increases on any of the web pages, the single event stream is put under increasing amounts of stress. This means that less important pages may substantially degrade the performance of important pages. A solution to this problem is to create a new materialized view of the data to support each page, as shown in Figure 22-8.

The solution in Figure 22-8 is CQRS; where commands (writes) go via the domain model to the event stream, and queries (reads) are run against the materialized views (also known as view caches). You can see that the problem of certain use cases impeding others is heavily mitigated because each has its own view cache for querying. As a direct benefit, you can see contention on the single event stream has been lessened; now fewer (if any) taxing ad hoc queries are run against it.

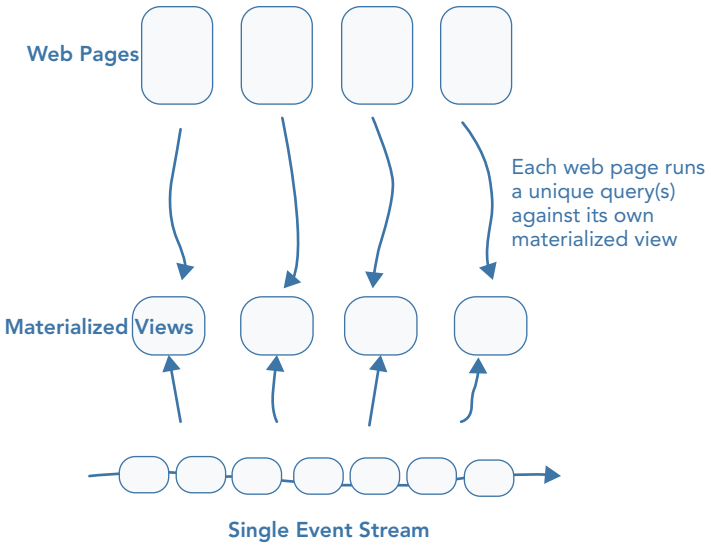


FIGURE 22-8: Creating materialized/denormalized views of the event stream to support each use case.

This section is just a quick glimpse of CQRS and how it synergizes with event sourcing. In Chapter 24: "CQRS: An Architecture of a Bounded Context," you will see a more in-depth discussion of CQRS, including how you can apply it to a variety of scenarios that don't involve event sourcing.

Using Projections to Create View Caches

To implement CQRS with event sourcing, you need some way to create denormalized views from your event streams. The answer is to use projections that were previously introduced. The materialized views in Figure 22-8 are examples of where projections can be used for CQRS.

As you've seen previously in this chapter, if you're using a purpose-built tool like Event Store, you already have the functionality to use projections and implement CQRS. However, when building an event store of your own, you need to create the functionality yourself, which might not be a trivial task. Fortunately, RavenDB provides the capability to create projections (`http://ayende.com/blog/4530/raven-event-sourcing`).

CQRS and Event Sourcing Synergy

Although CQRS and event sourcing are standalone concepts that you can use independently of the other, a strong following of developers find a significant advantage from combining them. Figure 22-6 illustrated the ease at which you can create view caches as projections from event streams. But there are other synergistic benefits, too.

Event Streams as Queues

One of the examples in Chapter 13: "Integrating via HTTP with RPC and REST," showed how Event Store exposes event streams as a hypermedia Atom feed. This was an alternative to using a message bus. Again, the benefit of this is that your event store saves your having to use a queue; likewise, it saves your having to use another technology for view caches and another for the denormalization process.

No Two-Phase Commits

Because an Event Store is your main source of data, the tool you use for projections, and your queueing technology, there are no two-phase commits (or distributed transactions) to worry about. Once an event is in a stream, it is successfully in the queue and has successfully been saved. In contrast, if you were storing events in a database, publishing them with a message bus, and updating a view cache in another database all inside the same transaction, you may have to carefully plan how a failure in either of these actions would be handled, such as rolling back the others. Greg Young further details this on his blog (`http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing/`).

RECAPPING THE BENEFITS OF EVENT SOURCING

If this is your first encounter with event sourcing, you may still be processing some of the details and building a mental model. That's completely understandable and something that everyone goes through. This section recaps some of the benefits that event sourcing can bring to your projects. Hopefully, at least if you don't remember all the details, you will still have an understanding of when you might want to consider using event sourcing.

Competitive Business Advantage

This chapter began by discussing the fundamental rationale for using event sourcing: allowing the business to gain a competitive advantage by being able to analyze not just its data, but the entire history of behavior that produced the current state of its data. If two companies are competing in a market and one is able to lean on event sourcing to perform powerful analysis, it may be able to use the knowledge to expedite development of its products or services.

Expressive Behavior-Focused Aggregates

Communicating with domain experts is a significant aspect of applying DDD and realizing its benefits. To support this, it's important to have a domain model that expresses the conceptual model using the ubiquitous language (UL). As you saw earlier in this chapter, event-sourced aggregates are almost declarative, with each overload of `When()` reading like a sentence:

```
When {Domain Event} {Apply Business Rules}
```

This makes the domain model even more useful in knowledge-crunching sessions. Even in general, it can speak much more clearly to other developers who might be new to the domain or codebase.

Simplified Persistence

Persisting aggregates with Object Relational Mappers (ORMs) has traditionally been a topic of controversy and a source of pain for many software teams due to the impedance mismatch. As you saw in this chapter, persisting and rehydrating events from an event stream do not suffer from this problem because there is no impedance mismatch. This means that the ORM technology does not constrain the database model or require complex mappings. In fact, with event sourcing, there is a real possibility that you can change the persistence technology with polymorphism. You saw in this chapter that multiple implementations of `IEventStore` can be switched as necessary.

It's worth noting that the `IEventStore` abstraction does have small leaks, so there's no guarantee you can easily switch out an event storage provider with another. One example is the stream names; Event Store's `fromCategory()` is based on the last hyphen in the name. You did see, though, that there was an easy workaround for this in `GetEventStore`.

Superior Debugging

Earlier in the chapter you saw how event sourcing can make systems easier to debug. If you think about it, you have the entire history of behavior stored in your event store. So you can easily rerun any sequence of events to work out which event caused an incorrect state change or some other type of bug. You don't have to guess about the sequence of events that may have led to a problem occurring.

WEIGHING THE COSTS OF EVENT SOURCING

It's important to be realistic about the negative aspects of event sourcing, too. Even event store vendors are advising people to think carefully about where to use it. You need to invest additional time, and the payback may not always be worth it—especially because there are a number of different considerations that may be demanding of your time.

Versioning

As you learn more about the domain and you continue to enhance the product(s) you are building, new concepts and information need to be added to the domain model. As a result, you may need to rename events, move data between events, or perform some other change that alters the format of your events. This presents a big problem, because you will already have an event stream containing events in the old format(s). There are solutions to this problem, and they don't always require much effort. However, versioning can definitely be a problem if you don't pay it enough respect.

New Concepts to Learn and Skills to Hone

Projections, temporal queries, snapshots—many concepts may be new to a team that is choosing to use event sourcing for the first time. As with many things, you need to combine theory with practical experimenting before you become proficient, so you should be prepared for your team's

rate of development to be slower in the short term. You may also consider allocating time for learning and experimentation. These costs may apply to some extent anytime someone new joins the team who is unfamiliar with event sourcing.

New Technologies to Learn and Master

You can reuse existing database technologies like SQL Server for event sourcing, but it's likely that many will choose to use a purpose-built event store. The cost of this is the extra time it takes to learn the technology—not just using it, but running it on live servers and monitoring its behavior and resource usage. It's hard to quantify just how much, but putting an event store into production for the first time almost certainly requires an investment in people hours.

Greater Data Storage Requirements

It's obvious that storing the entire history of activity that led to the state requires more information to be stored on disk than just storing state. Fortunately, storage is incredibly cheap nowadays, and this is unlikely to be a concern for many. However, it is something that you should keep in mind and monitor accordingly.

ADDITIONAL LEARNING RESOURCES

- Event Store provides a technology-agnostic discussion of event sourcing—`http://docs.geteventstore.com/`.
- Martin Fowler has an article on event sourcing—`http://martinfowler.com/eaaDev/EventSourcing.html`.
- Jérémie Chassaing has a number of event sourcing articles on his blog—`http://thinkbeforecoding.com/tag/Event%20Sourcing`.
- Lev Gorodinski shows how to apply DDD and event sourcing with F#—`http://gorodinski.com/blog/2013/02/17/domain-driven-design-with-fsharp-and-eventstore/`.
- Event sourcing with Akka (Scala or Java)—`http://doc.akka.io/docs/akka/snapshot/scala/persistence.html`.
- The DDD/CQRS Google group—`https://groups.google.com/forum/#!forum/dddcqrs`.

THE SALIENT POINTS

- Event sourcing replaces traditional snapshot-only storage with a full history of events that produce the current state.
- You can rewind the state to any previous point in history with event sourcing.
- Storing history allows powerful querying capabilities that revolve around time—temporal queries.

- Temporal queries can be a game-changing business capability, allowing greater analytical insights.
- Domain models need to contain event-oriented aggregates when using event sourcing.
- Event-sourced aggregates allow for the declarative expression of business rules and looser coupling to the persistence technology.
- You can implement an event store using existing storage options like document databases and SQL Server.
- Event Store is a purpose-built event store that natively supports the concept of streams and provides advanced functionality, like projections.
- CQRS and event sourcing are a synergistic combination that uses projections to create view caches.
- Event sourcing can often be a lot of effort without a worthwhile return on investment, so don't use it without careful consideration

PART IV
Design Patterns for Effective Applications

- **CHAPTER 23:** Architecting Application User Interfaces
- **CHAPTER 24:** CQRS: An Architecture of a Bounded Context
- **CHAPTER 25:** Commands: Application Service Patterns for Processing Business Use Cases
- **CHAPTER 26:** Queries: Domain Reporting



23

Architecting Application User Interfaces

WHAT'S IN THIS CHAPTER?

- An introduction to the UI challenges involved in loosely coupled, distributed, and nondistributed Domain-Driven Design (DDD) systems
- An example of building a UI that pulls in content from multiple bounded contexts that run as a single application
- An example of building a UI that pulls in content from distributed bounded contexts

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 23 download and individually named according to the names throughout the chapter.

Customers mostly care about the user interface of your application. If it looks compelling and allows them to achieve what they want, such as finding the perfect holiday, they will be happy to spend lots of money. But getting the UI right is more than just about letting designers come up with eye candy. There are significant engineering challenges tied to the performance, scalability, and loose coupling of your behind-the-scenes bounded contexts.

One of the fundamental engineering challenges of a UI is pulling together all the data. For an e-commerce application, you may want to show catalog items, prices, shipping options, special offers, and other types of information on a single page. You know from Part II of this

book, "Strategic Patterns: Communicating Between Bounded Contexts" that with event-driven applications, this variety of information is stored in multiple, eventually consistent bounded contexts. You also know that these types of systems are share-nothing; in other words, the web application cannot simply query the database of another bounded context because that increases coupling. To solve this problem, you have choices, each with a variety of trade-offs. For instance, you can combine the data on the server or via AJAX calls directly in the web page. Your bounded contexts have the option of returning plain data, usually XML or JSON, or they may return HTML that can be directly dumped onto the page. This chapter has examples of each of these scenarios, along with guidance about when each pattern is relevant and what trade-offs are involved.

Before commencing with the examples, though, this chapter begins by taking you through some of the main UI considerations from high-level decisions—such as which team should own it—to low-level decisions—like which programming language to use. After completing this chapter, you will learn about how the application tier deals with inputs coming from the UI and provides all the infrastructural glue to coordinate actions with bounded contexts.

DESIGN CONSIDERATIONS

It can be quite surprising to see the variety of options and trade-offs that are involved in designing UIs that bring in content from multiple bounded contexts. Some of the options may affect how you design your back-end application programming interfaces (APIs), whereas others may even impact your choice of programming language(s). In fact, your UI could even affect which data needs to be stored by some bounded contexts.

Owned UIs versus Composed UIs

Your first decision when designing a UI is to decide who logically owns it. For instance, it could live within a single bounded context (more specifically a single business component) and be owned by the team responsible for that bounded context. Alternatively, the view could pull in data from multiple bounded contexts but not be owned by any of them.

Autonomous

A UI for an autonomous application belongs to a single business component. It does not need to pull in content from another bounded context. However, this means the business component needs to store locally all the information that should be presented on the UI. To make this possible, the business component has to subscribe to events from other bounded contexts that contain the data it needs and store the data locally. This was discussed previously in Part II and is illustrated in Figure 23-1.

Figure 23-1 illustrates a content-enhancement application that the Catalog bounded context owns. This allows people working in the catalog team to update and override the content for specific products. All the content for products is stored in this business component, so it is fully available. However, when they're updating content, the business staff members want to know how often a product is sold so they can understand how much effort they should put into the quality of content.

This information is retrieved by subscribing to the Sales bounded context's Sale Completed event and stored in the Content Enhancement database, ready to be presented in the autonomous web application's UI.

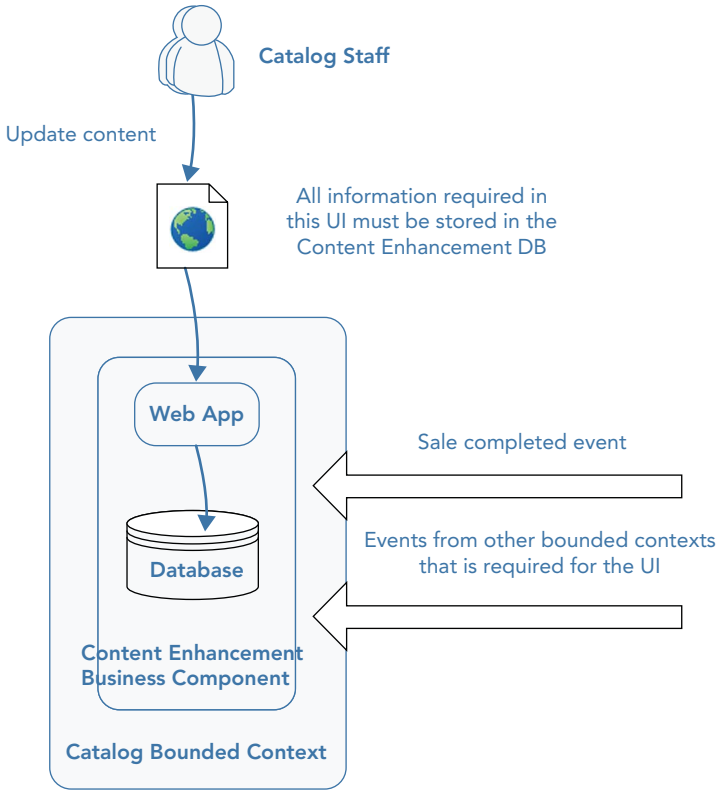


FIGURE 23-1: UI for autonomous applications.

Eventual consistency can be an important consideration with UIs in autonomous applications because the information shown may not be fully up to date. For the example in Figure 23-1, it is fine for the number of sales to be minutes, hours, or even days out of date because the catalog staff just need an idea of an item's popularity. But if data freshness was a big concern, an authoritative application may be a better choice.

Authoritative

When you want the latest snapshot of information from multiple bounded contexts in a single UI, the application has to request each piece of information directly from the authoritative bounded context. You can see this in Figure 23-2.

Figure 23-2 shows an e-commerce web page that calls into multiple bounded contexts, each the authority for the desired information, to get special offers, prices, and other kinds of information. This happens each time the page is requested, so the information is fully up to date, not eventually consistent.

UIs that defer to the authority of each piece of information do not belong to a bounded context. Instead, many companies have dedicated web teams that don't own bounded contexts but are completely responsible for the website. If that approach doesn't work for you, you can let teams that own a bounded context also be responsible for UIs that defer to authority. It's just important to remember that conceptually the UI does not belong to their bounded context.

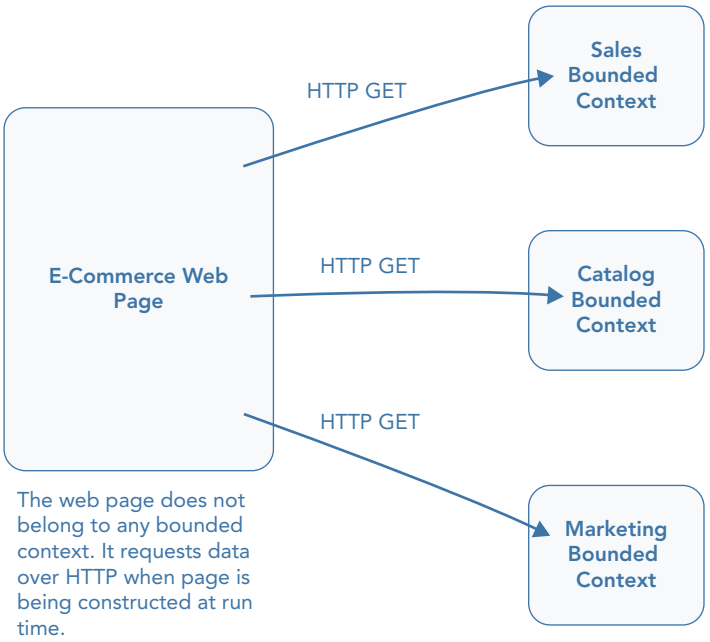


FIGURE 23-2: UI that defers to authority.

Some Help Deciding

A good starting point for choosing between autonomous and authoritative applications is to think about team relationships. If the UI is for a specific department, like an internal tool might be, it may be more efficient to keep it within that team. On the other hand, if the UI forms part of a bigger application that contains many UIs, such as a public website, you may want a dedicated web team to deal with all the web and front-end challenges.

Another consideration is the amount of extra data storage and complexity involved in enabling an autonomous application to have all the data it needs locally. If it is a lot of extra work for a relatively minor use case, the authoritative option might provide the most benefit for the least amount of short- and long-term effort. But if you do need fully up-to-date information, eventually consistent autonomous applications are probably not the best choice.

HTML APIs versus Data APIs

By constructing web pages with snippets of HTML that are returned from each bounded context, you give bounded contexts control of the appearance and behavior of specific regions of a page, as Figure 23-3 shows.

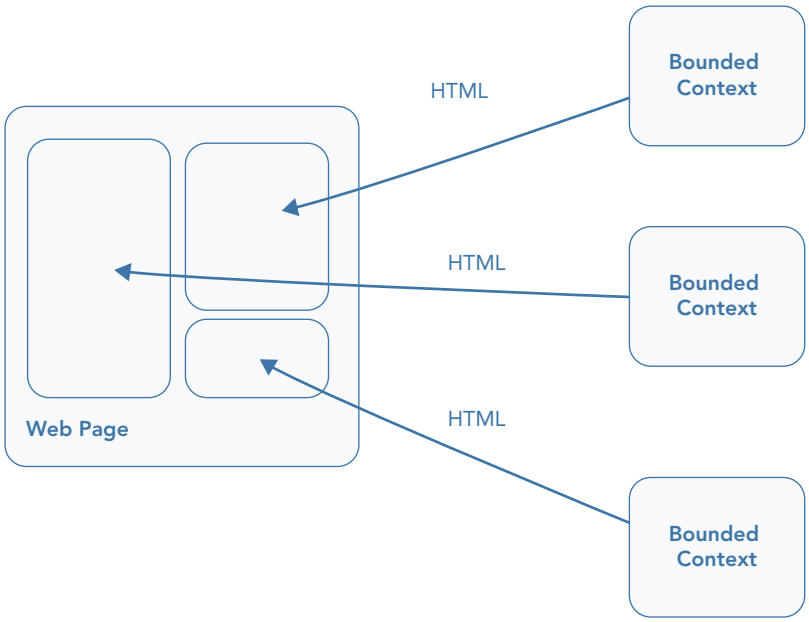


FIGURE 23-3: Composing a web page with HTML provided by bounded contexts.

Another option, which offers less presentational control to bounded contexts, is to have pages only pull in data from bounded contexts. With this alternative approach, you can manage all the presentation concerns in a single location, as shown in Figure 23-4.

Most online experience reports indicate that the second approach is by far the most prominent, and it's usually expressed as JSON APIs. But both approaches can work. One important consideration is whether you provide APIs that are used externally. It was mentioned in Chapter 13, "Integrating Via HTTP with RPC and REST," that dogfooding your API can have a number of benefits in such scenarios.

Client versus Server-Side Aggregation/Coordination

For a UI that pulls in content (data or HTML) from multiple bounded contexts, there is the choice of performing the aggregation on the client or the server. By making each request an AJAX request inside the web page, you can avoid the complexity and additional failure point of the server-side application. Conversely, you will have more complexity on the client as JavaScript. Building Single Page Applications (SPAs), as many teams are now, is one case in which this is less of a problem. The general recommendation on this topic tends to favor the client, but both approaches are in wide use. Figure 23-5 illustrates knitting together content on the client, whereas Figure 23-6 illustrates the server-side approach.

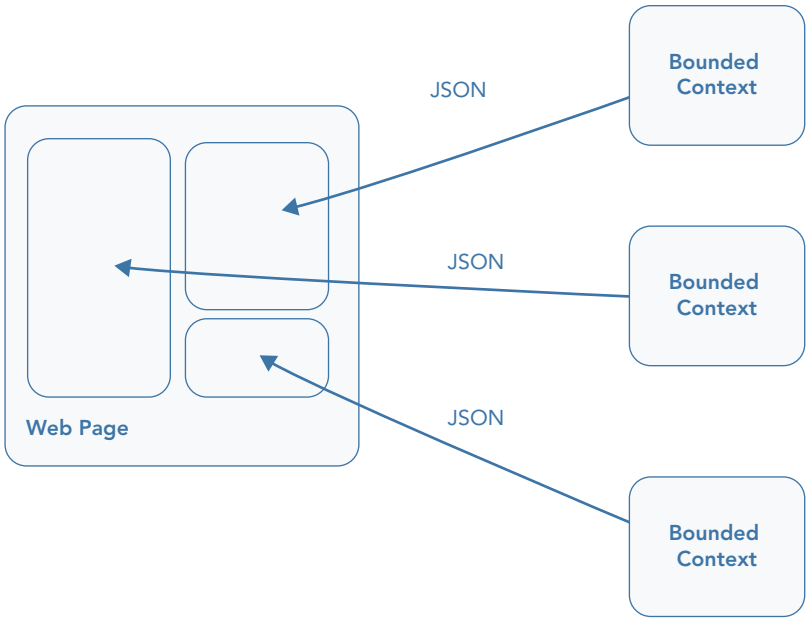


FIGURE 23-4: Pulling in data from multiple bounded contexts.

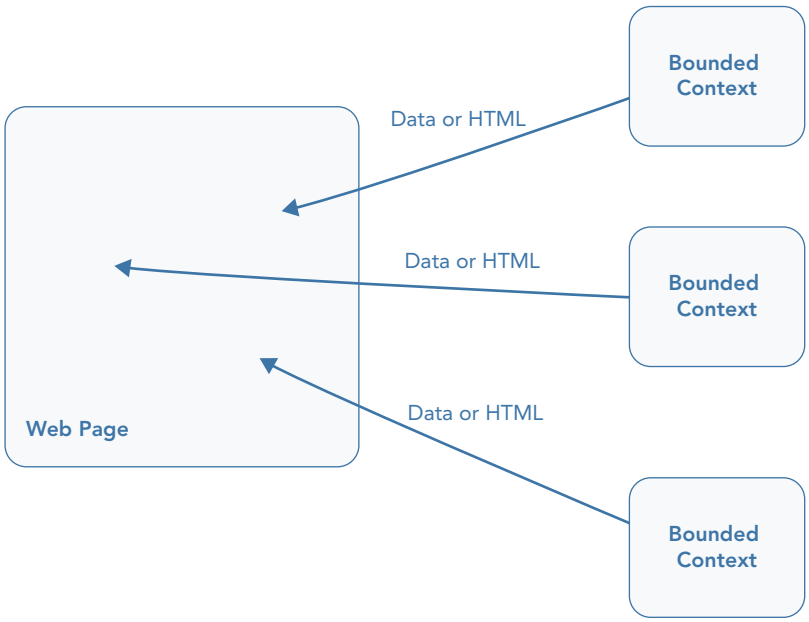


FIGURE 23-5: Aggregating on the client.

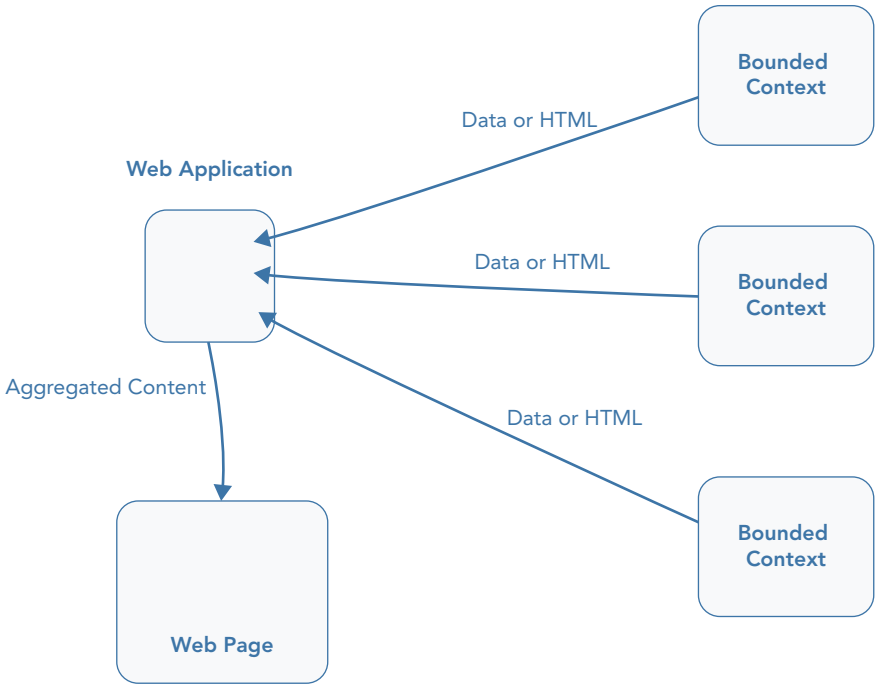


FIGURE 23-6: Aggregating on the server.

EXAMPLE 1: AN HTML API-BASED, SERVER-SIDE UI FOR NONDISTRIBUTED BOUNDED CONTEXTS

Even when all your bounded contexts live inside the same solution and run as a single application, UI composition can still be useful for partitioning presentational responsibility among bounded contexts. When one bounded context would like to alter its portion(s) of a page, the changes may be confined to that bounded context, meaning no interference with others. This is the same intention that motivates the Single Responsibility Principle (SRP). In this section, you implement this scenario using ASP.NET MVC's `RenderAction()`. You're going to create a simple page that pulls in HTML content from three bounded contexts that live inside the same solution, as shown in Figure 23-7.

To begin this example, you need to create a new ASP.NET web application called `PPPDDD.NonDist.UIComp`. Choose the Empty template, and check the MVC check box. This application contains only a single view, which will be the composite UI, so it's fine for it to be the initial page of the application. To achieve this, add a class called `HomeController` in the `Controllers` folder with the content shown in Listing 23-1.

> **NOTE** *ASP.NET MVC's default route is to look for an* `Index()` *method on a controller called* `HomeController`*. If your project has one of these, it is used to respond to the base URL (/).*

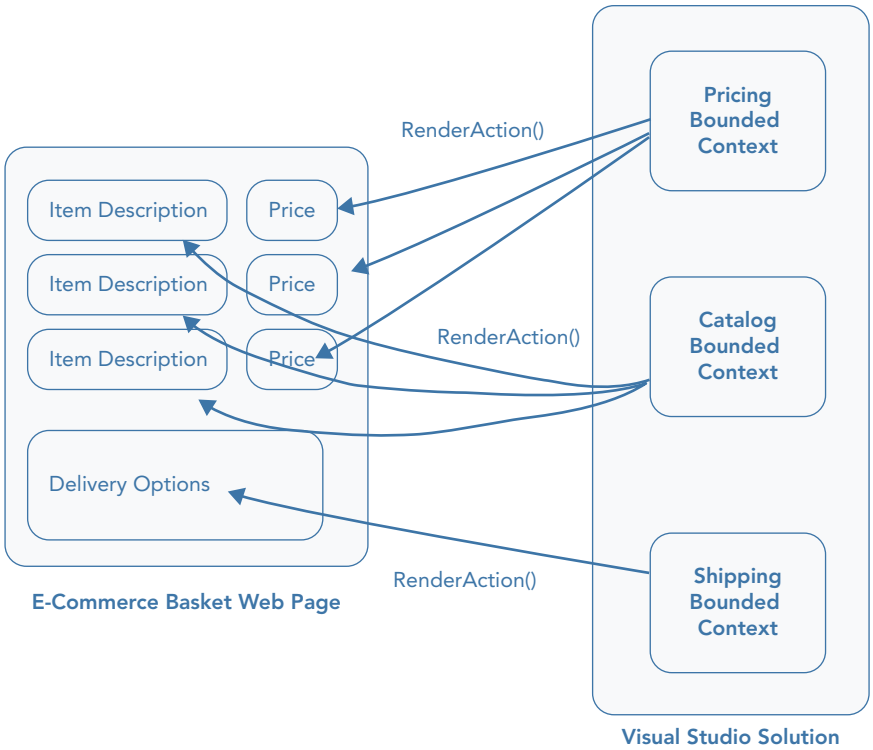


FIGURE 23-7: The design for this example.

LISTING 23-1: HomeController That Returns the Composite UI

```
using System;
using System.Web;
using System.Web.Mvc;

namespace PPPDDD.NonDist.UIComp.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

All the `HomeController` does is return a view, which you can create by adding a file called `Index.cshtml` in the `/Views/Home/` folder (which you must create). Once you've added the view, you need to replace its contents with the code shown in Listing 23-2.

LISTING 23-2: /Views/Home/Index.cshmtl—the Composite UI

```
@{
    Layout = null;
    var productIdsInBasket = new string[3] { "prod1", "prod2", "prod3"};
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>PPPDDD Composite UI</title>
    @*
        Can also use RenderAction here if bounded contexts
        need to inject custom javascript
    *@
</head>
<body>
    <div>
        <h1>Your Basket</h1>
        @foreach(var pid in productIdsInBasket)
        {
            <div class="basketItem" style="margin-bottom: 20px;">
    @{Html.RenderAction(
        "ItemInBasket","catalogBoundedContext", new{productId=pid});}
    @{Html.RenderAction(
        "Price","PricingBoundedContext",new{productid=pid});}
            </div>
            <br />
        }
        @{Html.RenderAction(
           "DeliveryOptions", "ShippingBoundedContext"); }
    </div>
</body>
</html>
```

Listing 23-2 shows a basic implementation of UI composition. You can see that the page itself is almost just a template. It pulls in its main content by directly rendering HTML provided by each bounded context by calling `RenderAction()`. The first argument is the name of the method to be called on a controller that has the name of the second argument. So in this example, the first `RenderAction()`calls `CatalogBoundedContextController.ItemInBasket()`, passing in the product ID of the catalog item that needs to be rendered. You can see at the top of the page that the list of product IDs is hard-coded. This keeps the example focused on the UI aspects and is not a recommended practice.

For this page to render correctly, you must implement the three controller methods to be called by each `RenderAction()`. Starting with the first, you need to add a class called `CatalogBoundedContextController` to the `Controllers` folder. It should contain the code shown in Listing 23-3.

LISTING 23-3: CatalogBoundedContextController

```
using System;
using System.Web;
using System.Web.Mvc;

namespace PPPDDD.NonDist.UIComp.Controllers
{
    public class CatalogBoundedContextController : Controller
    {
        [ChildActionOnly] // cannot be rendered as an individual page
        public PartialViewResult ItemInBasket(string productId)
        {
            var product =
                SalesBoundedContext.ProductFinder.Find(productId);

            /* convention will look for a partial view called:
             * /Views/CatalogBoundedContext/ItemInBasket.cshtml
            */
            return PartialView(product);
        }
    }
}

// This would actually be inside a separate project
namespace PPPDDD.NonDist.UIComp.SalesBoundedContext
{
    public static class ProductFinder
    {
        public static Product Find(string productId)
        {
            // simulate a database lookup
            return new Product
            {
                ID = productId,
                Name = "Product_" + productId,
                Description = "Lorem ipsum dolor sit amet",
                ImageUrl = "http://media.wiley.com/product_data/" +
                           "coverImage/84/04702927/0470292784.jpg"
            };
        }
    }

    public class Product
    {
        public string ID { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }

        public string ImageUrl { get; set; }
    }
}
```

Conceptually, the most important concern in Listing 23-3 is that this controller is getting all the information it needs from the Catalog bounded context by calling methods on its `ProductFinder`. You wouldn't want to call methods on other bounded contexts inside this controller because then you would have a coupling on multiple bounded contexts, where changes to one might affect the other. It might also lead to different teams getting in each other's way as they try to make changes at the same time.

Technically, the important detail is the call to `PartialView()`, which passes in a view model. This returns the HTML that is produced by a partial view page at the location `/Views/CatalogBoundedContext/ItemInBasket.cshtml` (which is a Razor view page). You can now add that file. The code for it is shown in Listing 23-4.

LISTING 23-4: ItemInBasket.cshtml

```
@model PPPDDD.NonDist.UIComp.SalesBoundedContext.Product

<div>
    <h3>@Model.Name</h3>
    <p>
        <img height="75" src="@Model.ImageUrl" style="float: left;" />
        @Model.Description
    </p>
</div>
```

In Listing 23-4, an HTML template is created that uses Razor syntax to populate placeholders with values from the passed-in view model. The HTML generated by this template is directly rendered onto the composite UI where the `ItemInBasket RenderAction()` occurs.

To complete this example, you need to add the other two controllers: `PricingBoundedContextController` and `ShippingBoundedContextController` in the `Controllers` folder. You also need to add the partial views: `/Views/PricingBoundedContext.Price.cshtml` and `/Views/ShippingBoundedContext/DeliveryOptions.cshtml`. The code for each of these files is shown in Listings 23-5 through 23-8.

LISTING 23-5: PricingBoundedContextController

```
using System;
using System.Web;
using System.Web.Mvc;

namespace PPPDDD.NonDist.UIComp.Controllers
{
    public class PricingBoundedContextController : Controller
    {
        [ChildActionOnly] // cannot be rendered as a page
        public PartialViewResult Price(string productId)
        {
```

continues

LISTING 23-5 *(continued)*

```
            var price = PricingBoundedContext
                        .PriceFinder
                        .PriceFor(productId);

            /* convention will look for a partial view called:
             * /Views/PricingBoundedContext/Price.cshtml
            */
            return PartialView(price);
        }
    }
}

// would actually live in a separate project
namespace PPPDDD.NonDist.UIComp.PricingBoundedContext
{
    public static class PriceFinder
    {
        private static Lazy<Random> random = new Lazy<Random>();

        public static int PriceFor(string productId)
        {
            // simulate a database lookup
            return random.Value.Next(1, 1000);
        }
    }
}
```

LISTING 23-6: Price.cshtml

```
@model int

<div class="price">
    $@String.Format(Model.ToString(), "##.##")
</div>
```

LISTING 23-7: ShippingBoundedContextController

```
using System;
using System.Web;
using System.Web.Mvc;

namespace PPPDDD.NonDist.UIComp.Controllers
{
    public class ShippingBoundedContextController : Controller
    {
        [ChildActionOnly] // cannot be rendered as a page
        public PartialViewResult DeliveryOptions()
```

```
        {
            var options = ShippingBoundedContext.DeliveryOptions.All();

            /* convention will look for a partial view called:
            * /Views/ShippingBoundedContext/DeliveryOptions.cshtml
           */
            return PartialView(options);
        }
    }
}

// would actually be in a separate project
namespace PPPDDD.NonDist.UIComp.ShippingBoundedContext
{
    using System.Collections.Generic;

    public static class DeliveryOptions
    {
        public static IEnumerable<DeliveryOption> All()
        {
            // simulate database lookup
            return new List<DeliveryOption>
            {
                new DeliveryOption
                {
                    ID = "ss1",
                    Name = "Cheap & Cheerful",
                    Price = 2,
                    Duration = new Tuple<int,int>(7, 14)
                },
                new DeliveryOption
                {
                    ID = "ss2",
                    Name = "Super Fast",
                    Price = 50,
                    Duration = new Tuple<int,int>(1, 2)
                }
            };
        }
    }

    public class DeliveryOption
    {
        public string ID { get; set; }

        public string Name { get; set; }

        public int Price { get; set; }

        public Tuple<int, int> Duration { get; set; }
    }
}
```

LISTING 23-8: DELIVERYOPTIONS.CSHTML

```
@model IEnumerable<PPPDDD.NonDist.UIComp.ShippingBoundedContext
                   .DeliveryOption>

<div class="deliveryOptions">
    <h2>Delivery Options</h2>
    @foreach(var option in Model)
    {
        <p>
            @Html.RadioButton("deliveryOptions", option.ID)
            @option.Name - $@String.Format(
                option.Price.ToString(), "##.##")
           (@option.Duration.Item1 - @option.Duration.Item2 days)
        </p>
    }
</div>
```

When running the application and navigating to the root URL, you should see the composite UI rendered, as shown in Figure 23-8.

EXAMPLE 2: A DATA API-BASED, CLIENT-SIDE UI FOR DISTRIBUTED BOUNDED CONTEXTS

Creating web pages that pull in information from multiple HTTP APIs is a common scenario in web development. In this example, you learn how to pull in information from multiple bounded contexts—each running as a separate application—and aggregate all the information using JavaScript directly in the browser. You can see the design this example simulates in Figure 23-9.

In Figure 23-9, the first key detail to note is that each API lives inside a bounded context that knows nothing about the other bounded contexts. You can write each bounded context in a completely different technology as long as it provides the required HTTP API. Another key detail is that each API returns JSON. This means all the presentation concerns are isolated in the main website.

> **NOTE** *As you work through this example, you will notice that each API lives inside the same ASP.NET project and does not run as a separate application, as shown in Figure 23-9. This helps to keep the example focused on the UI aspects, but it's crucial to remember that this example is simulating the design in Figure 23-9, where each API is a standalone application.*

To start this example, you need to create a new ASP.NET web application called `PPPDDD.Dist.UIComp`. As before, when creating the project, it is important to select the Empty template and check the MVC check box. With your project created, you can add a `HomeController` and its

corresponding view `/Views/Home/Index.cshtml`, which is automatically chosen as the default page for the application. The content of `HomeController` is shown in Listing 23-9, and the content of `Index.cshtml` is shown in Listing 23-10.

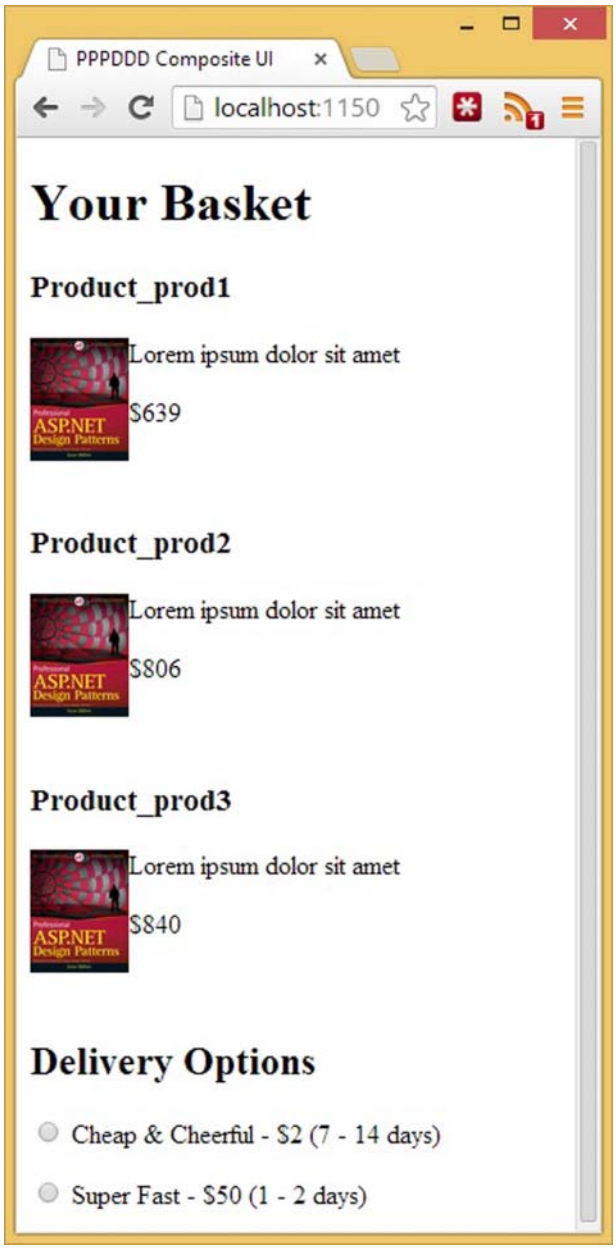


FIGURE 23-8: Rendering of composite UI.

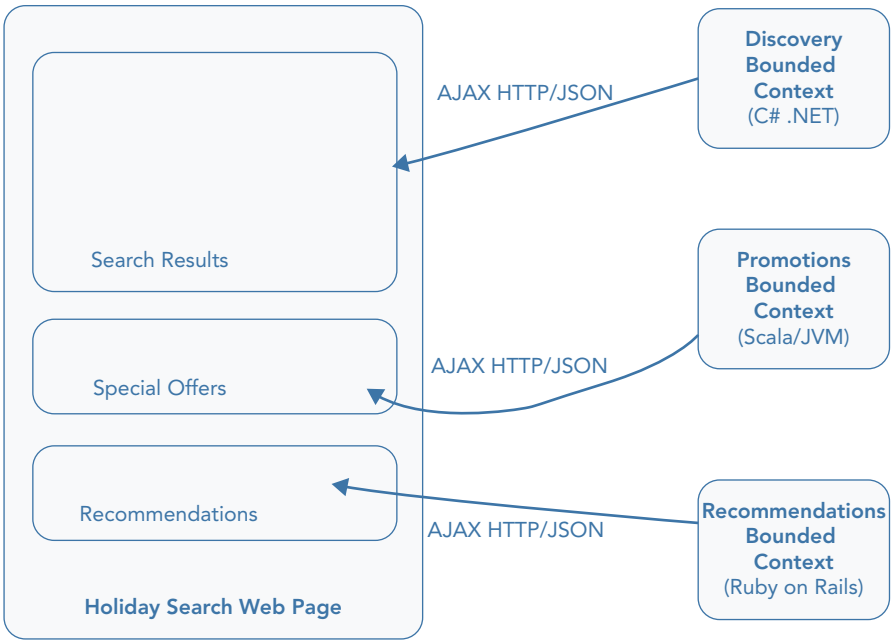


FIGURE 23-9: The design for this example.

LISTING 23-9: HomeController

```
using System;
using System.Web;
using System.Web.Mvc;

namespace PPPSSS.Dist.UIComp.Controllers
{
    public class HomeController : Controller
    {
        public ViewResult Index()
        {
            return View();
        }
    }
}
```

LISTING 23-10: /Views/Home/Index.cshtml

```
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
```

```
<head>
    <meta name="viewport" content="width=device-width" />
    <title>PPPDDD Distributed Composite UI</title>
    <script src="//code.jquery.com/jquery-1.11.0.min.js"></script>
    <script src="/Scripts/pppddd-application.js" type="text/javascript">
    </script>
</head>
<body>
    <div>
        <p>You searched for: Moderately Priced Autumn Sun</p>
        <div id="holidays">
          <h3>Holidays Matching Your Search</h3>
        </div>
        <div id="promotions">
            <h3>Special Offers</h3>
        </div>
        <div id="recommendations">
            <h3>Holiday Recommendations Just For You</h3>
        </div>
    </div>
</body>
</html>
```

In Listing 23-9, the `HomeController` renders the view. You can see the markup for the view in Listing 23-10. Notice that there is more content than in the UI from Example 1. This is because the bounded contexts only provide data and not HTML markup. Some of the markup is dynamically added as the data is asynchronously fetched from each bounded context's HTTP API. This happens inside the `pppddd-application.js` JavaScript file that is referenced in the head of the page. The content of `pppddd-application.js` is shown in Listing 23-11. You need to add this file to the `Scripts` folder.

LISTING 23-11: /Scripts/pppddd-application.js

```
function createHolidayView(holiday) {
    return '<div> ' +
           '<img src="' + holiday.ImgUrl + '" height="75"' +
               ' style="float: left;" /> ' +
           '<h4>' + holiday.Title + '</h4>' +
           '$' + holiday.Price + 'pp' +
           '</div>' +
           '<br />';
}

// when the page loads
$(document).ready(function() {

    // query for holidays that simulate a user search
    $.getJSON("/holidays", function (json) {
        $.each(json, function (index, holiday) {
            $('#holidays').append(createHolidayView(holiday));
```

continues

LISTING 23-11 *(continued)*

```
        });
    });

    // query for promoted holidays
    $.getJSON("/promotions", function (json) {
        $.each(json, function (index, holiday) {
            $('#promotions').append(createHolidayView(holiday));
        });
    });

    // query for user-specific recommendations
    $.getJSON("/recommendations", function (json) {
        $.each(json, function (index, holiday) {
            $('#recommendations').append(createHolidayView(holiday));
        });
    });
});

/*
* Jquery used for demonstration purposes. Other frameworks
* may be a better choice, including Angular.js, Knockout.js, etc.
*/
```

In `pppddd-application.js`, a web request for JSON is made to each bounded context when the page loads. When the JSON response is received, relevant parts of the page are updated. One important detail is the `createHolidayView()` function. Notice how it generates HTML that will be rendered on the page. This demonstrates that all presentation concerns remain inside the web application as opposed to being scattered among each bounded context as would be the case if the APIs returned HTML. (This is neither better nor worse; it's a conscious choice you need to make based on the trade-offs.)

> **WARNING** *Listing 23-11 has some shortcuts to keep the example simple. For instance, each HTTP API is required to return JSON in the same format (shown in Listing 23-12). In a real production system, this is unlikely and discouraged without good reason due to coupling. Another shortcut is the generation of HTML in the* `createHolidayView()` *function. In a real application, you may want to use templating libraries such as* `handlebars.js` *(*`http://handlebarsjs.com/`*).*

LISTING 23-12: Holidays JSON Format Used in This Example

```
[
   {
      "Title":"...",
      "Price": xx,
      "ImgUrl":"http://..."
```

```
    },
    ...
]
```

Your UI is now complete. All that is lacking is an HTTP API that each bounded context provides. In a real distributed system, each of those APIs would be running as a separate application inside a different bounded context. But in this example, they live inside the same web application to make your life easier. To add those APIs to the project, you need to add a `HolidaysController`, a `PromotionsController`, and a `RecommendationsController` in the `Controllers` folder. The code for each of these classes is shown in Listings 23-13 through 23-15.

LISTING 23-13: HolidaysController

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;

namespace PPPSSS.Dist.UIComp.Controllers
{
    /*
    * This controller represents an API provided by the Search
    * bounded context. It would run as its own dedicated application
    * and not live inside the same project as the other APIs
    * currently in this project
    */
    public class HolidaysController : Controller
    {
        public JsonResult Index()
        {
            var holidays = new List<Holiday>
            {
                new Holiday
                {
                    Title = "2 Weeks in Rhodes",
                    Price = 688,
                    ImgUrl = "http://media.wiley.com/product_data/" +
"coverImage/84/04702927/0470292784.jpg"
                },
                new Holiday
                {
                    Title = "1 Week in Barbados",
                    Price = 320,
                    ImgUrl = http://media.wiley.com/product_data/" +
"coverImage/84/04702927/0470292784.jpg"
                }
            };

            return Json(holidays, JsonRequestBehavior.AllowGet);
        }

        class Holiday
```

continues

LISTING 23-13 *(continued)*

```
        {
            public string Title { get; set; }

            public int Price { get; set; }

            public string ImgUrl { get; set; }
        }
    }
}
```

LISTING 23-14: PromotionsController

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;

namespace PPPSSS.Dist.UIComp.Controllers
{
    /*
    * This controller represents an API provided by the Search
    * bounded context. It would run as its own dedicated application
    * and not live inside the same project as the other APIs
    * currently in this project
    */
    public class PromotionsController : Controller
    {
        public JsonResult Index()
        {
            var holidays = new List<Holiday>
            {
                new Holiday
                {
                    Title = "Relaxing Med Cruise",
                    Price = 999,
                    ImgUrl = "http://media.wiley.com/product_data/" +
"coverImage/84/04702927/0470292784.jpg"
                },
                new Holiday
                {
                    Title = "Romantic Weekend Break in Paris",
                    Price = 120,
                    ImgUrl = http://media.wiley.com/product_data/" +
"coverImage/84/04702927/0470292784.jpg"
                }
            };

            return Json(holidays, JsonRequestBehavior.AllowGet);
        }

        class Holiday
```

```
        {
            public string Title { get; set; }

            public int Price { get; set; }

            public string ImgUrl { get; set; }
        }
    }
}
```

LISTING 23-15: RecommendationsController

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;

namespace PPPSSS.Dist.UIComp.Controllers
{
    /*
    * This controller represents an API provided by the Search
    * bounded context. It would run as its own dedicated application
    * and not live inside the same project as the other APIs
    * currently in this project
    */
    public class RecommendationsController : Controller
    {
        public JsonResult Index()
        {
            var holidays = new List<Holiday>
            {
                new Holiday
                {
                    Title = "2 Weeks in Mykonos",
                    Price = 450,
                    ImgUrl = "http://media.wiley.com/product_data/" +
"coverImage/84/04702927/0470292784.jpg"
                },
                new Holiday
                {
                    Title = "2 Weeks in Kos",
                    Price = 365,
                    ImgUrl = "http://media.wiley.com/product_data/" +
"coverImage/84/04702927/0470292784.jpg"
                }
            };

            return Json(holidays, JsonRequestBehavior.AllowGet);
        }

        class Holiday
        {
```

continues

LISTING 23-15 *(continued)*

```
            public string Title { get; set; }

            public int Price { get; set; }

            public string ImgUrl { get; set; }
        }
    }
}
```

As you can see, each controller does the bare minimum to return JSON in the required format (see Listing 23-12). In a real application, this is where your application services take control. As you'll learn in the next chapter, application services sit between the domain and external contracts, such as an HTTP API, to coordinate actions based on data coming in from the UI or going out to the UI. Before moving on to the next chapter, if you press F5 inside your project, you should see the page rendered as per Figure 23-10.

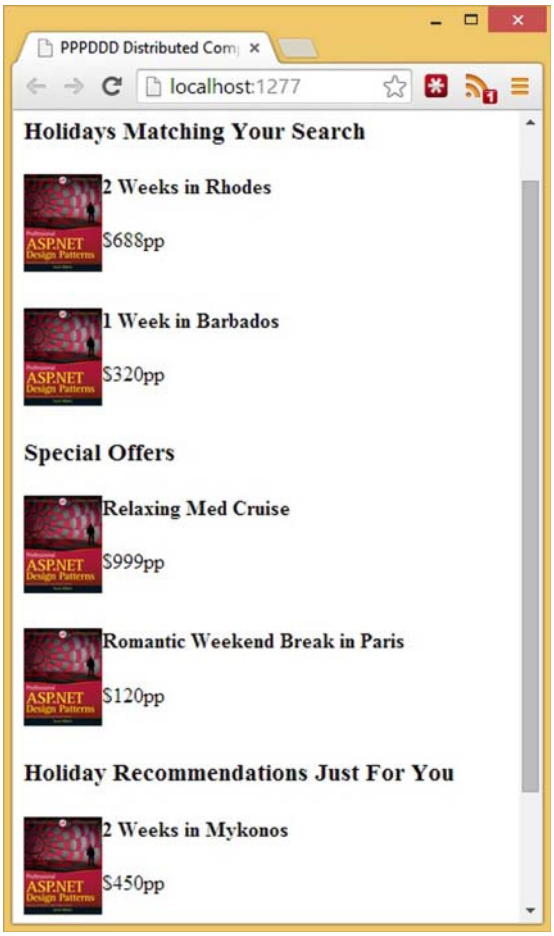


FIGURE 23-10: Client-side JSON API composition in action.

THE SALIENT POINTS

- The arrangements of the back-end bounded contexts can heavily influence the user interface and vice versa.
- Deciding which team owns a UI can significantly affect the team dynamics and engineering solutions.
- Pulling in data from multiple bounded contexts can occur on the client as JavaScript or on the server with your preferred technology.
- Client-side composition can reduce the complexity and coupling of an additional server component.
- Server-side aggregation and orchestration remove the reliance on JavaScript and the performance constraints of running inside a browser.
- UIs can be composed of HTML, or they can pull in data from each bounded context as JSON or XML.
- Composition with HTML gives more control to each bounded context but distributes presentational concerns.
- Aggregation of data isolates presentation concerns to a single web application but removes ownership of presentation concerns from each bounded context.

24

CQRS: An Architecture of a Bounded Context

WHAT'S IN THIS CHAPTER?

- The challenges of using a single model for complex presentation and domain logic needs
- How the CQRS pattern segregates the reading and writing models
- The popular misconceptions of the CQRS pattern
- How CQRS can help bounded contexts to scale

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The `wrox.com` code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 24 download and individually named according to the names throughout the chapter.

CQRS (Command Query Responsibility Segregation) is a simple pattern that you can apply to a bounded context. It separates the domain model into two models: a read model and a write model (sometimes called a transactional model).

The reason for the separation is to enable a model to serve the needs of a single context without compromise. The two contexts in question are reporting on the state of the domain and performing business tasks, also known as the read and write sides. Using a single model for bounded contexts that have complex presentation needs and rich domain logic often results in that model becoming overly complex and devoid of integrity, generating confusion for domain experts and a maintenance nightmare for developers. By applying the CQRS pattern, a model is split in two, enabling each model to be optimized to serve each context more effectively.

CQRS is not a top-level architecture; it is a pattern for handling complexity that can be applied against bounded contexts needing to support a presentation model that is not aligned to the structure of the transactional model. The majority of web applications see a disparity between queries and commands. CQRS splits these two and enables the sides to be optimized without compromise.

This chapter introduces you to the CQRS pattern, giving you a holistic understanding of where it can be effective. Chapters 25, "Commands: Application Service Patterns for Processing Business Use Cases," and Chapter 26, "Queries: Domain Reporting," go deeper into the implementation of the command and query sides of the architecture.

THE CHALLENGES OF MAINTAINING A SINGLE MODEL FOR TWO CONTEXTS

Figure 24-1 shows a typical layered architecture of a bounded context. Within the heart of this architecture lies a domain model. The domain model is created to enforce the invariants of the domain when handling transactional operations. The model is composed of small aggregate groupings of domain objects built for consistency and that express the rules and logic of the domain. The reporting needs of an application, however, may not be aligned with the structure of the aggregates, resulting in application services needing to Load many different aggregates to construct view models that may contain only a subset of the data that is retrieved within the aggregate instances. This view generation can quickly become complex and difficult to maintain and in the very worst scenarios can slow down the system.

To support view generation, domain models need to expose internal state and need to be adorned with presentation properties that have little to do with the invariants of the domain. Repositories often contain many extra methods on the contract to support presentation needs such as paging, querying, and free text searching. Because the read side of an application is typically used more frequently than the write side, users often seek improvements in report generation. To try to simplify as well as improve performance of the query side, the model is compromised. Aggregates are merged, and lazy loading is used to prevent pulling data that is not required for transactional business task processing needs, but that *is* required for presentational purposes. This leads to a single model that is full of compromises and is sub-standard for both reading and writing.

A BETTER ARCHITECTURE FOR COMPLEX BOUNDED CONTEXTS

Figure 24-2 demonstrates an architecture that employs the CQRS pattern. It treats the needs of the two conflicting contexts—namely, reads and writes—separately by providing two models instead of one. Each model can now be optimized for the specific context it serves while retaining its conceptual integrity. You are in a sense applying the bounded context pattern at a lower level by binding one model for the read context and a separate model for the write context.

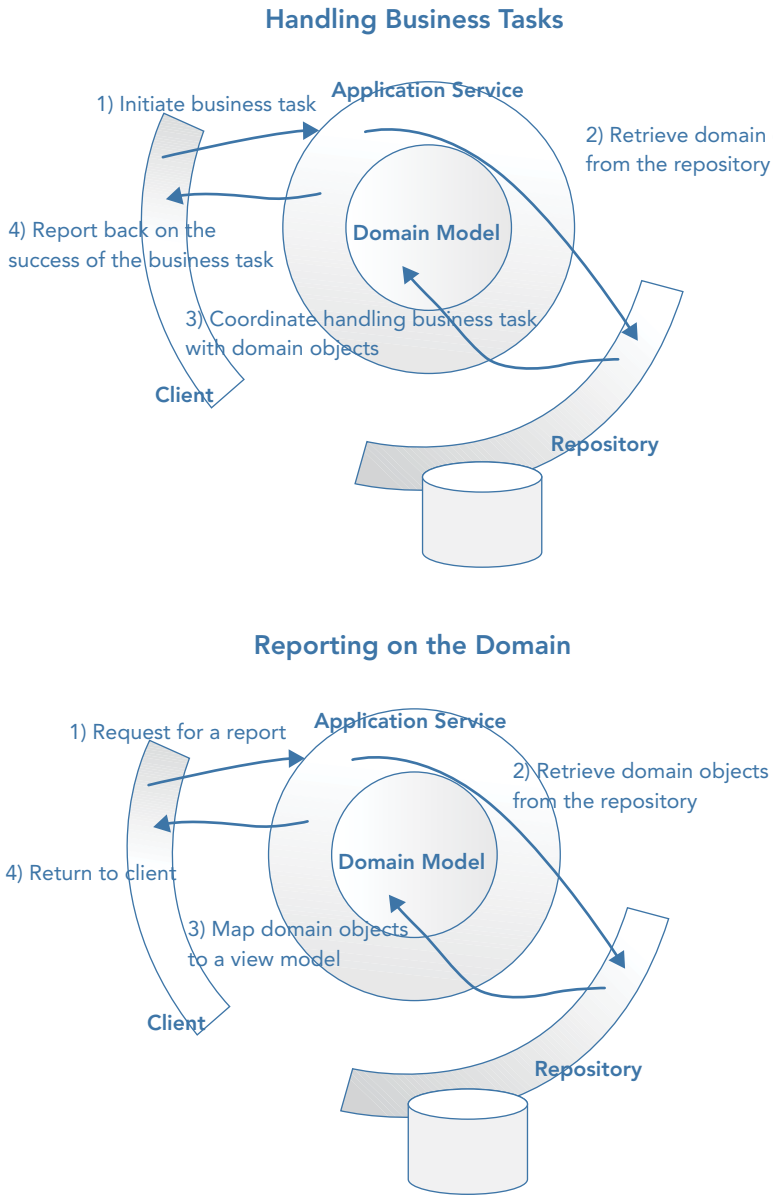


FIGURE 24-1: A single model fulfilling the read and write sides of an application.

Figure 24-2 shows the segregation between commands; the responsibility to fulfill business tasks, which are invoked by a client, and queries and the responsibility to fulfill reports, which are requested by a client. In Figure 24-2, the same data store is used for both the read and the write side of the architecture. This is not mandatory; a separate read store can be employed to scale the read side.

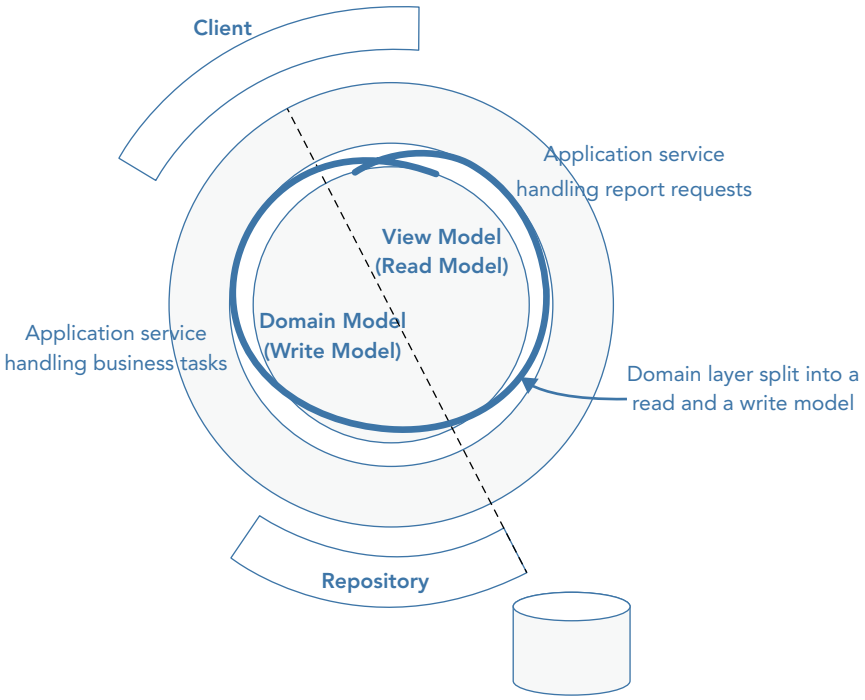


FIGURE 24-2: The CQRS pattern with a separate read and write model.

THE COMMAND SIDE: BUSINESS TASKS

The command side of the architecture is concerned with upholding the rules of the domain. It represents the domain logic that satisfies business tasks. The architecture shown in Figure 24-3 at first glance is the same as the typical layered approach; however, the command side does not support querying, and any responses are merely acknowledgements on the success of the business task a client initiates.

Explicitly Modeling Intent

A command is a business task, a use case of a system, and it lives within the application layer. You should write commands in the language of the business. This is not UL; it is the language that captures the behaviors of the systems rather than the terms and concepts of the domain model. Typically, if you are following a BDD approach, commands come from the use cases and stories you produce.

You should model commands as verbs rather than nouns. They should capture the intent of the user explicitly. An example of a command is shown in Listing 24-1. A command is a simple data transfer object (DTO) with simple parameter validation.

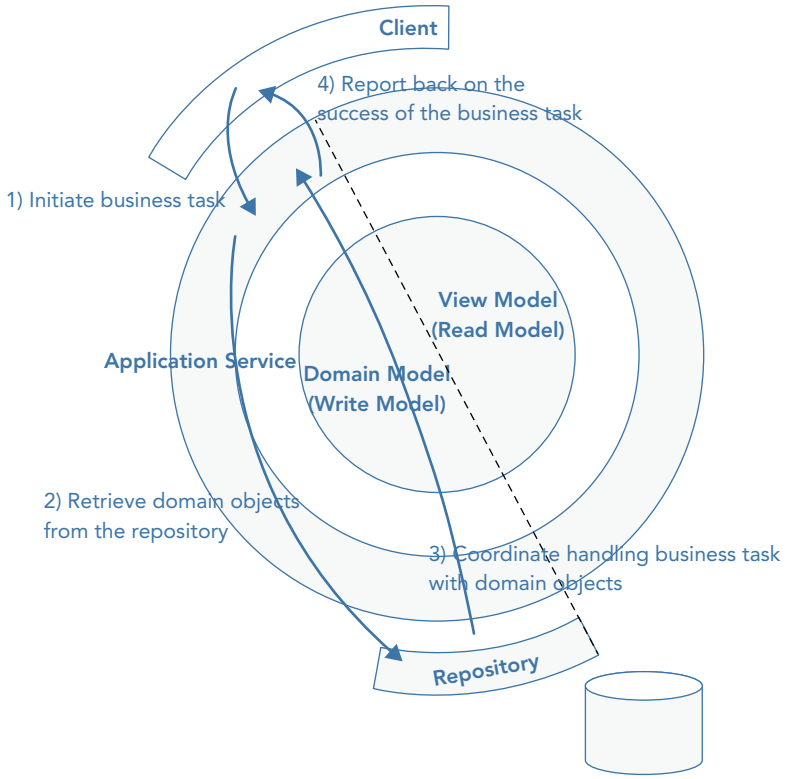


FIGURE 24-3: The command side of CQRS.

LISTING 24-1: An Example of An Explicit Command Object

```
public class CustomerWantsToRedeemAGiftCertificate
{
    public CustomerWantsToRedeemAGiftCertificate(Guid accountId,
                                                 string giftCertificate)
    {
        AccountId = accountId;
        GiftCertificate = giftCertificate;

    }
    public Guid AccountId { get; private set; }
    public string GiftCertificate { get; private set; }
}
```

As you can see from Listing 24-1, the command name reveals the intent of a user. In this case, it is for a customer to redeem a gift certificate. The command represents a request for a business task to be actioned and is therefore written in the present tense (for example: I want to do something) as opposed to domain events, which are written in past tense (for example: Something happened).

A Model Free from Presentational Distractions

A model that serves both the presentational and the transitional needs of an application often resembles the user interfaces of that application. The shape of aggregates is morphed from handling invariants into structures that match the user interface. For example, take the mockup of a user interface in Figure 24-4. This dashboard-like screen presents various attributes of a customer.

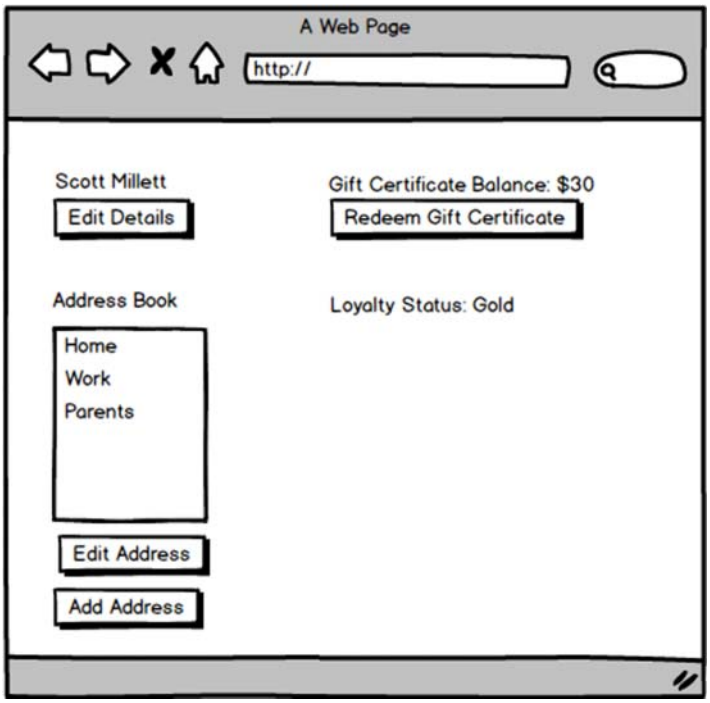


FIGURE 24-4: A user interface pulling data from many aggregates.

Listing 24-2 shows the type of domain object that is often created to meet the needs of the presentation while also implementing the logic of the domain.

LISTING 24-2: A Domain Object Used for Both Behavior and User Interface Needs

```
public class Customer
{
    // ...
    public ContactDetails ContactDetails { get; private set; }
    public LoyaltyStatus LoyaltyStatus { get; private set; }
    public Money GiftCertBalance { get; private set; }
    public IEnumerable<Address> AddressBook { get; private set; }
}
```

A view is now easy to generate for the presentation model because the domain model is in complete alignment with the user interface. It is simply a case of retrieving the full customer aggregate and mapping it to a view model, as demonstrated in Listing 24-3.

LISTING 24-3: A Specific View Model for User Interface Purposes Only

```
public class CustomerDashBoardView
{
    // ...

    public CustomerViewModel Generate(Guid customerId)
    {
         var customer = _customerRepository.FindBy(customerId);

         var customerView = MapCustomerViewModelFrom(customer);

         return customerView;

    }

}
```

However, the aggregate is now very large and is responsible for anything associated with a customer. The aggregate is structured around the UI rather than the invariants of the domain. In other words, the aggregates are based on a report screen instead of domain behavior. You can avoid these issues by applying CQRS and freeing the model from any presentational requirements. In the command model, there is no benefit in creating a single concept of a customer, often seen as a code smell and referred to as the god object. Instead, there will be an aggregate responsible for the rules governing loyalty, a separate aggregate for customer details, and a third for gift certificate balance. The UL will fit better around the behaviors of application than the UI screen. Domain experts will talk about behavior and rules, not about UI.

Without its added responsibilities the command model can be smaller and more focused on behavior with application services, and aggregates can become more concise. Repositories can be massively simplified as aggregate retrieval is restricted by ID rather than a host of querying methods such as paging and sorting. Developers can model aggregates around invariants and focus on transactional behavior without the noise of the presentation needs.

Handling a Business Request

A command handler is a flavor of an application service. The handler processes the command and contains logic to orchestrate the completion of a task. This logic can include delegation to the domain model, persistence and retrieval, and calling out to infrastructure services such as e-mail clients of payment gateways.

A command handler only returns an acknowledgement of the success or failure of a command; you should not use it to query or report in the domain state. Listing 24-4 presents an example of a command handler. Details of how to implement the command handler pattern are covered in the next chapter.

LISTING 24-4: A Command Handler, An Implementation of the Application Service

```
public class CreateOrUpdateCategoryHandler
{
    // ...

    public ICommandResult Execute(CreateOrUpdateCategoryCommand command)
    {
        var category = new Category
        {
            CategoryId = command.CategoryId,
            Name = command.Name,
            Description = command.Description
        };
        if (category.CategoryId == 0)
            categoryRepository.Add(category);
        else
            categoryRepository.Update(category);
        unitOfWork.Commit();
        return new CommandResult(true);
    }
}
```

Because a domain model on the command side is built to implement domain rules and logic, it does not need to contain unnecessary presentational properties. DDD aggregates support command processing rather than model real life. Handlers can help to focus aggregates on behavior and invariants rather than on real life. Specific commands don't need full domain entities when you're handling them. (That is, they don't need the customer name when you're performing some action on the customer aggregate.) Customer entity doesn't make sense. This helps keep aggregates small. This again aids you in modeling smaller aggregates with fewer associations.

THE QUERY SIDE: DOMAIN REPORTING

The architecture of the query side, as shown in Figure 24-5, is concerned with reporting on the domain. The objects returned from the query side are simple DTO view models tailored to the specific needs of the view. The domain model for the command side is not required, because a view can be generated directly from the data store. The query side does not need to create an abstraction over the persistence store, so a repository in this context makes no sense; a persistence framework or light-weight libraries like ADO.NET should be used here.

Because the read model is still within the domain layer, it is able to use domain objects from the read side to perform calculations if this data is not precalculated within the data store. Typically, specification classes have been employed to provide an answer to view model properties based on some data pulled back from the database.

Reports Mapped Directly to the Data Model

The read side of the architecture maps report requests modeled as view models directly to the data model, bypassing the command model completely. Views can be built within the data model for each UI screen or report. This results in presummarized views, which are fast to retrieve, simple

when mapping the raw data to view models, and able to handle paging and sorting and free text searches.

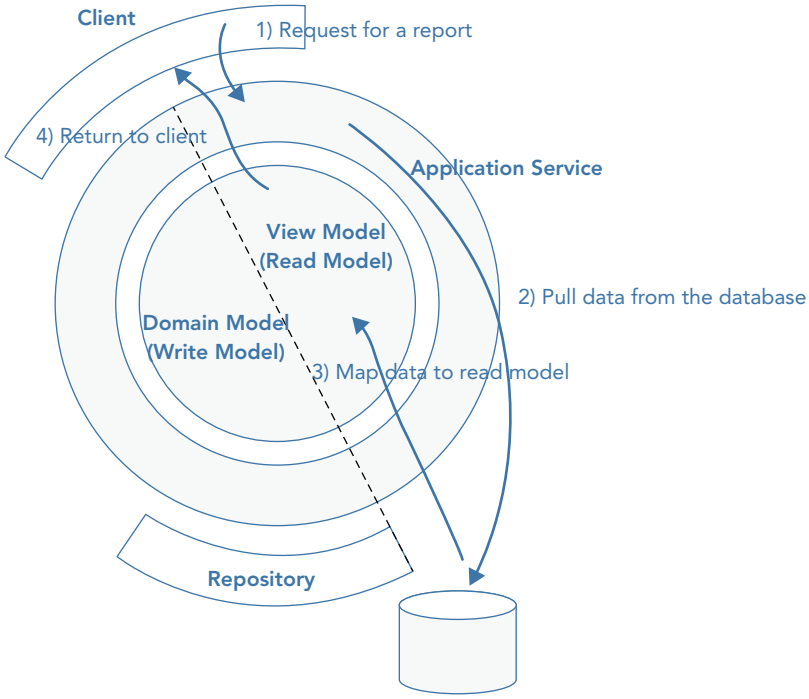


FIGURE 24-5: The query side of CQRS.

If the command side does not pre-computer a required value, use a specification or domain service to calculate the value on the fly as shown in Listing 24-5.

LISTING 24-5: Query Objects Can Reuse Domain Services and Specifications

```
public class OrderQuery
{
    // ...

    public OrderViewModel Generate(Guid customerId)
    {
         var customer = _customerRepository.FindBy(customerId);

         var customerView = MapCustomerViewModelFrom(customer);

         customerView.IsInfluential =
                        _influentialSpec.SatisfiedBy(customer);

         return customerView;

    }

}
```

You can use a micro Object Relational Mapper (ORM) in the query service—something lightweight that can quickly map raw data to a DTO view model. The read side will be simple—devoid of any logic save that of pulling data and mapping to DTOs and delegating to specifications or domain services for decisions based on saved state.

Materialized Views Built from Domain Events

You can go further still with the segregation of read and write by using a different data schema. You can build a read model from domain events raised from commands; you can then use these events to build materialized views, as shown in Figure 24-6.

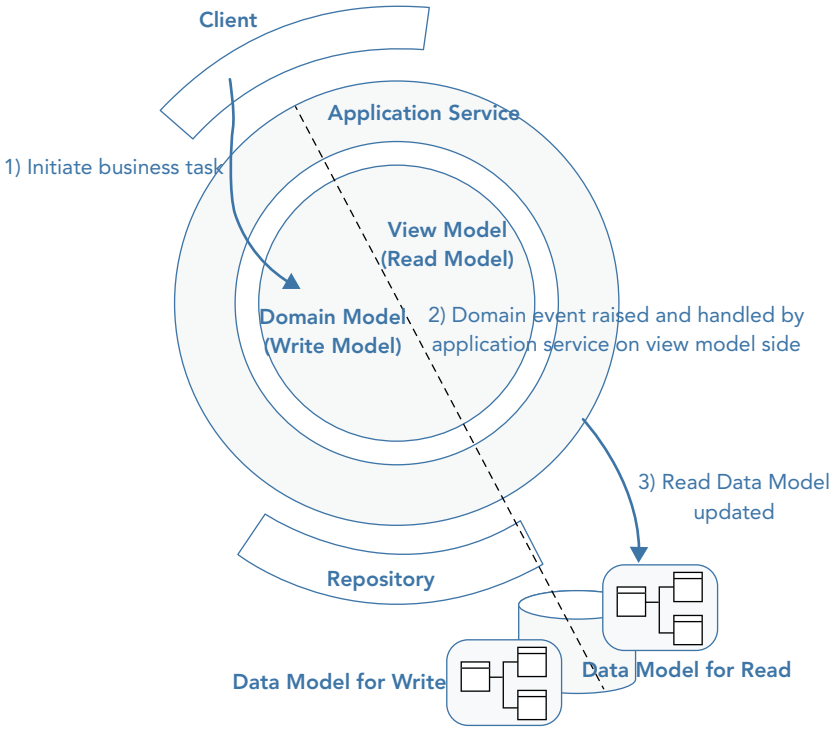


FIGURE 24-6: Using a different data store for querying.

A read data model can be denormalized and optimized for querying, including precalculated data.

Listing 24-6 shows how an action on the command model leads to a domain event being raised and then persisted by updating the read model. This can happen within the same transaction and the same database. Later you will see how updating the read model out of process can enable your application to scale.

LISTING 24-6: Command Handlers Can Orchestrate the Update of View Models

```
public class ModifiyCategoryHander
{
    public void Execute(ModifyCategoryCommand command)
```

```
        {
            var category = _catalogueRepository.FindBy(commmand.Id);

            using (DomainEvents.Register<CategoryUpdated>(onCategoryUpdated))
            {
                category.Update(command);
            }
        }

        private void onCategoryUpdated(CategoryUpdated @event)
        {
            _catalogueViewModel.Update(CategoryUpdated);
        }
    }
```

THE MISCONCEPTIONS OF CQRS

There are many misconceptions about the pattern of CQRS, but as you have read, it is simple, and at its core is the case of using a specific model for a specific context. If you have read anything of CQRS online, you may be thinking that you need to use heavyweight messaging frameworks, or your read store needs to be eventually consistent. This is not the case, although as you will read later you can employ these techniques to extend the CQRS pattern to scale your application when required. This section lists many of the popular misconceptions of the CQRS pattern that you may have heard.

CQRS Is Hard

If you have read this chapter from the beginning, you should be comfortable with CQRS, and you should have realized that it is a simple pattern. At a fundamental level, it's an implementation of the Single Responsibility Principle (SRP) applied at the domain model layer. It's useful for solving the complexity that arises when a presentational model is not in alignment with a transactional model. CQRS does not prescribe frameworks, multiple databases, or design patterns. It only states that the two contexts should be handled separately for better effectiveness. It's a conceptual mind shift rather than a collection of complex patterns and principles that you need to adopt.

CQRS Is Eventually Consistent

Eventual consistency is the practice of having a read model updated out of process and asynchronously to the update of the transactional model. This is not a prerequisite of CQRS, but it is often used to enable the read side of a model to scale. Eventually consistent read models add an extra layer of complexity to an application as users who check to see the result of their actions may be surprised to see an outdated screen. CQRS does not require you to be eventually consistent. You can use the same database and transaction to update the read model schema. In fact, your application's read store may already be eventually consistent if you are heavily using caching. If you are adopting the CQRS pattern due to complexities resulting in the misalignment between presentation and transaction concerns, try starting off being immediately consistent, and only move to eventually consistent if you have performance issues. There is an overhead that you will learn about later in this chapter through utilizing eventually consistent read stores.

Your Models Need to Be Event Sourced

As covered in Chapter 23, using event sourcing is an effective method to build both the read and the write models; however, there is no prerequisite to using event sourcing or in fact domain events with CQRS. Event sourcing is a solution to a problem of ensuring that your audit trail is accurate, but it does make building the read model easier because you can create whatever projections you want from the historical event data.

Commands Should Be Asynchronous

CQRS does not insist on commands being sent in a fire-and-forget fashion. For highly collaborative domains, in which multiple users are making changes to the same data, asynchronous commands make sense. This enables them to be handled in turn and allows the application to scale and not be overwhelmed with load. However, commands that don't return an acknowledgement regarding the success or failure require other ways to update the user to the success of an action. This could be via e-mail or extra behavior that handles failed messages. In the case of purchasing, this could be sourcing a substitution for the customer as opposed to simply failing an order outright.

CQRS Only Works with Messaging Systems

If you are looking to apply an eventually consistent read store or process commands asynchronously, then using a messaging framework is probably a good idea. However, if you are not, then adding a messaging system to your application is just needless complexity.

You Need to Use Domain Events with CQRS

Using events to build a materialized read model is an effective method to keep your read and write models separate; however, it is not critical, and you can use other methods of creating a materialized read store. As you have seen in Chapter 21, "Repositories," aggregates can reveal a state by using the memento pattern. You can also use some of the patterns in Chapter 26 to provide presentation information directly from your domain objects in a nonobstructive manner. Finally, you can build views based on the relational data model of the write model.

PATTERNS TO ENABLE YOUR APPLICATION TO SCALE

CQRS enables applications to perform well under heavy load. This is accomplished by the read and write side of an application being split. Separating the sides enables each to be scaled independently to meet the particular demands of the application. Read and write data can also be separated into stores that are best suited to their needs. A write store that deals solely with aggregates can utilize a document database or a key value store. A read model can utilize a relational database or a caching store.

However, to scale either side, you need to understand the trade-offs involved. Scaling out a system isn't simply a technical decision. It is vital that the business understands that changes to the architecture of a system result in changes to the user experience, and such changes need to be handled carefully *and* be acceptable to the business. The CAP theorem discussed in Chapter 11, "Introduction to Bounded Context Integration," states you can have two of the following: consistency, availability, and partition

tolerance. By sacrificing immediate consistency and moving to a more eventually consistent read side, you are able to scale applications that have high demands on the report from the system. For domains that are highly collaborative, you can sacrifice the availability guarantee, by not handling a request immediately. This involves messages sent to the system being queued and handled out of process by other means, such as e-mail, or a status report signifying to the client the success of the request.

The most important point to take way is that any trade-offs made to scale your application affect user experience, and this affects your business. Therefore, it is important that the business make decisions that affect user experience. This section examines ways to scale both the read and the write side using the CQRS pattern, along with the trade-offs that you need to consider.

Scaling the Read Side: An Eventually Consistent Read Model

If the demands of your application are far greater on the read side than on the write side, having an eventually consistent read store can allow you to increase the availability and performance of your application. You can store a read model in a separate database from your write side, or you can replicate it to multiple databases or any persistence store. The way you store your data on the read side can be completely different from the way you store the state of your domain on the write side. However, to do so you must publish state changes from the write side to the read side so that they can be denormalized and stored specifically for query retrieval. Figure 24-7 shows that there is a queue between the read and the write side of the architecture. This queue contains a domain event that is raised upon a state change within the domain model. The read side processes that domain event and updates its read store. Your read side becomes eventually consistent as it grows slightly out of sync with the write side.

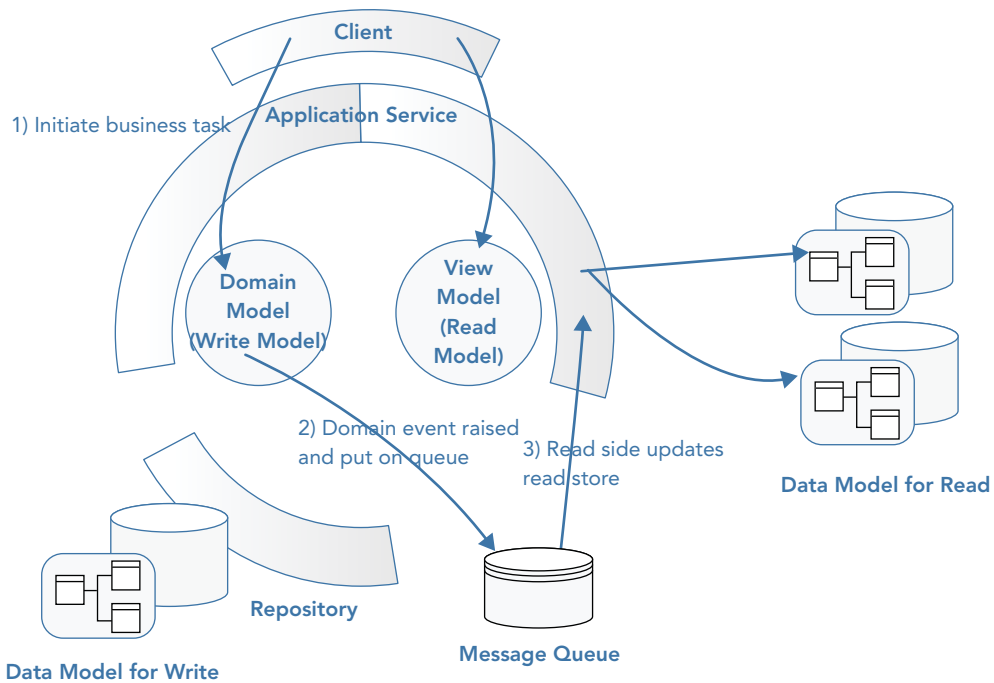


FIGURE 24-7: An eventually-consistent read model.

The Impact to the User Experience

An eventually consistent read store affects your user experience. How up-to-date your reporting data (UI display and traditional reporting) needs to be is a question for your business users. It is worth being explicit about the staleness of the data so users can consider it when making decisions on the data they have in front of them.

Use the Read Model to Consolidate Many Bounded Contexts

You can use a read model to consolidate views from across your enterprise to simplify report rendering. Other bounded contexts that expose reports on the state of their domains via RESTful URLs or messaging systems can be consumed and used within the read store to consolidate the data required to satisfy a composite UI. Figure 24-8 shows how such a system might look.

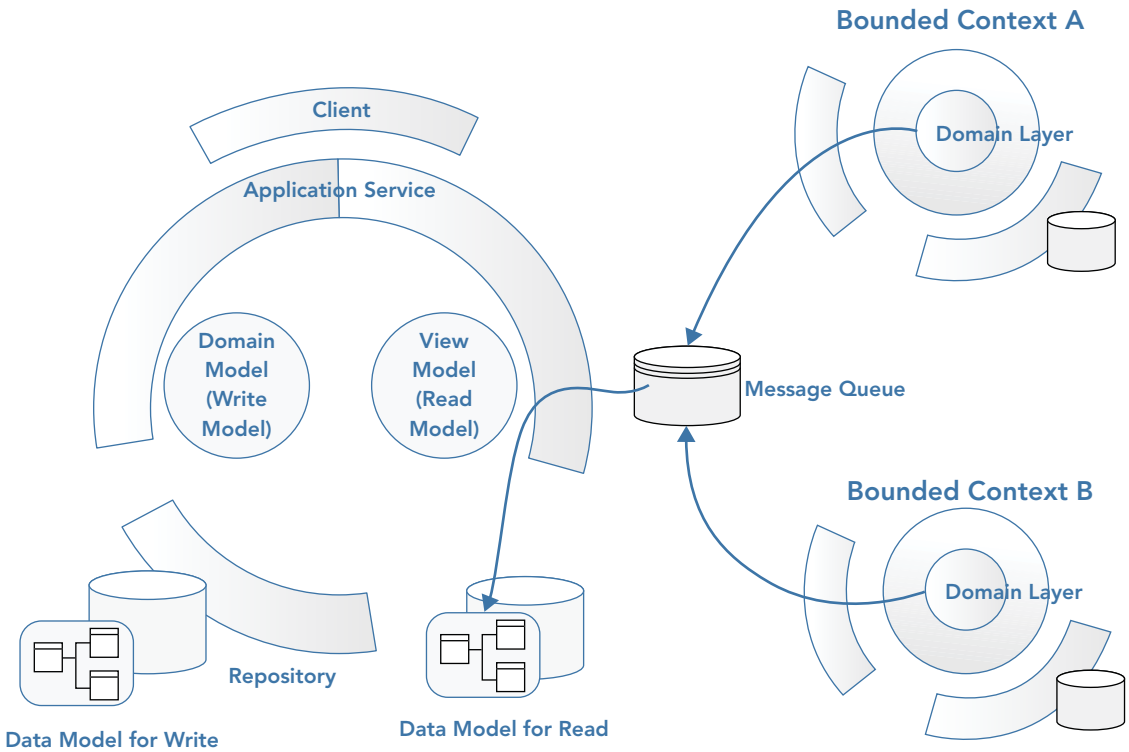


FIGURE 24-8: Consolidate data from many bounded contexts into a single read model.

Using a Reporting Database or a Caching Layer

You may already be using a copy of your transactional database as a reporting database that is replicated through log shipping. This is a form of separating your read and write concerns. If it is applicable, you can use this simple method to scale your read side, as shown in Figure 24-9.

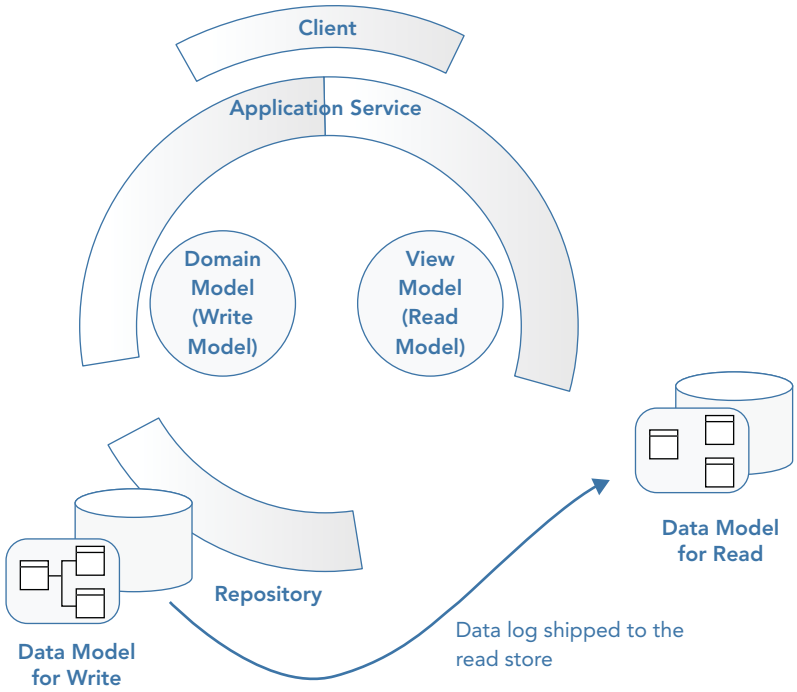


FIGURE 24-9: Use a copy of the transactional database for the read model.

Scaling the Write Side: Using Asynchronous Commands

If you have a highly collaborative domain with many users making changes to the same set of entities, handling business tasks out of process enables you to scale your application to handle the high load. Figure 24-10 shoes how you can use a message queue to store requests for business tasks. The application layer accepts the request from a client to perform a business task, but instead of executing the request straight away, it handles it out of process. The application layer can only acknowledge that the request was received; the client must be notified in a different manner through an e-mail or simply by checking on the status of the request.

Command Validation

If you are sending requests for business tasks asynchronously, you need to be sure, as far as you can, that they will succeed because you have no way of receiving immediate acknowledgement to the success or failure of the request. The application service should perform some basic validation of the request and perhaps even use the view store to check invariants before adding the request to the queue for processing later. An application should do enough validation work to ensure that if a request fails, it is because of business reasons and not because of a missing or incorrect parameter.

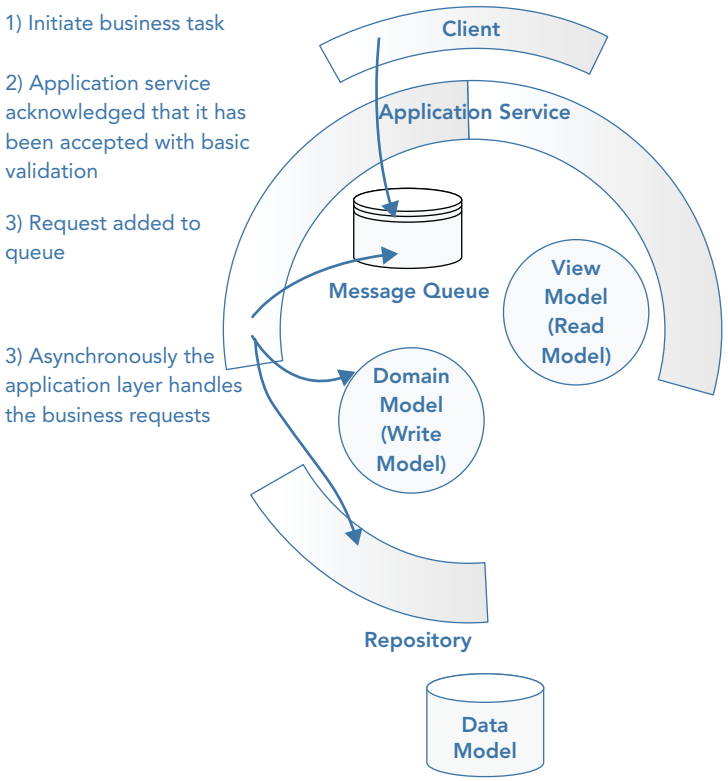


FIGURE 24-10: An asynchronous write side.

Impact to the User Experience

To minimize confusion for users, make it clear that the execution of a request is handled out of process, and an acknowledgement merely confirms that a request was accepted, not that it has succeeded. In some domains, this kind of experience, such as placing an order on an e-commerce site, is normal to a user. As long as the user receives confirmation that a request for an order has been placed and that he has an order ID, he is happy. He is also aware that when it comes to processing the order, his card may fail or some items that were in stock at the time of ordering may now be out of stock and on order for replenishment. If, however, you are in a domain in which users expect to view the changes that their request was intended to make, you need to make it explicit that there will be a delay.

Scaling It All

If you are working in a collaborative domain with heavy reads and writes, you can use an eventually consistent read model in conjunction with business tasks being handled out of process. Figure 24-11 shows you how both sides would look if you needed to scale out the read and write sides of an application.

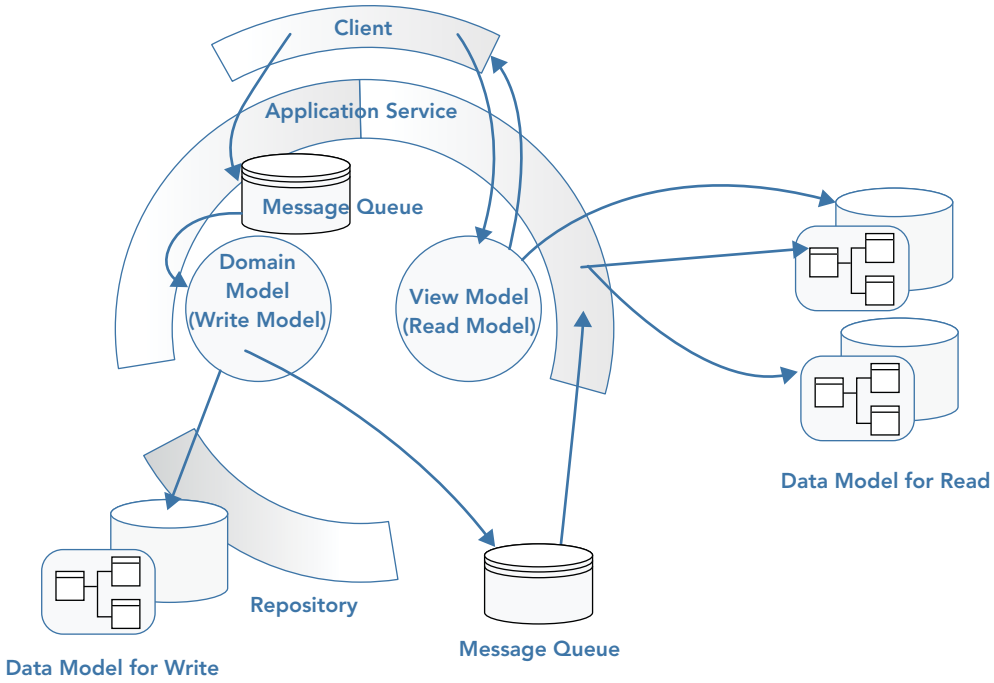


FIGURE 24-11: Scaling out the read and write sides of CQRS.

THE SALIENT POINTS

- CQRS is not an architectural pattern to apply to all bounded contexts of a system.
- Use CQRS if a domain model cannot meet the needs of complex presentation and domain logic without compromise. Split the model into two specific models: one for the read context and one for the write context.
- By using segregation, you can shape aggregates on the write side for behavior and design them around invariants and not for reporting needs.
- View models tailored to reporting needs on the read side can bypass the domain model and pull data directly from the database, which can be tuned for better performance.
- CQRS is a simple pattern. It's a simple case of applying the Single Responsibility Principle at the model level, creating two models instead of one.
- CQRS is often incorrectly thought of as the application of messaging, eventual consistency, domain events, and event sourcing. Although all the above can enhance a system and be useful in specific contexts, they are by no means essential to apply the CQRS pattern.
- CQRS can enable you to scale if you have a high number of reads by allowing you to introduce eventual consistency through separating the read side data model from the write side data model.

- If you are designing a system for a highly collaborative domain with many writes to the same sets of aggregates, you can introduce asynchronous request processing on the write side of the segregation.
- There are trade-offs with scaling and introducing eventual consistency to either the write or the read models. This trade-off must be understood and accepted by the business, and it must be considered in terms of the user experience.
- CQRS enables your system to have limitless scalability on both the read and the write side.

25

Commands: Application Service Patterns for Processing Business Use Cases

WHAT'S IN THIS CHAPTER?

- A discussion of the differences between application logic and domain logic
- Examples of concerns that are handled in the service layer
- Design patterns that can be applied to application services
- Suggestions for testing application services

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 25 download and individually named according to the names throughout the chapter.

Many of Domain-Driven Design (DDD's) benefits arise from disciplined use of a project's ubiquitous language (UL)—both in conversation and in code. One of the big challenges you face, though, is maintaining explicitness of domain concepts in code as you try to keep them isolated from purely technical concerns. For instance, when you are knowledge-crunching with domain experts, talking them through your domain model, it's not ideal to clutter your thinking—or the conversation—with threads, sockets, or database connections. Therefore, to maximize the explicitness of your domain model, a clear separation between real-world domain concepts and purely technical concerns is highly desirable. This

separation is one of the important roles carried out by application services, which belong to the application service layer.

Logically, the application service layer sits above the domain and is dependent upon it. This means that a crucial responsibility of application services is to coordinate with the domain to carry out full business use cases. As part of this responsibility, an application service has to translate inputs and outputs to protect domain structure, and it often needs to communicate with other bounded contexts using REST, messaging, and other concepts discussed in Part II, "Strategic Patterns: Communicating Between Bounded Contexts." Figure 25-1 provides a high-level illustration of the role of application services.

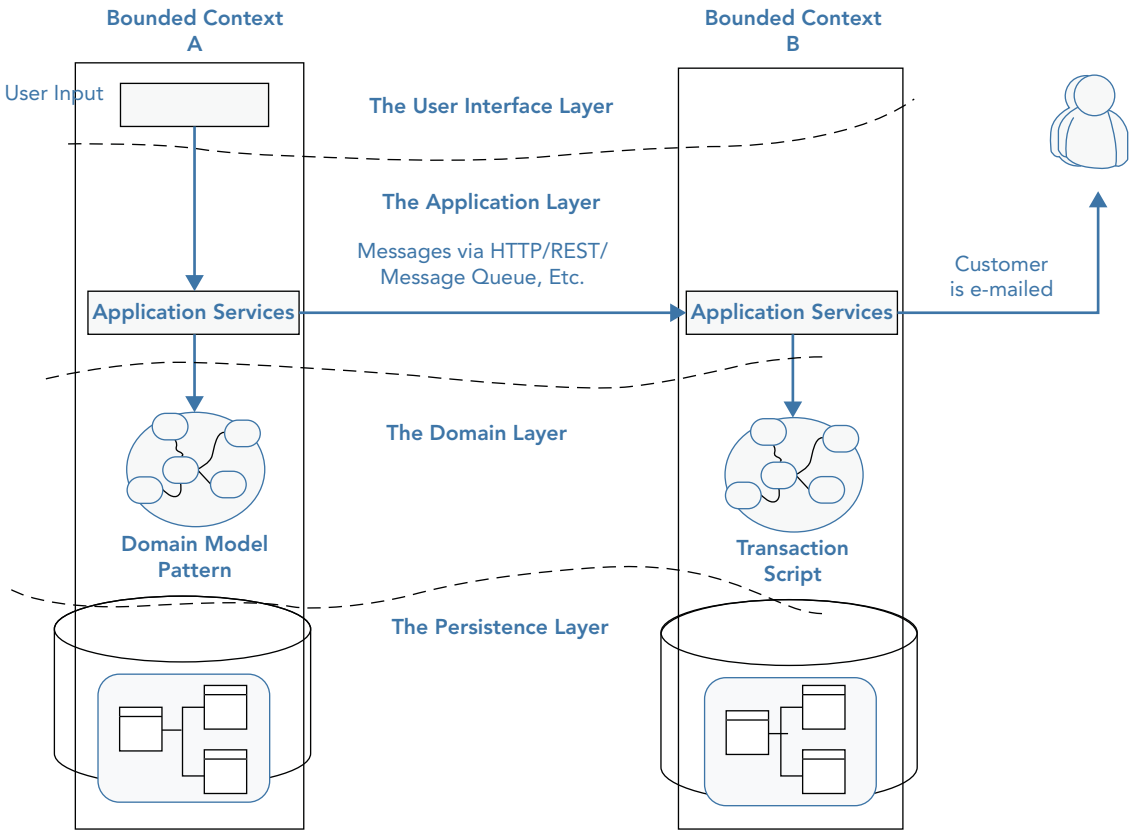


FIGURE 25-1: Where application services fit in.

Figure 25-1 sets the context for this chapter by visualizing how application services mediate between the domain and external services such as other bounded contexts. Before moving on to technical examples that demonstrate the responsibilities of application services, it is important to be clear about the distinction between application and domain logic. Even for experienced DDD practitioners, there are occasionally some challenges with this distinction.

DIFFERENTIATING APPLICATION LOGIC AND DOMAIN LOGIC

Understanding the difference between application logic and domain logic is crucial if you want to produce a model that accentuates domain concepts and isolates them from purely technical details. For the most part, it is not a difficult task, although there is a small gray area. This section aims to provide a clear picture so you rarely have to worry about where your code should live.

Application Logic

As a starting point, you can think of application services as having two general responsibilities. First, they are responsible for infrastructural concerns: managing transactions, sending e-mails, and similar technical tasks. In addition, application services have to coordinate with the domain to carry out full business use cases. Carrying out these responsibilities correctly helps prevent domain logic from being obfuscated or incorrectly located in application services.

To demonstrate the role of application services, this section walks through the creation of an application service that can be used in an online gambling application. It manages the Recommend-a-Friend use case, in which loyal gamblers are rewarded with $50 of free credit if they can entice their friends to sign up. Their friends are also rewarded with a $50 credit when their account is created. As an extra bonus, the business has decided the referrer's loyalty status should be upgraded to gold. Figure 25-2 illustrates the Recommend-a-Friend use case.

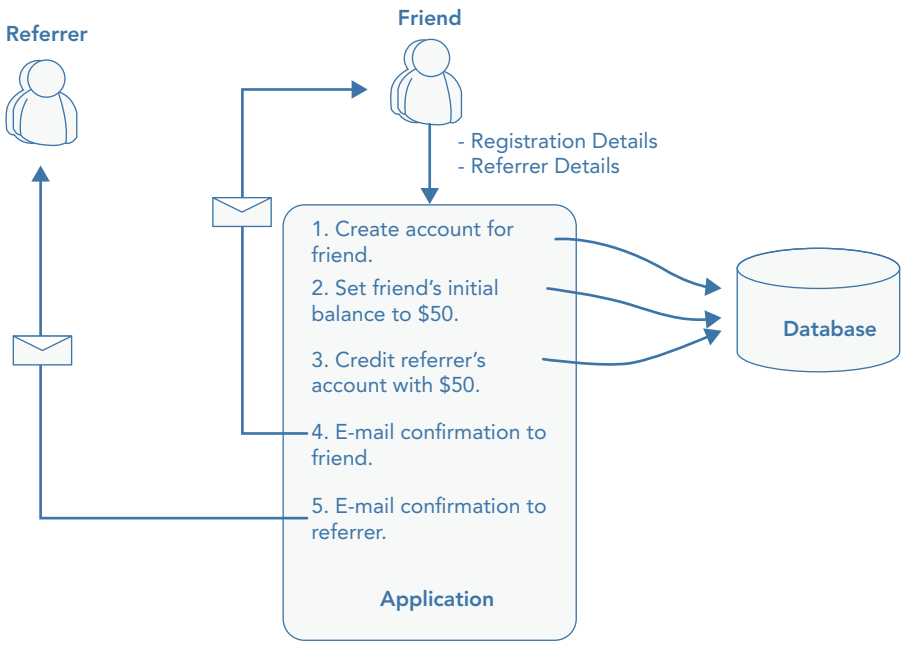


FIGURE 25-2: The Recommend-a-Friend use case.

Infrastructural Concerns

To invoke the functionality of your domain model, you need to carry out a number of infrastructural tasks. Setting up database connections is one common example. When you do a good job of isolating this infrastructural code, you're rewarded with a more maintainable domain model, free from technical clutter. In the following short sections, you see examples of infrastructural concerns being handled in the `RecommendAFriendService`—an application service that is responsible for carrying out the Recommend-a-Friend use case in the online gambling scenario. One of the first tasks this application service has to deal with, like many others, is validating inputs.

> **NOTE** *Adding the "service" suffix to the name of an application service is debatable. This chapter uses the "service" suffix as a learning aid, but it is not essential in your systems.*

Application-Level Validation

Examples of application-level validation include checking that parameters are the correct data type, the correct format, and the correct length. They aren't business rules that domain experts care about, but they can still cause error conditions in a system. Rather than cluttering domain logic with these technical details, you can perform this type of validation in application services.

The `RecommendAFriendService` takes the account ID of the referrer and basic account details of the friend who is signing up. An initial implementation of the `RecommendAFriendService` carrying out application-level validation is shown in Listing 25-1.

LISTING 25-1: RecommendAFriendService Performing Application-Level Validation

```
public class RecommendAFriendService
{
    public void RecommendAFriend(int referrerId, NewAccount friendsAccountDetails)
    {
        Validate(friendsAccountDetails);

        ...
    }

    // technical validation carried out at the application level
    private void Validate(NewAccount account)
    {
        if (!account.Email.Contains("@"))
```

```
            throw new ValidationFailure("Not a valid email address");

        if (account.Email.Length >= 50)
            throw new ValidationFailure(
                "Email address must be less than 50 characters"
            );

        if (account.Nickname.Length >= 25)
            throw new ValidationFailure(
                "Nickname must be less than 25 characters"
            );

        if (String.IsNullOrWhiteSpace(account.Email))
            throw new ValidationFailure("You must supply an email");

        if (String.IsNullOrWhiteSpace(account.Nickname))
            throw new ValidationFailure("You must supply a Nickname");
    }
}

public class NewAccount
{
    public string Email { get; set; }

    public string Nickname { get; set; }

    public int Age { get; set; }
}

public class ValidationFailure : Exception
{
    public ValidationFailure(string message) : base(message) { }
}
```

Listing 25-1 exemplifies the role of application-level validation. Each clause in `Validate()` checks a technical detail such as string length or string format in the case of an e-mail. None of them are a business rule, and none of them belong in the domain.

Transactions

In a typical business use case there are often multiple actions that need to succeed or fail together inside a transaction. By managing transactions in application services, you have full control over which operations that you request of the domain will live inside the same transaction boundary. This can be demonstrated using an updated `RecommendAFriendService`. Imagine the business has decided that if the referral policy cannot be applied, it should not create the new account. Therefore, the transactional boundary encapsulates creating the new account and applying the referral policy to both accounts, as shown in Figure 25-3.

Using the .NET Framework's `TransactionScope`, you can add transactional behavior to the `RecommendAFriendService`, as shown in Listing 25-2. Not all databases, ORMs, and other frameworks use `TransactionScope`, but application programming interfaces (APIs) for creating, committing, and aborting transactions are usually similar.

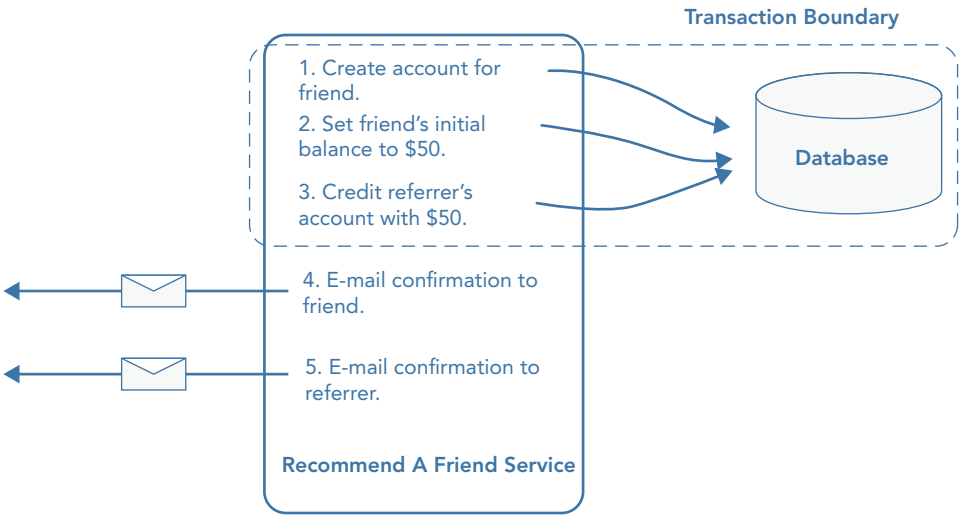


FIGURE 25-3: Transactional boundary for Recommend-a-Friend.

LISTING 25-2: RecommendAFriendService to Handle Transactions

```
public void RecommendAFriend(int referrerId, NewAccount friendsAccountDetails)
{
    Validate(friendsAccountDetails);

    // most technologies have similar transaction APIs
    using (var transaction = new System.Transactions.TransactionScope())
    {
        try
        {
            // ... interact with domain multiple times
            transaction.Complete();
        }
        catch
        {
            // transaction will roll back if Complete() not called
        }
    }
}
```

Error Handling and Error Translation

Not all interactions with the domain will be successful. In such cases, the domain will likely throw exceptions or return error codes. These are cases in which domain validation fails (even though application-level validation was successful). An application service's job is to handle these error conditions and translate them into suitable representations for external parties, so that no external parties are coupled to the structure of the domain errors. External parties could be either human users of a website or other software systems.

One domain error that can occur in the Recommend-a-Friend use case is when the referrer has a long-term outstanding balance. The business has decided that these customers should not

be rewarded with a loyalty bonus. Because the domain will notify of this error condition, the `RecommendAFriendService` is responsible for handling the error and transforming it into a suitable external representation, as shown in Listing 25-3.

LISTING 25-3: RecommendAFriendService Handling and Translating Domain Errors

```
public void RecommendAFriend(int referrerId, NewAccount friendsAccountDetails)
{
    Validate(friendsAccountDetails);

    // most technologies have similar transaction APIs
    using (var transaction = new System.Transactions.TransactionScope())
    {
        try
        {
            // ... interact with domain multiple times
            transaction.Complete();
        }
        catch(ReferralRejectedDueToLongTermOutstandingBalance)
        {
            throw new ApplicationError(
                "Sorry, this referral cannot be completed. The referrer " +
                "currently has an outstanding balance. Please contact " +
                "Customer Services"
            );
            // transaction will roll back if Complete() not called
        }
    }
}
```

> **NOTE** *Technically, a* `TransactionScope` *will roll back the transaction when its* `Dispose()` *method is called if* `Complete()` *has not been called. In Listing 25-3,* `Dispose()` *will be called automatically at the end of the* `using` *block.*

Listing 25-3 shows how application services can protect the domain structure by translating exceptions into a standard format. In this example, the external format is an `ApplicationError` exception. Using an `ApplicationError` exception prevents clients of the application service becoming coupled to the domain exception. Instead, clients are coupled to the `ApplicationError`, which is a more stable interface hiding the potential volatility of the domain.

Using a convention for representing errors, such as an `ApplicationError` exception, gives you the opportunity to handle all errors consistently. For example, in an ASP.NET MVC application, you can create a `HandleErrorAttribute` that handles `ApplicationError`s. When the filter catches an `ApplicationError`, it knows that it was thrown by an application service, meaning that it is safe to present the message to the outside world. In contrast, when it catches any other type of exception, it does not know if it is secure to disclose the details of the error, so it has to return a generic error message, as shown in Listing 25-4.

LISTING 25-4: ASP.NET MVC Error Filter Applying Error Conventions

```
public class ErrorFilter : HandleErrorAttribute
{
    public override void OnException(ExceptionContext filterContext)
    {
        if (filterContext.Exception.GetType() == typeof(ApplicationError))
        {
            // return specific message to user
            var msg = filterContext.Exception.Message;
            ErrorResponse(msg, filterContext);
        }
        else
        {
            // return generic message for security reasons
            var msg = "Sorry. Something really unexpected has occurred";
            ErrorResponse(msg, filterContext);
        }
    }

    public void ErrorResponse(string msg, ExceptionContext ec)
    {
        var routeData = new RouteValueDictionary(new { message = msg });
        var response = new RedirectToRouteResult("ErrorPage", routeData);
        ec.Result = response;
        ec.ExceptionHandled = true;
    }
}
```

Logging, Metrics, and Monitoring

Response times, errors, and other types of diagnostic information allow you to see how your application is performing and spot any potential issues at an early stage. But capturing this information can add unnecessary clutter to your domain logic and obfuscate important concepts. Sometimes you have to suffer this pain, but on many occasions, you can lean on application services to report this information instead.

Listing 25-5 shows an updated version of the `RecommendAFriendService` that logs the duration of each action and whether the overall result is an exception or a success so that the code does not have to be added to the domain model.

LISTING 25-5: RecommendAFriendService Capturing Logs and Metrics

```
public void RecommendAFriend(int referrerId, NewAccount friendsAccountDetails)
{
    Validate(friendsAccountDetails);

    using (var transaction = new System.Transactions.TransactionScope())
    {
        try
        {
```

```
            // ... interact with domain multiple times
            transaction.Complete();

            // log here to avoid cluttering domain
            logger.Debug("Successful friend recommendation");
            StatsdClient.Metrics.Counter("friendReferrals");
        }
        catch (ReferralRejectedDueToLongTermOutstandingBalance ex)
        {
            // log here to avoid cluttering domain
            logger.Error(ex);
            StatsdClient.Metrics.Counter("ReferralRejected");
            throw new ApplicationError(
                "Sorry, this referral cannot be completed. The referrer " +
                "currently has an outstanding balance. Please contact " +
                "customer support"
            );
        }
    }
}
```

The key details to look for in Listing 25-5 are the usages of `logger.Debug()`, `logger.Error()`, and `StatsDClient.Metrics()`. Each of these method calls could have easily ended up mixed in with domain logic. Instead, each has been brought up into the application service.

Authentication and Authorization

A common infrastructural concern that most applications are forced to deal with is authentication. To demonstrate authentication in the Recommend-a-Friend scenario, this example models the case in which customer support is logging in to the admin interface and manually triggering the referral policy because there was a problem when the new customer signed up. Listing 25-6 shows how the `AdminRecommendAFriendService` checks whether the user is authenticated before it applies the referral policy.

LISTING 25-6: AdminRecommendAFriendService Ensuring User Is Authenticated

```
public class AdminRecommendAFriendService
{
    private IAuthenticationService authentication;

    ...

    public void RecommendAFriend(int referrerId, int friendId)
    {
        if (!authentication.IsLoggedInUser())
            throw new AuthenticationError();

        // look up customers

        // apply referral policy
    }
}
```

For many applications, authentication isn't a strong enough security measure, and authorization is additionally required. Authorization is the process that checks whether users have the appropriate privileges to carry out the requested action. Therefore, it's an important tool in applications that have different types of users with different privileges.

Assuming that the `AdminRecommendAFriendService` is part of an application that contains a number of different users—customers and admins—authorization will be required to prevent normal users from applying the referral policy; business would be negatively affected if users could continually top up their online gambling account with free credits by just hacking a few uniform resource locators (URLs). (This does happen.) An upgraded version of the `AdminRecommendAFriendService` that performs an admin authorization check is shown in Listing 25-7.

LISTING 25-7: AdminRecommendAFriendService Ensuring User Is Authorized to Apply Policy

```
public class AdminRecommendAFriendService
{
    private IAuthenticationService authentication;
    private IAuthorizationService authorization;

    // .. constructor

    public void RecommendAFriend(int referrerId, int friendId)
    {
        if (!authentication.IsLoggedInUser())
            throw new AuthenticationError();

        if (!authorization.IsCurrentUserAdmin())
            throw new AuthorizationError();

        // apply referral policy
    }
}
```

Communication

Events that happen inside a domain from one bounded context may trigger events that are handled by other bounded contexts. You saw examples of this in Part II with the messaging and REST examples. It is the responsibility of an application service to transmit events between bounded contexts. This may involve publishing messages using a message bus (see Chapter 12, "Integrating Via Messaging," for detailed examples) or an atom feed (see Chapter 13, "Integrating Via HTTP with RPC and REST," for detailed examples), or other kinds of communication. Importantly, again, the goal is to decouple domain concepts from infrastructural concerns. Listing 25-8 shows an updated version of the `RecommendAFriendService` that also publishes events using NServiceBus.

LISTING 25-8: RecommendAFriendService Publishing Event

```
public class RecommendAFriendService
{
    private ICustomerDirectory customerDirectory;
    private IReferAFriendPolicy referAFriendPolicy;
```

```
        private IBus bus;

        public RecommendAFriendService(ICustomerDirectory customerDirectory,
                           IReferAFriendPolicy referAFriendPolicy, IBus bus)
        {
            this.customerDirectory = customerDirectory;
            this.referAFriendPolicy = referAFriendPolicy;
            this.bus = bus;
        }

        public void RecommendAFriend(int referrerId, NewAccount friendsAccountDetails)
        {
            Validate(friendsAccountDetails);

            using (var transaction = new System.Transactions.TransactionScope())
            {
                try
                {
                    var referrer = customerDirectory.Find(referrerId);
                    var friend = customerDirectory.Add(friendsAccountDetails);
                    referAFriendPolicy.Apply(referrer, friend);

                    transaction.Complete();

                    var msg = new CustomerRegisteredViaReferralPolicy
                    {
                       ReferrerId = referrerId,
                       FriendId = friend.Id
                    };
                    bus.Publish(msg);

                }
                catch (ReferralRejectedDueToLongTermOutstandingBalance)
                {
                    var msg = new ReferralSignupRejected
                    {
                        ReferrerId = referrerId,
                        FriendEmail = friendsAccountDetails.Email,
                        Reason = "Referrer has long term outstanding balance"
                    };
                    bus.Publish(msg);
                }
            }
        }

        private void Validate(NewAccount account)
        {
            ...
        }
    }
```

Although Listing 25-8 shows an application service communicating via NServiceBus, the method of communication could be a remote procedure call (RPC), REST, or another. Application services should also be used to handle other concerns presented in Part II of the book, including handling

external events, handling and publishing internal events (such as the messaging gateway in Chapter 12), and producing an atom feed.

Coordinating Full Business Use Cases

Looking back to the latest version of the `RecommendAFriendService` in Listing 25-8, you can see it handles a lot of infrastructural concerns that do not intrinsically add business value. They are only there to support the second major responsibility of application services that does add value—coordinating the domain model to carry out full business use cases. Listing 25-9 shows a section of an updated version of the `RecommendAFriendService` that builds on the infrastructural foundations by coordinating the domain to carry out the Recommend-a-Friend use case.

LISTING 25-9: RecommendAFriendService Fulfilling Use Case by Interacting with Domain

```
try
{
    // customerDirectory is a domain repository
    Customer referrer = customerDirectory.Find(referrerId);
    Customer friend = customerDirectory.Add(friendsAccountDetails);

    // RecommendAFriendPolicy is a domain policy
    RecommendAFriendPolicy.Apply(referrer, friend);

    transaction.Complete();

    // log here to avoid cluttering domain
    logger.Debug("Successful friend recommendation");
    StatsdClient.Metrics.Counter("friendReferrals");
}
```

You can see the `customerDirectory` and `RecommendAFriendPolicy` variables being used in Listing 25-9. They represent domain objects and are used in this example to exemplify how an application service has to interact multiple times with the domain to carry out the entire use case. Here, it fetches the referrer, asks the domain to create the new customer, and finally tells the domain to apply the policy.

Application Services and Framework Integration

Application services are notorious for becoming bloated with loosely related logic that doesn't appear to belong anywhere else. Sometimes chunks can be refactored into their own classes. In the `RecommendAFriendService`, `Validate()` is a good candidate for moving into its own class. Another source of bloat found in application services is boilerplate duplication. In the `RecommendAFriendService`, the transaction code is typical boilerplate code that could end up being duplicated in every application service. You could create a base application service that follows the Template Method pattern, or you could create a utility that takes a lambda and wraps it with a transaction.

Even with the patterns just mentioned, there is sometimes a cleaner solution than using all-encompassing application services. Many modern frameworks provide hooks for you to inject your infrastructural concerns. ASP.NET MVC is a good example. It has the concept of `ActionFilters`,

which act like a pipeline. Using action filters, you can inject filters that handle validation, opening and closing transactions, logging, and other infrastructural concerns. Listing 25-10 shows an example `TransactionFilter ActionFilter`.

LISTING 25-10: Action Filter for Framework Integration with ASP.NET MVC

```
public class TransactionFilter : ActionFilterAttribute, IActionFilter
{
    // executes before controller
    public override void OnResultExecuting(ResultExecutingContext filterContext)
    {
        // start a transaction
        var t = new TransactionScope();
        HttpContext.Current.Items["transaction"] = t;
        base.OnResultExecuting(filterContext);
    }

    // executes after controller
    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        // close the transaction created at start of this request
        var t = (TransactionScope)HttpContext.Current.Items["transaction"];
        t.Complete();

        base.OnActionExecuted(filterContext);
    }
}
```

When the framework takes care of all your infrastructural concerns, the need for an application service is debatable. Instead, you may be tempted to put your few lines of domain model coordination directly inside a controller, as Listing 25-11 does.

LISTING 25-11: Moving Coordination Logic Directly into the Controller

```
public class RecommendAFriendController : Controller
{
    private ICustomerDirectory directory;
    private IRecommendAFriendPolicy policy;

    public RecommendAFriendController(
        ICustomerDirectory customerDirectory, IRecommendAFriendPolicy policy)
    {
        this.directory = customerDirectory;
        this.policy = policy;
    }

    // all infrastructure concerns handled by framework
    public ActionResult Index(int referrerId, NewAccount friend)
    {
        var referrer = directory.Find(referrerId);
```

continues

LISTING 25-11: *(continued)*

```
            var newAcct = directory.Create(friend);
            policy.Apply(referrer, newAcct);

            return View();
        }
    }
```

As enticing as frameworks are, there is a big gotcha to be aware of. If you are using the application service in multiple places—perhaps a web front end and a desktop front end—then using an all-encompassing application service may prevent you from having to duplicate the infrastructural logic in both front ends. You need to make a per-project decision about how tightly you want to couple yourself to the frameworks you are using.

Domain Logic from an Application Service's Perspective

A challenging activity for many DDD practitioners is drawing the line between application logic and domain logic. Infrastructural concerns are normally easy to identify, but the coordination logic that pieces together full use cases can sometimes appear to contradict the "No business rules in application services" principle. If you look back to Listing 25-9, you can see three interactions with the domain: fetching the referrer, asking the domain to create the new customer, and applying the policy. Some would argue that this logic should live in the domain.

One approach to deciding if a sequence of interactions belongs in the domain is to ask, "Should this always happen?" or "Are these steps inextricable?" If so, that sounds like a domain policy, because those steps always have to happen together. However, if those steps can be recombined in a number of ways, potentially it's not a domain concept. In the Recommend-a-Friend use case, you might argue that the referral policy should only ever be applied to an existing customer (the referrer) and a new customer (the referrer's friend). It follows, then, that the logic in the application service is actually a business rule that may be best encapsulated inside a domain service. However, sometimes the business wants to apply the policy to two existing accounts if there was an error during creation. In that case, the creation of the new account and the policy aren't as coupled, suggesting this is application logic.

A sign that you have avoided leaking domain concepts into application services is that each interaction with the domain is expressive. An example of this is `RecommendAFriendPolicy.Apply()`. The implementation of this method credits each account with $50 and promotes the referrer to gold loyalty status. If those steps lived in the application service, it would be less expressive, more verbose, and a big sign that domain concepts had leaked out of the domain.

APPLICATION SERVICE PATTERNS

You can use a variety of design patterns and principles inside application services based on your preferences and based on context. In the `RecommendAFriendService` from the previous section, you saw vanilla, script-like, object-oriented code. In many cases, that is the simplest possible

solution and a good choice. On other occasions, you may find it presents maintenance headaches or undesirable coupling. The following patterns have arisen within the community for addressing concerns like these.

Command Processor

Using the command processor pattern, you can avoid developing large application services with many concerns. Instead, you have a command and a processor for each use case. Listing 25-12 shows a monolithic application service that can be considered to have multiple responsibilities.

LISTING 25-12: Potentially Monolithic Application Service

```
public class BloatedRecommendAFriendService
{
    public void RecommendAFriend(int referrerId, NewAccount friend)
    {
        ...
    }

    public void RecommendAFriendInDifferentCountry(int referrerId,
                                                   NewAccount friend)
    {
        ...
    }

    public void ReverseFriendReferral(int referrerId, int friendId)
    {
        ...
    }

    public void ReferAFriendWithoutLoyaltyBonus(int referrerId, NewAccount friend)
    {
        ...
    }

    // more methods like this
}
```

Friction can occur when you have application services like the `BloatedRecommendAFriendService` in Listing 25-12. The implementations for each method may vary considerably, with each using different dependencies. Low cohesion is especially common in applications that use CQRS. Consequently, the application service can grow into a source of development friction. The command processor pattern can be used to alleviate these pains.

To create an alternative version of the `BloatedRecommendAFriendService` using the command processor pattern, the first step is to create a command that expresses intent and contains all the relevant information needed for it to be carried out. You can see an example of this in Listing 25-13.

LISTING 25-13: RecommendAFriend Command

```
// command expressing intent
public class RecommendAFriend
{
    public int ReferrerId { get; set; }

    public NewAccount Friend { get; set; }
}
```

After creating the command, you then create a command processor. An interface for a command processor that processes `RecommendAFriend` commands is shown in Listing 25-14.

LISTING 25-14: RecommendAFriendProcessor Command Processor Interface

```
public interface IRecommendAFriendProcessor
{
    void Process(RecommendAFriend command);
}
```

Creating a command and processor as above is fundamentally just a case of creating a separate interface for each use case in the application, with the aim of isolating responsibilities and increasing expressiveness. But many DDD practitioners have taken the pattern further by adding a layer of indirection that provides looser coupling and the ability to chain command processors. Chaining can be used to create a pipeline that carries out infrastructural concerns such as validation, transactions, and logging, allowing you to isolate domain coordination. This version of the pattern requires common processor interfaces, as shown in Listing 25-15.

LISTING 25-15: Common Command Processor Interface

```
public interface ICommandProcessor<T>
{
    void Process(T command);
}
```

Each command processor then implements this interface so they can be chained together. First, you want to create a handler that is specific to the use case you are implementing. In this case, that would be a `RecommendAFriendProcessor`, as shown in Listing 25-16.

LISTING 25-16: Use Case-Specific Command Processor

```
public class RecommendAFriendProcessor : ICommandProcessor<RecommendAFriend>
{
    public void Process(RecommendAFriend command)
    {
        Console.WriteLine("Processing RecommendAFriend command");
    }
}
```

Generic command processors that you can apply to any use case also need to implement the `ICommandProcessor` interface. You can see demonstrative logging and transaction processors in Listing 25-17.

LISTING 25-17: Generic Processors That Can Be Applied to Any Use Case

```
public class LoggingProcessor<T> : ICommandProcessor<T>
{
    private ICommandProcessor<T> nextLinkInChain;

    public LoggingProcessor(ICommandProcessor<T> processor)
    {
        this.nextLinkInChain = processor;
    }

    public void Process(T command)
    {
        // log something before
        nextLinkInChain.Process(command);
        // log something after
    }
}

public class TransactionProcessor<T> : ICommandProcessor<T>
{
    private ICommandProcessor<T> nextLinkInChain;

    public TransactionProcessor(ICommandProcessor<T> processor)
    {
        this.nextLinkInChain = processor;
    }

    public void Process(T command)
    {
        // start transaction
        try
        {
            nextLinkInChain.Process(command);
            // commit transaction
        }
        catch
        {
            // roll back transaction
        }
    }
}
```

When you're looking at Listing 25-17, the most important detail to discern is that each command processor takes another processor in its constructor. When a processor processes a command, it invokes the processor passed into its constructor. Essentially, it is wrapping (or decorating) the "child" processor to create a pipeline that can be infinitely long. Another important detail to be aware of occurs inside `Process()`. A processor can perform logic before and after the processor it

wraps. As you can see with the `TransactionProcessor`, it starts a transaction, invokes the child (which may invoke other children), and then commits or rolls back the transaction depending on whether the wrapped processor threw an exception.

> **NOTE** *The* `TransactionProcessor` *in Listing 25-17 contains* `try/catch` *logic. In your applications, you may want to move this into a separate error-handling processor.*

Wiring up command processors is the last remaining detail. You can see the simplest possible version of this in Listing 25-18. If you compare the code with Figure 25-17, it should help you see how the pipeline is being created.

LISTING 25-18: Wiring Up Command Processors to Form a Pipeline

```
public static class Bootstrap
{
    public static ICommandProcessor<RecommendAFriend>
        RecommendAFriendProcessor { get; set; }

    public static void ConfigureApplication()
    {
        // create inner processor
        var RecommendAFriendProcessor = new RecommendAFriendProcessor();

        // wrap inner processor with logging
        var loggingProcessor =
        new LoggingProcessor<RecommendAFriend>(RecommendAFriendProcessor);

        // wrap logging processor (that wraps inner) with a transaction
        var transactionProcessor =
        new TransactionProcessor<RecommendAFriend>(loggingProcessor);

        RecommendAFriendProcessor = transactionProcessor;

        // alternatively, you can use dependency injection
    }
}
```

There are other ways of wiring up pipelines than that shown in Listing 25-18. Some teams do prefer to manually wire up each pipeline in a similar fashion to this using a few helper methods to avoid duplication. Other teams prefer to use dependency injection. It's completely up to you to decide how to configure your pipelines.

Publish/Subscribe

A pattern for looser coupling is publish/subscribe, whereby application services subscribe to events in the domain. You may want to consider this pattern when your domain logic is inherently event based, especially when you pass commands into the domain but do not receive a return value. An interface for an event-based version of the `ReferralPolicy` is shown in Listing 25-19.

LISTING 25-19: Event-Based Interface for the ReferralPolicy

```
public interface IReferralPolicy
{
    event EventHandler<Referral> ReferralAccepted;

    event EventHandler<Referral> ReferralRejected;

    void Apply(RecommendAFriend command);
}
```

To use the `IRefferalPolicy` in Listing 25-19, an application service passes a command into `Apply()` as usual. However, to learn whether the command was successful, the application service has to subscribe to the two events: `ReferralAccepted` and `ReferralRejected`. This pattern is illustrated in Listing 25-20.

LISTING 25-20: Subscribing to Events on a Domain Model

```
public class RecommendAFriendService
{
    private IReferralPolicy policy;

    public RecommendAFriendService(Domain.IReferralPolicy policy)
    {
        // subscribe to events on domain model
        policy.ReferralAccepted += HandleReferralAccepted;
        policy.ReferralRejected += HandleReferralRejected;

        this.policy = policy;
    }

    private void HandleReferralAccepted(object sender, Domain.Referral e)
    {
        // send confirmation e-mails, etc.
    }

    private void HandleReferralRejected(object sender, Domain.Referral e)
    {
        // send rejection e-mails, etc.
    }

    public void RecommendAFriend(int referrerId, NewAccount friend)
    {
        var command = new RecommendAFriend
        {
            ReferrerId = referrerId,
            Friend = friend
        };
        policy.Apply(command);
    }
}
```

Subscription to the `IReferralPolicy`'s events occurs in the `RecommendAFriendService`'s constructor shown in Listing 25-20. Anytime the domain model fires either of those events, the appropriate handler is called: `HandleReferralAccepted()` or `HandleReferralRejected()`. These events are triggered by passing commands into the domain model, as occurs inside `RecommendAFriend()`.

A problem with the code in Listing 25-20 is that it cannot handle transactions properly because the event handlers do not have access to the transaction object to commit or roll back. A solution to this problem is to have an instance field as a transaction. For this to work, though, you have to ensure that a new instance of the `RecommendAFriendService` is created for each new transaction to avoid multithreading issues.

Another transaction-related problem with the code in Listing 25-20 occurs in multithreading scenarios. Consider the case in which the command is applied asynchronously in another thread using a C# task:

```
Task.Factory.StartNew(() => policy.Apply(command));
```

The transaction scope wrapping this call is no longer in scope when `Apply()` is called asynchronously. Fortunately, .NET's `TransactionScope` has modes that help in async scenarios (`http://stackoverflow.com/questions/13543254/get-transactionscope-to-work-with-async-await`). When using transactions and threads like this, you still need to be careful. You may also want to consider using `async/await`.

Request/Reply Pattern

Before choosing patterns with a higher complexity trade-off, it's always worth considering the simplicity of the request/reply pattern. This pattern follows the One Model In One Model Out Approach (OMIOMO), where a data transfer object (DTO) is passed into the application service and a DTO is returned, as Listing 25-21 demonstrates.

LISTING 25-21: An Application Service Based on the Request/Reply Pattern

```
public class RecommendAFriendService
{
    private IReferralPolicy policy;

    public RecommendAFriendService(Domain.IReferralPolicy policy)
    {
        this.policy = policy;
    }

    public RecommendAFriendResponse RecommendAFriend(
        RecommendAFriendRequest request)
    {
        try
        {
            var command = new RecommendAFriend
            {
```

```
                ReferrerId = request.ReferrerId,
                Friend = request.Friend
            };

            policy.Apply(command);

            return new RecommendAFriendResponse
            {
                Status = RecommendAFriendStatus.Success
            };
        }
        catch (ReferralRejectedDueToLongTermOutstandingBalance)
        {
            return new RecommendAFriendResponse
            {
                Status = RecommendAFriendStatus.ReferralRejected
            };
        }
    }
}
```

A common convention is to suffix the type of the input model with "request" and the output model with "response." Though this is an optional aspect of the pattern, both objects should be simple DTOs that do not contain domain objects, and both should reside within the application services layer. The `RecommendAFriendRequest` DTO in Listing 25-22 and the `RecommendAFriendResponse` DTO in Listing 25-23 exemplify these characteristics.

LISTING 25-22: A DTO Request Model

```
public class RecommendAFriendRequest
{
    public int ReferrerId { get; set; }

    public NewAccount Friend { get; set; }
}
```

LISTING 25-23: A DTO Response Model

```
public class RecommendAFriendResponse
{
    public RecommendAFriendStatus Status { get; set; }
}

public enum RecommendAFriendStatus
{
    Success,
    ReferralRejected
}
```

Another common convention when using request/reply is for the response object to contain the status of the use case. When an error occurs or a policy is rejected, for example, instead of the application service throwing an exception, it will set the appropriate status on the response object.

Overall, request/reply is a simplistic pattern that relies on procedural code. If you don't need the benefits of other complex patterns, it's best to keep things simple and make life easier for other developers by using request/reply until it starts to cause friction.

async/await

C# developers can write single-threaded-like code that has the benefit of being asynchronous thanks to the `async` and `await` keywords added in C# 5. You should definitely consider them as an option if you want to build asynchronous, nonblocking applications. This pattern can conflict with your needs for a clear and expressive domain model, though, due to asynchronous methods requiring a return type of `Task<T>`. Listing 25-24 demonstrates this problem.

LISTING 25-24: async/await Compatible Repository

```
public interface ICustomerDirectory
{
    Task<Customer> Find(int customerId);

    Task<Customer> Create(NewAccount details);
}
```

In Listing 25-24, you can see the customer directory returns `Task<Customer>`. This is so that an implementation can use nonblocking database calls that are more thread efficient and scalable. Clearly, this pollutes the domain a little. The conciseness of `async` and `await` does mitigate the syntactic noise, as the `RecommendAFriendService` in Listing 25-25 shows.

LISTING 25-25: Async/await Compatible RecommendAFriendService

```
public async void RecommendAFriend(int referrerId, NewAccount friend)
{
    // ...
    var referrer = await directory.Find(referrerId);
    var newAcct = await directory.Create(friend);
    policy.Apply(referrer, newAcct);
    // ...
}
```

One benefit of making the three calls in `RecommendAFriend()` and not the domain is that the technical details of `async/await` do not clutter the domain logic, as Listing 25-25 illustrates. Also note in Listing 25-25 that `RecommendAFriend()` is marked with the `async` keyword. This ensures that when this method is called, it is executed asynchronously if it contains asynchronous calls. Both of the first two lines of code are in fact asynchronous calls; the `await` keyword signifies

this. When the .NET runtime reaches the `await` keyword, it tries to avoid blocking threads waiting for the invocation to complete. Again, with this solution, you need to take a bit of extra care when working with transactions (`http://stackoverflow.com/questions/13543254/get-transactionscope-to-work-with-async-await`).

You have to weigh the pros and cons when using `async` and `await`. Their use can be more resource efficient, but they also add noise to your code. Fortunately, in Listing 25-25, most of the noise was confined to the application service and not the domain. This might be a compromise that you can aim for in your own designs, where possible.

> **NOTE** *You can read about the asynchronous* `async/await`*-compatible methods in ADO.NET on MSDN (*`http://blogs.msdn.com/b/adonet/archive/2012/07/15/using-sqldatareader-s-new-async-methods-in-net-4-5-beta-part-2-examples.aspx`*). And you can learn about the state-machine-based implementation of* `async/await` *on Jon Skeet's blog (*`http://msmvps.com/blogs/jon_skeet/archive/2011/05/08/eduasync-part-1-introduction.aspx`*).*

TESTING APPLICATION SERVICES

You can cover large vertical slices of functionality or features by testing at the application service level. Ideally, the aim is to test as much as possible so that the environment is as live-like as possible. These types of tests can fall into a variety of groups, including system tests, acceptance tests, integration tests, and functional tests.

> **NOTE** *System, acceptance, and functional tests can also be carried out at a higher level on a running instance of the application. Common examples are Hypertext Transport Protocol (HTTP) response testing (for APIs) and automated browser testing for web pages.*

Use Domain Terminology

Tests can be a fantastic opportunity to express domain concepts by writing them in the UL. You may want to sit down with domain experts to create tests based on their acceptance criteria using Behavior Driven Development (BDD). This gives you the opportunity to verify with the domain expert(s) that the wording of your tests aligns precisely with domain concepts.

A high-level test outline for the Recommend-a-Friend use case testing at the application service level is shown in Listing 25-26. Notice how the names of the tests express domain concepts.

LISTING 25-26: Outline for a High-Level Test Expressed in the Project's UL

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Tests
{
    [TestClass]
    // class named after domain concept/use case
    public class Recommend_a_friend
    {
        // this will run first (once only) then each test will run
        [ClassInitialize]
        public void When_a_user_signs_up_with_a_referral_from_a_friend()
        {
            ...
        }

        [TestMethod]
        public void The_referrer_has_50_dollars_credited_to_their_account()
        {
            ...
        }

        [TestMethod]
        public void The_friend_has_an_account_created_with_an_initial_50_dollars()
        {
            ...
        }

        [TestMethod]
        public void The_referrers_loyalty_is_upgraded_to_gold_status()
        {
            ...
        }

        [TestMethod]
        public void The_referrer_gets_an_email_notifying_of_the_referral()
        {
            ...
        }

        [TestMethod]
        public void The_friend_gets_an_email_notifying_of_account_creation()
        {
            ...
        }
    }
}
```

Test as Much Functionality as Possible

Testing as much as possible increases your confidence that everything will work together when the application is deployed. To do this, you need to avoid using mocks and stubs, preferring to

use concrete implementations instead. Listing 25-27 shows how this applies to the high-level `Recommend _ a _ friend` test.

LISTING 25-27: Testing Against Real Implementations Where Possible

```
[ClassInitialize]
public void When_a_user_signs_up_with_a_referral_from_a_friend()
{
    // test as much of the implementation as possible
    directory = new CustomerDirectory(new InMemoryDatabase());
    var policy = new ReferralPolicy();

    // cannot test e-mailing implementation - easier to stub
    emailer = MockRepository.GenerateStub<IEmailer>();

    service = new RecommendAFriendService(directory, policy, emailer);

    service.RecommendAFriend(referrerId, friendsDetails);
}
```

In Listing 25-27, the `RecommendAFriendService` is being constructed with a concrete implementation of the `CustomerDirectory`. This is the same implementation to be used when the application is deployed. You can see how this test will validate that the `CustomerDirectory` repository and `RecommendAFriendService` application service are integrating as required.

You will also notice in Listing 25-27 that the `CustomerDirectory` is constructed with an in-memory database. This is because it is not possible to test against a real database in a test easily and quickly. Instead, the database is replaced with an in-memory version that is easy to set up and fast to test against. This isn't completely live-like, though, so there's still a small chance the test may pass when something database related may fail at run time. Adding a few full end-to-end tests to your suite that do hit a real database can improve your confidence.

Another component that cannot be tested easily and quickly is the `emailer`, so it is stubbed using RhinoMocks. This means that this component is not being covered; this test will succeed as long as methods on the `emailer` are called with the correct arguments, as shown in Listing 25-28.

LISTING 25-28: Verifying Stub Invocations When Implementations Cannot Be Tested

```
[TestMethod]
public void The_refferer_gets_an_email_notifying_of_the_referral()
{
    var referrer = directory.Find(referrerId);
    emailer.AssertWasCalled(em =>
    {
        em.SendReferralAcknowledgement(referrer);
    });
}
```

RhinoMocks provides a method called `AssertWasCalled()` that throws an exception if the passed-in lambda was not invoked during the test run. In Listing 25-28, therefore, RhinoMocks throws an

exception if the `emailer`'s `SendReferralAcknowledgement()` was not called with an argument that represented the referrer. As mentioned, though, mocks and stubs are often a last resort, when you cannot test the full functionality. When you can test the full functionality, you can directly test the expected outcome, as shown in Listing 25-29.

LISTING 25-29: Testing Full Outcome Without Mocks and Stubs

```
[TestMethod]
public void The_referrers_loyalty_is_upgraded_to_gold_status()
{
    var referrer = directory.Find(referrerId);
    Assert.AreEqual(LoyaltyStatus.Gold, referrer.LoyaltyStatus);
}
```

THE SALIENT POINTS

- Application services and a service layer allow you to isolate technical concerns from domain logic.
- Technical concerns include transactions, database connections, and e-mails.
- Application services are responsible for coordinating with the domain to carry out full business use cases.
- When communicating with the domain, application services should invoke expressive high-level APIs on domain objects.
- An important responsibility of application services is to protect domain structure by presenting higher layers and external components with more stable interfaces to couple themselves to.
- You can use design patterns like the command processor pattern and asynchronous patterns in the service layer.
- Testing application services is an opportunity to express high-level behaviors or full business use cases, in the ubiquitous language, while covering a high percentage of the implementation.

26

Queries: Domain Reporting

WHAT'S IN THIS CHAPTER?

- Guidance on building reports that aren't overly coupled to domain structure
- Building reports using existing domain services
- Building reports that bypass the domain and hit the database directly
- Building reports in applications that use event sourcing
- A discussion on the trade-offs involved when choosing between reports that belong to a bounded context and reports that need to integrate data from multiple bounded contexts

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 26 download and individually named according to the names throughout the chapter.

Software systems are built to support the needs of the business. Not only do these needs include revenue-generating functionality, but they also include the ability to assess how well the business is performing. This is the role of reports: to track important metrics and key performance indicators (KPIs) like sales, financial targets, and customer satisfaction. As you've seen so far in this book, there are many ways to build a system when applying Domain-Driven Design (DDD). Equally, there are many ways to implement reporting.

Choosing how to implement reporting in your applications involves considering familiar trade-offs: speed of development, maintainability, performance, and even scalability. Sometimes you can simply create a new web page that reuses all your existing code. Other times, when you have distributed bounded contexts, you may need to create an entirely separate reporting bounded context that subscribes to events from many other bounded contexts and stores all the information locally. This chapter aims to present you with a variety of options to make you familiar with the trade-offs and better equip you to make diligent decisions on the projects you are involved in.

> **NOTE** *In the context of this chapter, a report is defined as the presentation of a coherent set of data for some analytical purpose. In reality, this just means that a report takes on the generic role of information presented to users, rather than the formal notion of a report that is used specifically by managers or for business information (BI) purposes. As an example, the summary of a twitter profile shown on a public-facing twitter profile page—the number of followers, tweets, and retweets—would fall into this chapter's broad definition of a report.*

DOMAIN REPORTING WITHIN A BOUNDED CONTEXT

Having all your data collocated inside the same bounded context is the least difficult reporting scenario. It's also an indication that you have correctly identified the boundaries for your bounded contexts, because reports are usually intended for a specific department within a business, which should map onto a single bounded context. One of the biggest decisions you have to make when you don't have to worry about distribution is whether to use your domain code when generating reports.

Deriving Reports from Domain Objects

To build a web page displaying reports, you can use existing domain code to build a view model. If you need to build a page quickly, this is usually the first option to consider, because it can require the least amount of effort. The big trade-off is performance, because you have less control over the query that is made to your datastore. If performance isn't much of an issue but development speed is, this can be the perfect choice.

In the following examples, you learn how to create a Dealership Performance Report, which an automotive franchise uses to track the performance of its dealerships. First, you see an example using basic mappings, and then you see an alternative implementation using the mediator design pattern.

Using Simple Mappings

Probably the quickest way to build a report is to take domain objects and map their properties onto a view model that provides data and presentation logic required to build the view. Given the domain objects shown in Listing 26-1, which are already used in other parts of the application, you can easily populate the view model shown in Listing 26-2 by simply mapping across the properties shown in Listing 26-3.

LISTING 26-1: Domain Entities That Provide Data for the Dealership Performance Report

```
public interface IDealershipRepository
{
    Dealership Get(int dealershipId);
}

public interface IDealershipRevenueCalculator
{
    DealershipPerformanceActuals CalculateFor(Dealership dealership,
                                    DateTime start, DateTime end);
}

public interface IDealershipPerformanceTargetsProvider
{
    DealershipPerformanceTargets Get(Dealership dealership,
                                   DateTime start, DateTime end);
}

public class DealershipPerformanceTargets
{
    public int TargetRevenue { get; set; }

    public int TargetProfit  { get; set; }
}

public class DealershipPerformanceActuals
{
    public int TotalRevenue { get; set; }

    public int NetProfit { get; set; }
}

public class Dealership
{
    public int Id { get; set; }

    public string Name { get; set; }
}
```

LISTING 26-2: View Model Used to Build the Dealership Performance Report View

```
public class DealershipPerformanceReport
{
    public DateTime ReportStartDate { get; set; }

    public DateTime ReportEndDate { get; set; }

    public List<DealershipPerformanceStatus> Dealerships { get; set; }
}

public class DealershipPerformanceStatus
```

continues

LISTING 26-2 *(continued)*

```
{
    public string DealershipName { get; set; }

    public int TotalRevenue { get; set; }

    public int TargetRevenue { get; set; }

    public int NetProfit { get; set; }

    public int TargetProfit { get; set; }
}
```

LISTING 26-3: Mapping from Domain Objects to View Model

```
var statuses = new List<DealershipPerformanceStatus>();
foreach (var id in dealershipIds)
{
    // select N+1 - potentially bad for performance and efficiency
    // reusing existing domain code - quick to implement
    var dealership = repository.Get(id);
    var targets = provider.Get(dealership, start, end);
    var actuals = calculator.CalculateFor(dealership, start, end);

    // map from domain to view model so user interface (UI) is
    // not coupled to domain objects
    // could move this logic into a separate mapper
    statuses.Add(new DealershipPerformanceStatus
    {
        DealershipName = dealership.Name,
        TotalRevenue = actuals.TotalRevenue,
        TargetRevenue = targets.TargetRevenue,
        NetProfit = actuals.NetProfit,
        TargetProfit = targets.TargetProfit
    });
}

// mapping logic could live in a dedicated mapper
var viewModel = new DealershipPerformanceReport
{
    ReportStartDate = start,
    ReportEndDate = end,
    Dealerships = statuses
};
```

NOTE *You can download full code examples for this chapter at* `www.wrox.com/go/domaindrivendesign` *on the Download Code tab.*

As you can see from the code in Listings 26-1 to 26-3, you can easily create the report by using existing domain objects. One of the biggest decisions you need to make is where to locate your mapping logic. You can put it directly in application services or controllers, you can create dedicated mapper classes, or you can do the mapping inside the view model in the constructor or via a static factory method. Listing 26-4 shows how this solution looks using an application service called DealershipReportBuilder to build the report.

LISTING 26-4: An Application Service That Builds a Report and Performs All the Mapping

```
public class DealershipPerformanceReportBuilder
{
    private IDealershipRepository repository;
    private IDealershipRevenueCalculator calculator;
    private IDealershipPerformanceTargetsProvider provider;

    public DealershipPerformanceReportBuilder(IDealershipRepository repository,
                                      IDealershipRevenueCalculator calculator,
                                IDealershipPerformanceTargetsProvider provider)
    {
        this.repository = repository;
        this.calculator = calculator;
        this.provider = provider;
    }

    public DealershipPerformanceReport BuildReport(IEnumerable<int> dealershipIds,
                                                   DateTime start, DateTime end)
    {
        var statuses = BuildStatuses(dealershipIds, start, end);

        return new DealershipPerformanceReport
        {
            ReportStartDate = start,
            ReportEndDate = end,
            Dealerships = statuses
        };
    }

    private List<DealershipPerformanceStatus> BuildStatuses(
        IEnumerable<int> dealershipIds, DateTime start, DateTime end)
    {
        var statuses = new List<DealershipPerformanceStatus>();
        foreach (var id in dealershipIds)
        {
            var dealership = repository.Get(id);
            var targets = provider.Get(dealership, start, end);
            var actuals = calculator.CalculateFor(dealership, start, end);

            statuses.Add(new DealershipPerformanceStatus
            {
```

continues

LISTING 26-4 *(continued)*

```
                DealershipName = dealership.Name,
                TotalRevenue = actuals.TotalRevenue,
                TargetRevenue = targets.TargetRevenue,
                NetProfit = actuals.NetProfit,
                TargetProfit = targets.TargetProfit
            });
        }
        return statuses;
    }
}
```

Unfortunately, the benefits do come with a cost. Reusing the domain objects from Listing 26-2 made light work of creating the report. However, the logic to build the view model in Listing 26-4 carries a potentially big performance hit because of it. For each dealership ID, the dealership and its performance details are retrieved. There can be up to three object-relational mapper (ORM)-generated database queries. In a system that has ten dealerships, that can be thirty database queries, which may have severe consequences under peak load. Performance isn't always critical, but in reporting scenarios like this, it's important to have an idea of how much you are giving away and how that might hurt you in production—especially when ORMs with features like lazy-loading are involved.

Another problem with using mappings is that you may need to expose additional properties on domain objects that you probably prefer to keep private and internal to the domain. Subsequently, this increases the opportunity for the service layer to be coupled to domain structure. To reduce unwanted coupling between the service layer and the domain, you may want to consider other patterns, like mediator.

Using the Mediator Pattern

To create a view model that contains all the relevant information for a report but isn't overly coupled to domain structure, you can use the mediator design pattern. Using the mediator, you instead pass your view model into a mediator, which itself is passed into the domain. The domain objects then interact with the mediator, which updates the view model accordingly. This doesn't break layering, because the mediator implements an interface that belongs in the domain.

As part of an alternative implementation of the dealership performance report, Listing 26-5 shows the mediator interface along with modified domain objects that no longer expose their structure, but instead provide a method that accepts and interacts with a mediator.

LISTING 26-5: A Mediator Interface and Domain Objects That Use It

```
// mediator interface - stable domain structure that can be
// exposed to the application service layer
public interface IDealershipAssessment
{
```

```
    int TotalRevenue { get; set; }

    int TargetRevenue { get; set; }

    int NetProfit { get; set; }

    int TargetProfit { get; set; }
}

public class DealershipPerformanceTargets
{
    // private fields hide potentially volatile domain structure
    private int targetRevenue;
    private int targetProfit;

    public void Populate(IDealershipAssessment mediator)
    {
        mediator.TargetRevenue = targetRevenue;
        mediator.TargetProfit = targetProfit;
    }
}

public class DealershipPerformanceActuals
{
    // private fields hide potentially volatile domain structure
    private int totalRevenue;
    private int netProfit;

    public void Populate(IDealershipAssessment mediator)
    {
        mediator.TotalRevenue = totalRevenue;
        mediator.NetProfit = netProfit;
    }
}
```

`IDealershipAssessment` is the mediator interface in Listing 26-5. Whenever a concrete implementation of the mediator is passed into the `DealershipPerformanceTargets` or `DealershipPerformanceActuals`'s `Populate()`, fields are set on the mediator using the values from private instance variables. The benefit to doing this is that those private variables are not exposed outside the domain. Without the coupling, they are free to change. This is in direct contrast to the previous example. The implementation of the mediator shown in Listing 26-6 attempts to clarify this.

LISTING 26-6: Mediator Implementation

```
// "mediator" suffix on class name used for demo purposes
public class DealershipAssessmentMediator : IDealershipAssessment
{
    private DealershipPerformanceStatus status;

    public DealershipAssessmentMediator(DealershipPerformanceStatus status)
```

continues

LISTING 26-6 *(continued)*

```
        {
            this.status = status;
        }

        public int TotalRevenue
        {
            get { return status.TotalRevenue;  }
            set { status.TotalRevenue = value;  }
        }

        public int TargetRevenue
        {
            get { return status.TargetRevenue; }
            set { status.TargetRevenue = value; }
        }

        public int NetProfit
        {
            get { return status.NetProfit;  }
            set { status.NetProfit = value;  }
        }

        public int TargetProfit
        {
            get { return status.TargetProfit;  }
            set { status.TargetProfit = value;  }
        }
    }
```

In Listing 26-6, the `DealershipAssessmentMediator` wraps a `DealershipPerformanceStatus` view model. When the mediator is passed into the domain objects, those domain objects set properties on the mediator. In turn, the mediator sets properties on the `DealershipPerformanceStatus` view model it encapsulates. This is also how the domain and the view model remain decoupled in a similar fashion to the mapping approach.

Deciding when to use the mediator comes down to experience, judgment, and a few key criteria. If you find yourself wanting to share a private domain state, the mediator should be high on your list of considerations. However, if your domain is still growing and the extra complexity of a mediator is not needed, it's likely to be a suboptimal choice. Performance-critical reports are another scenario in which you may want to avoid the mediator pattern due to the lack of low-level control. Where performance is a significant factor, you may want consider going directly to the datastore.

Going Directly to the Datastore

When performance and efficiency are important, or when going through layers of complexity and mappings is not desired, many DDD practitioners pull data for their reports directly from the database. In applications that use CQRS, dedicated, denormalized copies of the data are created for each report that needs them. When applications don't CQRS, it's common to query the datastore using raw data access technologies like ADO.NET. But it's also common to use low-level features of ORMs, such as NHibernate's HQL.

> **NOTE** *CQRS is covered in more detail in Chapter 24: "CQRS: An Architecture of A Bounded Context."*

In this section, you see an example of querying a project's main datastore with an ad-hoc reporting query. After that, you see an example of querying a denormalized copy of the data (a view cache), used specifically for reporting. Each of these examples involves creating a loyalty report for an online sports store. This report indicates to the business how successful its loyalty program is. Table 26-1 shows the format of the loyalty report.

TABLE 26-1: Display Format of the Loyalty Report

	POINTS (PER $)	NET PROFIT (% OF OVERALL)	SIGN-UPS	PURCHASES (% OF OVERALL)
Month A	…	…	…	…
Month B	…	…	…	…

Understanding how much profit the loyalty scheme is generating is the most important requirement of the loyalty report. As Table 26-1 shows, this is achieved by showing what percentage of overall profit came from the loyalty scheme for a given month. As part of their loyalty-optimization strategy, and to compete with rival companies, the online sports store often adjusts the number of points awarded. Using the loyalty report, the business can draw inferences about how changing this ratio affects the overall success of the scheme. Finally, no report would be complete without vanity metrics, so the loyalty report shows the number of loyalty scheme sign-ups as well.

Querying a Datastore

Building reports by directly querying a datastore gives you greater control and the ability to write efficient queries. To generate the loyalty report shown in Table 26-1, a SQL query may need to join and pull in data from a number of tables, including `orders`, `users`, `loyaltyAccounts`, `loyaltySettings`, and maybe even more. Many teams find that trusting an ORM to perform complex queries with lots of joins like this is a recipe for disaster. As a result, Micro-ORMs have become very popular because they provide some of the benefits Big-ORMs bring, yet they cut out a lot of the complexity. Micro-ORMs are a lower level of abstraction than Big-ORMs, providing you with more control over your queries and a better opportunity to make them fast and efficient.

Listing 26-7 shows an application service that uses Dapper (`https://code.google.com/p/dapper-dot-net/`), a concise Micro-ORM, to run a SQL query directly against the project's main SQL database without involving the domain. You can also see the definition of the view and database models being mapped to and from in Listing 26-8.

> **WARNING** *The SQL in this chapter's examples is not recommended as a source of inspiration. Rather, it aims to exemplify the greater control developers have when dealing with performance-sensitive data access.*

LISTING 26-7: Using Dapper to Query a SQL Database Directly

```
public LoyaltyReport Build(DateTime start, DateTime end)
{
    IEnumerable<PurchasesAndProfit> profits;
    IEnumerable<SignupCount> signups;
    IEnumerable<LoyaltySettings> settings;

    using(var con = new SqlConnection(connString))
    {
        con.Open();

        var pointsQuery = "select [Month], [PointsPerDollar] from loyaltySettings "
                        + "where [Month] >= @start "
                        + "and [Month] >= @end";
        settings = con.Query<LoyaltySettings>(pointsQuery,
                                              new{start=start, end=end});

        var signupsQuery = "select count(*) from loyaltyAccounts" +
                           "where isActive = true " +
                           "where [created] >= @start " +
                           "and [created] < @end ";
        signups = con.Query<SignupCount>(signupsQuery,
                                         new {start=start, end=end} );

        var profitQuery = "select " +
                        + "concat(month(o.[date]), '/', year(o.[date])) as Month, "
                        + " (select ((cast(count(*) as decimal) / (" +
                          "      select count(*) from orders" +
                          "      where [date] >= @start" +
                          "      and [date] <  @end" +
                          "  )) * 100)) as Purchases," +
                          "  (select ((sum(netProfit) / (" +
                          "      select sum(netProfit) from orders" +
                          "      where [date] >= @start" +
                          "      and [date] < @end " +
                          "  )) * 100)) as NetProfit" +
                          "  from orders o" +
                          "  join Users u on o.userId = u.id" +
                          "  join LoyaltyAccounts la on u.id = la.userId" +
                          "  where la.isActive = 1" +
                          "  and o.[date] >= @start" +
                          "  and o.[date] < @end" +
                          "  group by concat(month(o.[date]), '/', year(o.[date]))";
      profits = con.Query<PurchasesAndProfit>(profitQuery,
                                              new{start=start, end=end});
    }

    return Map(profits, signups, settings, start, end);
}
```

LISTING 26-8: Loyalty Report View and Database Models

```
// view / presentation models
public class LoyaltyReport
{
    public IEnumerable<LoyaltySummary> Summarries { get; set; }
}

public class LoyaltySummary
{
    public DateTime Month { get; set; }

    public int PointsRatio { get; set; }

    public double NetProfit { get; set; }

    public int SignUps { get; set; }

    public int Purchases { get; set; }
}

// database models
public class LoyaltySettings
{
    public DateTime Month { get; set; }

    public int PointsPerDollar { get; set; }
}

public class SignupCount
{
    public DateTime Month { get; set; }

    public int Signups { get; set; }
}

public class PurchasesAndProfit
{
    public DateTime Month { get; set; }

    public int Purchases { get; set; }

    public double Profit { get; set; }
}
```

Dapper adds the `Query<T>` extension method onto the native ADO.NET `SQLConnection`, as shown in Listing 26-7. `Query<T>` maps the results of the query you pass onto an object of type `T` that it creates for you. But the important issue in Listing 26-8 is that the developer is completely in control of the SQL being generated. In performance-critical reporting scenarios, low-level data access

circumvents the inefficiency associated with ORMs. Unfortunately, greater control comes at the cost of potential concept duplication.

> **NOTE** *You can see all the remaining code for this example, and all the other examples in this chapter, in this chapter's download files, including the implementation of* `Map()` *and a SQL Server Database project showing you the schema. Why not experiment with ORMs and compare the efficiency of the SQL it generates for a large dataset?*

> **NOTE** *Micro-ORMs are libraries that provide convenience methods—sometimes even DSLs—for querying databases. They are micro in the sense that they do not try to hide or abstract away the database; they just make querying it a little easier. You don't have to user a Micro-ORM; you can use raw ADO.NET or the equivalent low-level library for any datastore you are using.*

One of the challenges with direct-datastore queries is duplication. Some domain entities have computed properties. If you look at the SQL for the `profitQuery` in Listing 26-7, you can see the percentage of loyalty net profit being calculated against overall net profit in the same period. This is likely to be a calculation that occurs somewhere in the domain model as well. It's a risk and a violation of the don't repeat yourself (DRY) principle, because if this calculation were to change for any reason, both the SQL query in the report and the domain logic would both need to be updated—something that can easily be overlooked or forgotten.

Duplicating domain logic anywhere is not ideal. Clearly, if you update the logic in the domain and forget to update the logic in the datastore query, you can have several problems that annoy users or give the business completely wrong numbers. If you're worried about that concern, you may also want to consider storing the value of the computed property. In this scenario, you need to calculate the value of the computed property and then save it to the database whenever there is an update. However, if you update the database in multiple places, you need to recompute the value in multiple places or have a database trigger.

> **NOTE** *A common opinion is to avoid putting any business logic in the database because it can be harder to read, maintain, and test. Sometimes it may be necessary, but it's important to be aware of the trade-offs so that you can make an informed decision.*

Reading Denormalized View Caches

Sometimes, even directly querying the database with handcrafted SQL can be inefficient. For this reason, some DDD practitioners choose to create view/report-specific denormalized copies of the data (view caches). This is close in concept and implementation to CQRS. Whenever an update occurs, the main database is updated, but so are the relevant denormalized view caches. Figure 26-1 shows how you can implement this pattern for the loyalty report.

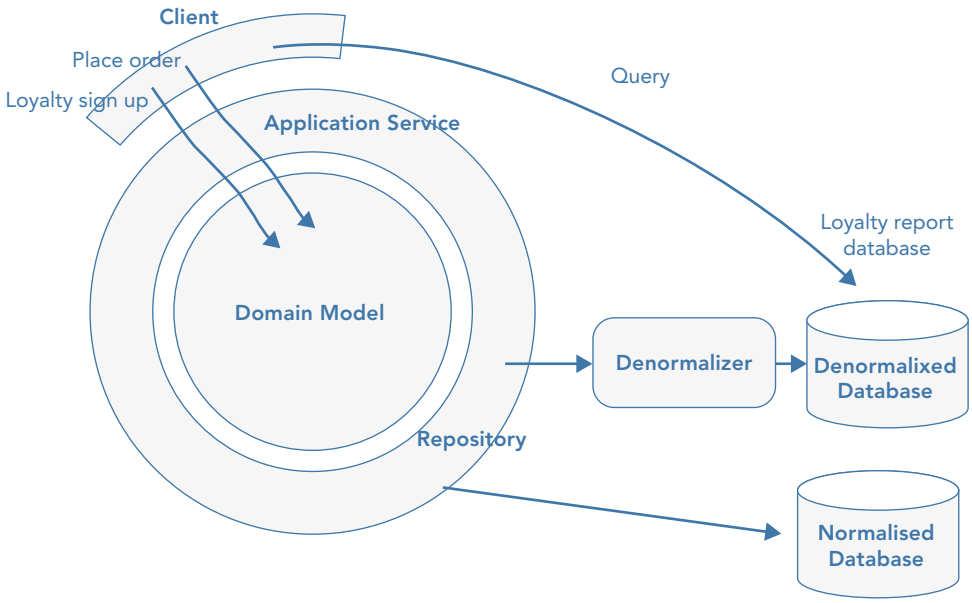


FIGURE 26-1: Denormalized view cache for the loyalty report.

As orders are placed and new users sign up (manifested as method calls, commands, domain events, and so on), the domain is invoked as usual. When creating denormalized views, though, the updates usually follow one path to the main database and at least one other via a denormalizer to the denormalized view cache, as per Figure 26-1. The denormalizer's job is usually to flatten the data so that queries are simple SQL select statements. Listing 26-9 shows an alternative `LoyaltyReportBuilder` that pulls in data from a denormalized view, emphasizing just how simple the query can be, by offloading the complexity to a denormalizer.

LISTING 26-9: Simple SQL Select Statements When Using Denormalized View Caches

```
public LoyaltyReport Build(DateTime start, DateTime end)
{
    IEnumerable<LoyaltySummary> summaries;
    using(var con = new SqlConnection(connString))
    {
       con.Open();
       var query = "select [Month], PointsPerDollar, NetProfit, Signups, Purchases"
                + " from denormalizedLoyaltyReportViewCache " +
                   "where [Month] >= @start " +
                   "and [Month] < @end";
       summaries = con.Query<LoyaltySummary>(query,
                                          new {start = start, end = end});
```

continues

LISTING 26-9 *(continued)*

```
        }

        return new LoyaltyReport
        {
            Summarries = summaries
        };
    }
```

As you can see in Listing 26-9, the complexity is massively reduced to just a single SQL select without joins. This is all thanks to extra up-front effort of denormalizing the data. You have to decide if that effort provides enough of a reduction in complexity or enough of a performance improvement on your projects before using this approach. You can mix and match where appropriate on your projects, though.

Building Projections from Event Streams

Applications that use event sourcing, which was introduced in Chapter 22, "Event Sourcing," require a different technique to generate reports because they don't store the current representation of the application state. Instead, event-sourced applications rely on a feature called projections. *Projections* are really just queries against event streams that produce some desired state or new streams, based on the contents of the events in the original stream.

Projection usage in a reporting context will be demonstrated in the following examples, where projections are used to create a health care diagnosis report. A health care authority uses this report to track the number of diagnoses made for certain medical conditions on a monthly basis. You can see the format of this report in Table 26-2.

TABLE 26-2: Health Care Diagnosis Report Format

	02/2014		03/2014		04/2014		05/2014	
	TOTAL	%	TOTAL	%	TOTAL	%	TOTAL	%
Diagnosis A	—	—	—	—	—	—	—	—
Diagnosis B	—	—	—	—	—	—	—	—

As Table 26-2 shows, each row in the health care diagnosis report tracks the number of times a diagnosis is made each month. For each month, the number of diagnoses made is shown alongside its percentage relative to all diagnoses made in that month. Using this report, the staff at the Health Care Foundation can look for trends in certain diagnoses. This may help them understand seasonal differences or correlate changes with other events such as the introduction of new vaccines or medical practices.

To implement this report, each monthly summary, for each diagnosis, is created as a new event stream with the naming format *diagnosis-{diagnosisId}-{month}*. These new streams are created from a projection that operates on a single event stream containing every diagnosis (the "diagnoses" stream). Figure 26-2 illustrates this process.

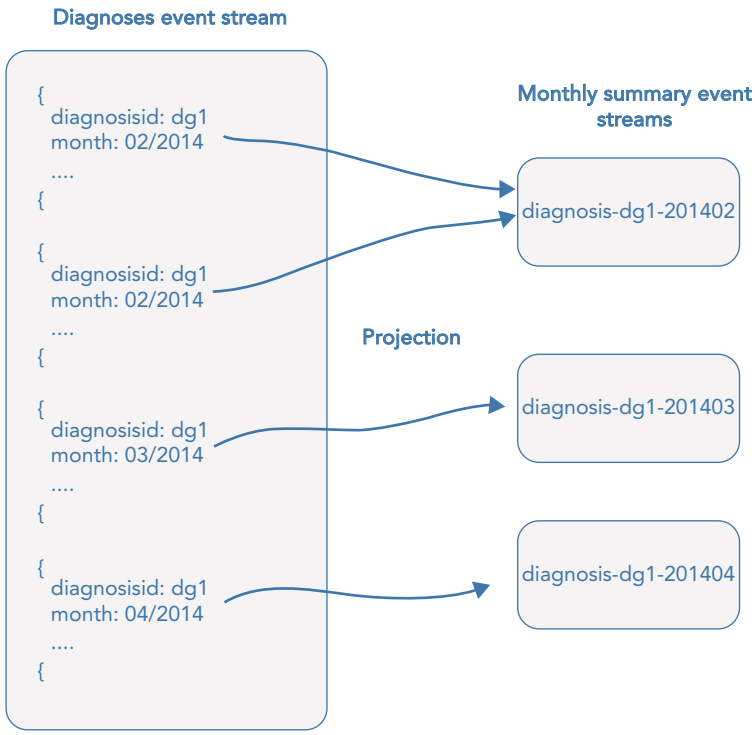


FIGURE 26-2: Projecting the "diagnoses" event stream onto event streams representing the monthly summary of each diagnosis.

For every diagnosis made, a stream contains all its events for each month, as Figure 26-2 shows. As an example, all diagnoses made for the diagnosis with ID `dg1` in February 2014 are projected into the stream `diagnosis-dg1_201402`. This means that when you come to build the report, all you have to do is count the number of events in a stream to get the total for that month. As you can see, using projections involves a similar philosophy to creating denormalized view caches—all the hard work is done up front to reduce the complexity involved in reading the data.

Setting Up ES for Projections

To work through the examples in this section, you need Event Store v3 rc2 to take advantage of newer projection capabilities. So you need to download the Event Store (`http://download.geteventstore.com/binaries/EventStore-OSS-Win-v3.0.0-rc2.zip`), extract it into a folder of your choice, and then run the following start-up command from PowerShell (as Administrator) from inside the directory you extracted the Event Store to:

```
.\EventStore.SingleNode.exe --db .\ESData --run-projections=all
```

Once the Event Store is started, you need to make a few changes to its configuration that enable some projection features. You can make these changes by navigating to the Projections tab in your

browser (`http://localhost:2113/projections`) and starting the projections `$by_category` and `$stream_by_category`.

> **NOTE** *If you are working through these examples, you can use a utility in the sample code that creates the "diagnoses" stream and inserts test data for you. You need to run the* `PPPDDD.Reporting` *web application and navigate to the URL* `http://localhost:{yourPort}/HealthcareEventProjectionReport`. *To see how this works, or to modify the test data, all the relevant code is located in the* `HealthcareEventProjectionReportController` *class.*

> **NOTE** *The Event Store is constantly evolving. In versions beyond v3 rc2, at some point, all projection features will be enabled by default. You may want to post questions on the Event Store's Google group (*`https://groups.google.com/forum/#!forum/event-store`*) or this book's discussion forum.*

Creating Reporting Projections

Projections are created using JavaScript, which can either be posted to the Hypertext Transport Protocol (HTTP) application programming interface (API) or manually entered into the admin website. This example uses the latter approach, which you can carry out by first navigating the Projections tab and then choosing New Projection. Because the projection you need to create groups all events for a diagnosis by month, it is called `DiagnosesByMonth`. The code for this projection is shown in Listing 26-10 and needs to be added to the Source input editor. When creating the `DiagnosesByMonth`, you need to select Continuous mode and check the Emit Enabled check box. Once this is complete, you can click Post to create the projection.

LISTING 26-10: Projection JavaScript That Partitions Each Diagnosis by Month

```
fromStream('diagnoses')
.whenAny(function(state, ev) {
    var date = ev.data.Date.replace('/', '');
    var diagnosisId = ev.data.DiagnosisId;
    linkTo('diagnosis-' + diagnosisId + '_' + date, ev);
});
```

The Event Store applies projections to each event in the stream. So the JavaScript in Listing 26-10 is applied to each event in the diagnoses stream. Each of those events is going to create a reference to the event in another stream. That other stream represents all diagnoses with the same `diagnosisId` in the same month. This is the process that was illustrated in 26-2. The Event Store supports this projection behavior with its `linkTo()`. `linkTo` adds a reference to the event passed in as the second argument on the stream whose name matches the first argument, creating that stream if necessary. Therefore, projections do not actually copy events; they just create references or pointers.

To check that the projection has worked, you can navigate to the Event Store's Streams tab and observe the names of newly created events. You should see events of the format `diagnosis-{diagnosisId}_{month}`, such as `diagnosis-d13_201402`. If you click on one of these streams, you see pointers to events that reside in the `diagnoses` stream, which the projection is based on.

Counting the Number of Events in a Stream

Each row in the report needs to show the number of diagnoses made in each month. As discussed previously, these totals are just the number of events in each stream created by the projection in Listing 26-10. One approach for querying the size of an event stream is to create another projection. It's an approach that many recommend, and it will be used in this example.

To create the projection that counts the monthly total for each diagnosis, you need to use the JavaScript in Listing 26-11. To follow along with this example, name this projection `DiagnosesByMonthCounts`. It should again use the continuous mode, but you can leave Emit Enabled unchecked. The projection is then ready to be created by clicking Post.

LISTING 26-11: Projection to Get Number of Events in All Streams Belonging to a Category

```
fromCategory('diagnosis')
.foreachStream()
.when({
     $init : function(s,e) {return {count : 0}},
     "diagnosis" : function(s,e) { s.count += 1} //mutate in place works
});
```

In the Event Store, *categories* are streams that have the same prefix. A *prefix* is a string of text preceding a hyphen. So all the streams that begin `diagnosis-` created by the first projection are in the diagnosis category. Categories provide the capability for the behavior of the projection in Listing 26-11. `foreachStream()` operates on each stream in a category, so the projection in Listing 26-11 goes through each stream in the diagnosis category and counts how many events there are. This count is stored in the projection's state. You can confirm this by querying the state for the projection, making sure to supply the name of the stream you want the state for as the value of the `partition` parameter. For example, a request for `http://localhost:2113/projection/DiagnosesByMonthCounts/state?partition=diagnosis-dg1_201402` gets the count of all events in that stream in the following format:

```
{
   count: 1
}
```

Creating As Many Streams As Required

With the Event Store, creating streams is usually a cheap operation, as was mentioned in Chapter 22. So getting the total number of diagnoses made in any given month is an opportunity to use projections that create further streams. Creating these streams follows the same pattern as the last two. First, events can be partitioned by month using the JavaScript shown in Listing 26-12, using

the same settings as the `DiagnosesByMonth` projection. To follow along with this example, call this projection `Months`.

LISTING 26-12: A Projection to Partition Diagnoses by Month

```
fromStream('diagnoses')
.whenAny(function(state, ev) {
    var date = ev.data.Date.replace('/', '');
    linkTo('month-' + date, ev);
});
```

Once you have run the `Months` projection, you then just need to sum up the numbers in each stream in the same way as the `DiagnosesByMonthCounts` projection. You can see the code for this projection in Listing 26-13. It should look familiar. Once this projection is running, all the streams that are needed to build the report will be in place.

LISTING 26-13: A Projection to Count the Overall Number of Diagnoses in Each Month

```
fromCategory('month')
.foreachStream()
.when({
    $init : function(s,e) {return {count : 0}},
    "diagnosis" : function(s,e) { s.count += 1 } //mutate in place works
 });
```

Building a Report from Streams and Projections

With a set of event streams containing all the needed data, building the report is reduced to a series of HTTP calls (or interactions with a client library) and mapping between objects. An application service called `HealthcareReportBuilder` demonstrates that in this final part of the current example. Listing 26-14 shows the initial version of the `HealthcareReportBuilder` containing the high-level logic required to build the report.

LISTING 26-14: High-Level Report-Building Logic

```
public class HealthcareReportBuilder
{
    public HealthcareReport Build(DateTime start, DateTime end,
        IEnumerable<string> diagnosisIds)
    {
       // report columns
       var monthsInReport = GetMonthsInRange(start, end).ToList();
       var monthlyOverallTotals = FetchMonthlyTotalsFromES(monthsInReport);
       var queries = BuildQueriesFor(
           monthsInReport, diagnosisIds).ToList();
       var summaries = BuildMonthlySummariesFor(
           queries, monthlyOverallTotals).ToList();

        return new HealthcareReport
        {
```

```
            Start = start,
            End = end,
            Summaries = summaries
        };
    }
    ...
}
```

To build the `HealthcareReport`, the `HealthcareReportBuilder` starts by calculating each month in the specified date range. For each of those months, it first fetches the total number of diagnoses from the Event Store, with the call to `FetchMonthlyTotalsFromES()` whose implementation is shown in Listing 26-15.

LISTING 26-15: Fetching Monthly Totals from the Event Store

```
private Dictionary<DateTime, int> FetchMonthlyTotalsFromES(
    IEnumerable<DateTime> months)
{
    // don't hard-code this URL. Access via entry point resource
    var projectionStateUrl = "http://localhost:2113/projection/MonthsCounts/state";

    var totals = new Dictionary<DateTime, int>();
    foreach(var m in months)
    {
        var streamName = "month-" + m.ToString("yyyyMM");
        var url = projectionStateUrl + "?partition=" + streamName;
        var response = new WebClient().DownloadString(url);
        var count = Json.Decode<DiagnosisCount>(response);
        totals.Add(m, count == null ? 0 : count.Count);
    }

    return totals;
}
```

To get the total for each month, the code in Listing 26-15 constructs a URL for the `MonthsCounts` projection's state resource. The name of the stream containing all the diagnoses for that month is used as the partition value. In response, the Event Store API returns the count as JSON. You can see this JSON response being mapped onto a `DiagnosisCount`, which is a data transfer object (DTO) that matches the structure of the JSON response, as Listing 26-16 shows. This object's `Count` property is then stored as the count for the month. When there are no diagnoses for a given month, there are no count values either. The code sets a value of zero in those cases.

LISTING 26-16: DiagnosisCount DTO That Matches API Response Format

```
public class DiagnosisCount
{
    public int Count { get; set; }
}
```

After obtaining the overall totals for each month, the `HealthcareReportBuilder` then gets the monthly total for each diagnosis. Before querying the Event Store, though, it carries out an intermediate step with the call to `BuildQueriesFor()`. `BuildQueriesFor()` creates a collection of strongly typed DTOs of the format shown in Listing 26-17 to make the code more expressive.

LISTING 26-17: DiagnosisQuery DTO

```
public class DiagnosisQuery
{
    public DateTime Month { get; set; }

    public string DiagnosisId { get; set; }
}
```

After creating the collection of `DiagnosisQueries`, the `HealthcareReportBuilder` uses them to finally query the Event Store for the monthly totals for each diagnosis, with the call to `BuildMonthlySummariesFor()`. The implementation of this is similar to `FetchMonthlyTotalsFromES()` in that the actual hard work is querying the Event Store and mapping the response, as shown in Listing 26-18.

LISTING 26-18: Fetching Monthly Total for Each Diagnosis from the Event Store

```
private IEnumerable<DiagnosisSummary>
BuildMonthlySummariesFor(IEnumerable<DiagnosisQuery> queries,
            Dictionary<DateTime, int> monthlyTotals)
{
    foreach (var q in queries) // may want to run these in parallel for perf
    {
        var diagnosisTotal = FetchTotalFromESFor(q);
        var monthTotal = monthlyTotals[q.Month];
        var percent = monthTotal == 0 ? 0:
((decimal)diagnosisTotal/monthTotal)*100;

        yield return new DiagnosisSummary
        {
            Amount = diagnosisTotal,
            DiagnosisName = GetDiagnosisName(q.DiagnosisId),
            Month = q.Month,
            Percentage = percent,
        };
    }
}

private int FetchTotalFromESFor(DiagnosisQuery query)
{
    // don't hard-code this URL. Access via entry point resource
    var projectionStateUrl =
"http://localhost:2113/projection/DiagnosesByMonthCounts/state";

  var streamname = "diagnosis-" + query.DiagnosisId + "_" + query.Month.
```

```
ToString("yyyyMM");

    // may want to use caching here
    var response = new WebClient().DownloadString(projectionStateUrl +
"?partition=" + streamname);
    var count = Json.Decode<DiagnosisCount>(response);

    return count == null ? 0 : count.Count;
}
```

Aside from fetching the totals, `BuildMonthlySummaries()` calculates the percentages, using the monthly total previously fetched, and maps the results onto a `DiagnosisSummary`. Upon completion, each `DiagnosisSummary` is mapped onto the `HealthcareReport` view model, as shown in Listing 26-14. All this hard work is then complete, and you can render the report.

> **WARNING** *As you can see from Listing 26-14, fetching the total for each diagnosis for each month can result in a lot of HTTP requests. However, these values never change, so you can be aggressive with your caching strategy to heavily mitigate any performance issues.*

> **NOTE** *Using projections to create many streams might result in large numbers of stream within your Event Store instance or cluster. You should monitor performance metrics and disk space, as always, but you shouldn't be too concerned by default. The Event Store was built to handle instances with millions of streams (*`http://geteventstore.com/blog/20130210/the-cost-of-creating-a-stream/`*).*

DOMAIN REPORTING ACROSS BOUNDED CONTEXTS

Unfortunately, producing reports is not always as easy as querying a single datastore. When you have a distributed system, such as those discussed in Part II, "Strategic Patterns: Patterns for Distributed Domain-Driven Design," each bounded context has its own datastore(s) that requires additional work on your behalf to produce the reports. This section outlines two approaches that rely on techniques presented in earlier chapters. One approach is to use the event-driven principles of Chapter 12 to create a dedicated reporting bounded context that subscribes to lots of events having all the information it needs locally in a single database. Sometimes, though, you can get away with a much lighter approach, using the UI composition techniques outlined in Chapter 23, "Composing Applications."

Composed UI

Combining data from multiple bounded contexts to form a report can work, but usually only when most of the processing can be carried out in distinct phases, each by a single bounded context. Any other supporting information, like translating IDs to names, can also be carried out afterward by querying the bounded context that owns the source of the lookup. A territorial record label comparison report can be used to demonstrate this. An online music streaming organization can

use this report to show the popularity of each record label in a variety of countries. Popularity is a measure of the combined total of streams and downloads for every song belonging to a record label. Table 26-3 shows the layout of a territorial record label comparison report.

TABLE 26-3: Territorial Record Label Comparison Report

	NORTH AMERICA	EUROPE	ASIA
Record Label 1	—	—	—
Record Label 2	—	—	—
Record Label 3	—	—	—

One of the big challenges involved in producing the territorial record label comparison report is that streaming and downloads are completely independent parts of the business, each with its own bounded context. So to get the total of streams and downloads for each record label, the information from each of those bounded contexts needs to be combined, as Figure 26-3 shows.

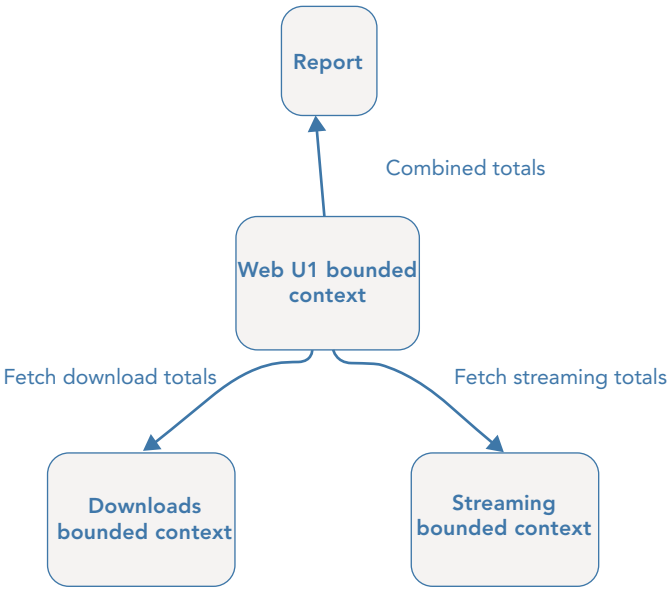


FIGURE 26-3: Aggregating data from multiple bounded contexts into a single report.

Fortunately, the aggregation can occur in distinct phases. Total downloads can be retrieved from the Downloads bounded context. At the same time, the total number of streams for each label can be retrieved from the Streaming bounded context. Using client- or server-side aggregation, as demonstrated in Chapter 23, the totals for each record label in each territory can easily be combined.

Separate Reporting Context

For reasons of performance, efficiency, or convenience, having all your data for reporting live inside the same datastore may be an important criteria. One example of this is data warehousing, in which the business wants to slice and dice all its data in new ways as it seeks to uncover insights. Often a business employs data scientists to carry out this important role. Having read Part II,

"Strategic Patterns: Patterns for Distributed Domain-Driven Design," about building distributed bounded contexts, you know that by default this is not possible due to each bounded context having its own datastore(s) and being loosely coupled to the others. But you also know that bounded contexts communicate with events, opening the possibility to create a special reporting context that subscribes to events from many bounded contexts, enabling it to gather all the data it needs.

Implementing a report context can vary drastically in scope and implementation. In the simplest case, it may be like any other bounded context in that it subscribes to events and stores them in a SQL database, as shown in Figure 26-4. At the other end of the scale, it may be pushing data through a variety of database technologies, recommendation engines, and machine learning algorithms, similar to Netflix (`http://techblog.netflix.com/2013/01/hadoop-platform-as-service-in-cloud.html`), as Figure 26-5 illustrates.

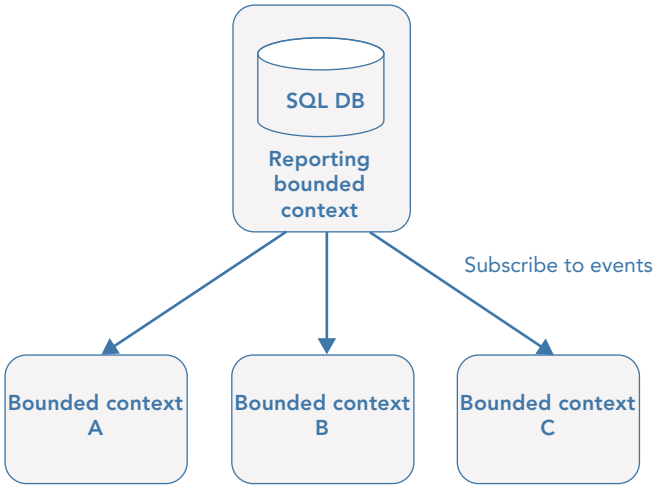


FIGURE 26-4: Standard reporting context.

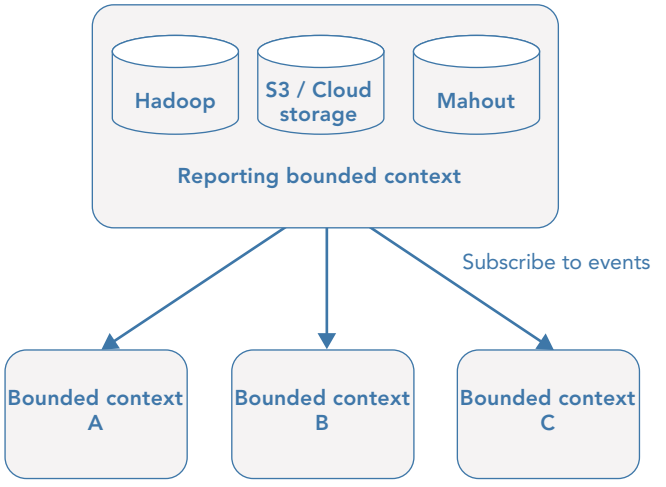


FIGURE 26-5: Complex data-processing reporting context.

To learn more about reporting and business intelligence in an event-driven Service Oriented Architecture (SOA) system, Arnon Rotem-Gal-Oz has published a detailed article on the InfoQ website (`http://www.infoq.com/articles/BI-and-SOA`).

THE SALIENT POINTS

- Reports can be created using a variety of tools and technologies that avoid domain coupling.
- Some reports operate on data within a bounded context, but some may need to query data from multiple bounded contexts.
- Mapping from domain objects onto view models is often the quickest approach, but it provides little control over low-level data access.
- Design patterns like the mediator pattern can be used to build reports or to juggle trade-offs such as coupling.
- It's okay to go directly to the datastore if you need queries to be inefficient, but duplication of concepts and violation of DRY is a concern to be mindful of.
- Querying the main database and querying denormalized view caches are two direct data access approaches.
- Denormalized view caches move all the hard work into the denormalization process in return for simpler queries.
- Using projections of event streams also trades off background processing in favor of simpler reads.
- You can query data from multiple bounded contexts by using UI composition in some cases and a separate reporting context in others.

INDEX

A

abstraction, aggregates and, 435
Active Record pattern, 67
aggregates, 318–322, 427–428, 434–435, event–sourced, implementation
  abstraction and, 435
  boundaries, 442
    business use cases, 449–450
    collections, 448–449
    containers, 448–449
    eBidder case study, 443–444
    HAS-A relationships, 449
    invarients, 444–446
    refactoring, 449
    transaction alignment, 446–448
    UI influence, 448
  consistency, 435–441
  event-sourced
    concurrency and, 609–610
    persistence, 605–609
    rehydrating, 605–609
    structuring, 600–605
    testing, 610–611
  implementation
    concurrency, 465–467
    eventual consistency, 463–465
    persistence, 458–462
    referencing aggregates, 454–458
    root, 450–454
    transactional consistency, 462–463
  invariants and, 435
    boundaries, 444–446
  size, 441–442
  UIs, 649–651
ambiguity, eliminating, 144–145
analysis model, 5, 43
Anemic Domain Model, 67–68
  functional programming and, 68–71
anticorruption layer, 38, 95–96
antipatterns, repositories
  ad hoc queries, 506–507
  lazy loading, 507
  reporting and, 507–508
APIs (application programming interfaces), hypermedia–driven
  business components, 225–226
  dogfooding, 248
  HTML *versus* data, 649
  hypermedia-driven
    entry point resource, 273–274
    HAL, 274–279
    URI templates, 279–286
  sharing, 227
application architecture, 105
  application service layer, 108
  bounded context architectures, 111–112
  communication, layers, 108–109
  dependency injection, 107
  domain layer, 107
  infrastructural layers, 108
  layered, 106
  schema sharing, 109–111
  testing, 109
application clients, 117–119
application logic, 689
  application services, 698–700
  frameworks, 698–700
  infrastructure, 690
    authentication, 695–696
    authorization, 695–696
    business use cases, 698
    communication, 696–698
    errors, 692–694
    logging, 694–695
    metrics, 694–695
    monitoring, 694–695
    transactions, 691–692
    validation, 690–691
application service layer, 108, 112–113
  application logic, 114
  business use cases, 115
  capabilities, 114–115

application service layer (*continued*)
CRUD and, 115
domain layer, implementation and, 115–116
domain logic, 114
domain reporting, 116
read models, 116–117
testing, domain events, 424–425
transactional models, 116–117
application service patterns
async/wait, 708–709
command processor, 701–704
publish/subscribe, 704–706
request/reply, 706–708
application services, testing
application logic and, 698–700
testing
domain terminology and, 709–710
functionality, 710–712
applications
authoritative, UIs, 647–648
autonomous, UIs, 646–647
architecture[RG1], evolutionary, 191
assumptions, challenging, 142
asynchronous messaging, event-driven reactive DDD, 173
async/wait pattern, 708–709
audit trails, repositories, 502
authoritative applications, UIs, 647–648
autonomous applications, UIs, 646–647
autonomy of bounded contexts, 153
availability, 161

B

BBoM (Big Ball of Mud), 4, 38–39
anticorruption layer, 38
development and, 5–6
BDD (Behavior-Driven Development), 22–23
boundaries, aggregates
business use cases, 449–450
collections, 448–449
containers, 448–449
eBidder case study, 443–444
HAS-A relationships, 449
invariants, 444–446
refactoring, 449
transactions, 446–448
UI influence, 448
bounded contexts, 79–81, 152, distributed bounded contexts, REST, reporting across, reporting from within. *See also* context maps
anticorruption layer, 95–96
architecture, application architecture and, 111–112
autonomy, 153
business capabilities and, 83
communications, teams, 84–85
complex, CQRS and, 670–672
CQRS and, 670
data storage, local, 216–224
distributed bounded contexts
integration, 162–165
RPC (remote procedure call), 166–168
implementation, 85–89
integration, 97–98, 155–156
database integration, 155–156
legacy systems, 158–161
models blur, 156–157
multiple teams, 156
mass transit and, 235–236
messaging bridge, 236–243
namespaces, 154
open host service, 97
partnership, 98
physical boundaries, 157–158
relationships, 95–100
reporting across
composed UI and, 733–734
separate reporting context, 734–736
reporting from within
datastore, 720–726
event streams, projections, 726–733
reports from domain objects, 714–720
REST
DDD, 268–269
SOA, 269
schema sharing, 109–111
shared kernel, 96–97
as SOA services, 175–178
versus subdomains, 85
teams, 83–84
upstream downstream relationship, 98–100
business analysts, 17
business components, APIs, 225–226
business model, 27–28

C

Challenging Model, 29
change tracking, persistence and, 497–499
classes, FollowerDirectory, 250
code
common language lack, 4–5
model, 43–44
ubiquitous language in analysis, 47–48
reusing [RG2], generic repositories, 483–486
Code Probing, 29

collaboration
  common language, 16–17
  CRC (Class Responsibility Collaboration Cards), 21
collection
  repositories, 502
  value objects and, 349–351
command processor pattern, 701–704
commands
  e-commerce application, 200–204
  messaging, 185–186
  NServiceBus, 200–204
common language, collaboration and, 16–17
communication across[RG3] layers, 108–109
component diagrams, 188
composed UIs, 646–648
composite UIs, 88
concepts, naming, 21–22
concurrency
  aggregates, event-sourced, 609–610
  Event Store, 618–621
  repositories, 502
consistency, 215–216
containers diagrams, 188–191
context maps, 91–92, reality maps. *See also* bounded contexts
  business work flow, 102
  communication, 102
  integration, 100
  new starters, 102
  obstacles, 102
  ownership, 101
  reality maps
    core domain, 94
    organizational reality, 93–94
    relevant realities, 94
    technical reality, 92–93
  responsibility, 101
  retaining integrity, 101
continuity, entities and, 362
continuous modeling, 142
core domain, 11, 35–36
  context maps, 94
  initial product, 39
  lack of, 39–40
  Pottermore website, 36
  as product, 36
core problem, focusing on, 33–37
CQRS (Command Query Responsibility Segregation), 87, 596, 669–670, command side, query side
  command side
    business requests, 675–676
    explicit modeling, 672–673
    presentational distractions, 674–675
  event sourcing and, 637–638
    synergy, 638–639
    view caches, 638
  misconceptions, 679–680
  patterns, scale and, 680–685
  query side
    materialized, domain events and, 678–679
    reports mapped to data model, 676–678
CRC (Class Responsibility Collaboration Cards), 21
CRUD (create, read, update, delete), 87

D

data APIs, 649
data duplication, 223–224
data model, mapping domain model to, 486–487
database integration, distributed bounded contexts, 162–163
datastore, reporting and, 720–726
DDD (Domain-Driven Design), 3, emerging patterns, life cycle patterns, models, patterns
  applying principles, 133–142
  behavior, capturing, 134
  collaboration, 11, 125–126
  communication, 11–12, 125
  component diagrams, 188
  context, 124–125
  core domain, 11
  cost of applying, 127–130
  domain services, 317–318
  emerging patterns
    domain events, 324–326
    event sourcing, 326–327
  event-driven reactive DDD, 170–174
  events, 187–188
  as framework, 13
  life cycle patterns
    aggregates, 318–322
    factories, 322–323
    repositories, 323–324
  misconceptions, 12–13
  models
    applicability, 12
    creating, 11
    evolving, 12
  modules, 318
  patterns
    complexity and, 6–10
    problem space, 9–10
    solution space, 9–10
    strategic, 6–9
    tactical, 9, 12–13
  problem complexity, 126–127
  solution modeling, 135–142

DDD (Domain-Driven Design) (*continued*)
stakeholders, 132–133
tactical patterns, 122, 310
bounded contexts, 122
code *versus* DDD principles, 123–124
perfection beliefs, 122–123
team, educating, 132
value objects, 314–317
vision, 133
debugging, 201–202
DefiningCommandsAs( ) method, 196
deliberate discovery, models and, 28–29
dependency injection
architecture, 107
domain services and, 400
design, domain–driven. *See also* DDD (Domain-Driven Design)
containers diagrams, 188–191
domain-driven
component diagrams, 188
events, 187–188
evolutionary architecture, 191
Model-Driven Design, 41
analysis model, 43
code model, 43–44
team modeling, 45–47
upfront design, 44–45
developers as problem solvers, 146
development team, role, 18
diagrams
component diagrams, 188
containers, 188–191
discovery, 143
distributed bounded contexts, integration
integration
database integration, 162–163
flat file integration, 163–164
messaging, 165
RPC, 164–165
RPC (remote procedure call), 166–167
coupling and, 168
resiliency and, 167
scalability and, 167–168
distributed transactions
reliability and, 169–170
scalability and, 169–170
documentation, Harvesting and Documenting, 29
dogfooding APIs, 248
domain events, 405–406, event handling, implementation patterns, testing
domain services and, 402–403
event handling
application logic, 410
domain logic, 409–410
implementation patterns
DomainEvents class, 415–418
in-memory bus, 412–415
IoC containers, 421–422
.Net Framework Events model, 410–412
return events, 419–421
testing
application service layer, 424–425
unit testing, 422–423
domain events pattern
asynchrony, 407–408
external, 408–409
internal, 408–409
reacting to events, 407
domain experts
role, 18
working with, 19
domain knowledge, 17
domain layer, 60, 107
domain logic, 700
domain model pattern, 62–65
data model, mapping to, 486–487
event streams, 491
versus persistence model, 482–483
domain models, 42
abstractions, 54–56
anemic, 395–396
code model and, 44
creating, 52–56
versus domains, 42–43
implementation patterns, 60–61
Active Record pattern, 67
Anemic Domain Model, 67–71
domain model pattern, 62–65
table module pattern, 67
Transaction Script pattern, 65–67
relevance, 54
terminology, 54
truth and, 52–54
domain services, 317–318, 389–390
versus application services, 396
contract representation, 394
domain and, 398–399
dependency injection, 400
domain events, 402–403
double dispatch, 401–402
factory methods, 399–400
service locators, 400–401
encapsulation, 390–393
policy encapsulation, 390–393
process encapsulation, 390–393
service layer and, 397–398

domains
versus domain models, 42–43
generic, 37
subdomains, 37
supporting, 37
vision, 32–33
vision statements, 33
double dispatch, domain services and, 401–402

E

eBidder case study, 443–444
e-commerce application, 186–187, design
commands, 200–204
consistency, 215–216
design
containers diagrams, 188–191
domain-driven, 187–188
evolutionary architecture, 191
publishing events, 204–206
subscribing to events, 206–207
web application, commands from, 192–199
emerging patterns
domain events, 324–326
event sourcing, 326–327
enterprise model, 79
entities, 361, identifiers
behavior and, 374–377
context-dependency, 363
continuity and, 362
design for distribution, 378–379
entity base class, 313–314
identifiers
datastore-generated, 368–369
GUIDs/UUIDs, 365–366
incremental numeric counters, 364–365
natural keys, 363–364
strings, 367–368
identity and, 362
invariants, 371–374
specifications and, 380–382
memento pattern, 385–386
patterns, 310–314
pushing behavior to value objects, 369–371
real-world modeling fallacy, 377
side effects, 386–388
state pattern, 382–385
validation, specifications and, 380–382
Entity Framework, repositories
application service, 571–574
configuration, 576–577
implementation, 566–571
model, 559–565
queries, 574–576
solution setup, 558–559
event sourcing, 595–596, aggregates, state
aggregates
concurrency and, 609–610
persistence[RG4], 605–609
rehydrating, 605–609
structuring, 600–605
testing, 610–611
benefits, 639–640
costs, 640–641
CQRS with, 637–639
repositories, 605–607
state
as snapshot, 596–597
as stream of events, 597–600
Event Store, event streams, temporal queries
Atom feed, 298–303
C# client library, 628–632
concurrency, 618–621
event streams
appending to, 614–615
creating, 614
queries, 615–616
event subscriber, 298–303
IEventStore interface, 612
installation, 291–292
persisting events, 286–291
viewing, 292–294
projections, 635–637
publishing events, 294–298
snapshots, 616–618
SQL Server-based, 621–627
storage format, 612–614
temporal queries
multiple streams, 634–635
single stream, 632–634
Young, Greg, 628
Event Storming, 25
event streams
appending to, 614–615
creating, 614
number of events, 729
queries, 615–616
reporting, 726–733
event-driven reactive DDD, 170–174
events
domain events, 324–326
domain-driven design, 187–188
evolutionary architecture, 191
explicitness
versus implicitness, 144–145
repositories as explicit contract, 492–493
value objects and, 331–333

F

factories, 322–323, factory methods
  creation type decisions, 472–474
  encapsulation, 470–472
  factory methods
    aggregates and, 474–475
    domain services and, 399–400
  reconstitution, 475–476
  static factory methods, 345–346
  use *versus* construction, 470
FindUsersFollowers( ) function, 250
flat file integration, distributed bounded contexts, 163–164
FollowerDirectory class, 250
frameworks, application logic and, 698–700

G

generic domains, 37
generic repositories, 483–486
graphs, object graphs
  associations, 430–431
  IDs *versus* object references, 431–434
  single traversal direction, 428–430
GWT (Given, When, Then) format, 22

H

Harvesting and Documenting, 29
HTML (Hypertext Markup Language), APIs, 649
HTTP (Hypertext Transport Protocol), 246
  reasons to use, 247–248
  REST and, 266–268
  RPC over, 248
    JSON, 259–263
    SOAP, 249–259
    WCF, 250–258
    XML, 259–263
  status codes, 266–267
  verbs, 266
hypermedia, REST and, 265, 271–272

I

identity, entities and, 362
impact mapping, 25–27
implementation patterns, domain models, 60–61
  Active Record pattern, 67
  Anemic Domain Model, 67–71
  domain model pattern, 62–65
  table module pattern, 67
  Transaction Script pattern, 65–67
implicitness *versus* explicitness, 144–146
infrastructural layers, 108
invariants, 322
  aggregates and, 435
    boundaries, 444–446
  specifications and, 380–382

J

JSON (JavaScript Object Notation), 248

K

knowledge crunching, 15–16
  BDD (Behavior-Driven Development), 22–23
  business analysts and, 17
  collaboration, 16–17
  conversations, 19–20
  CRC (Class Responsibility Collaboration Cards), 21
  models, concept naming, 21–22
  paper-based systems, 24
  process, 17–18
  questions to ask, 20
  rapid prototyping, 23–24
  sketching, 20–21
  use cases, 20
KPIs (key performance indicators), 713

L

layered architecture, 106
legacy code, 78–79
life cycle patterns
  aggregates, 318–322
  factories, 322–323
  repositories, 323–324
local storage, bounded contexts, 216–224

M

mediator design pattern, reports and, 718–720
Memento pattern, 385–386, 488
messaging, gateways, message versioning, monitoring
  application maintenance, 227–235
  commands, 185–186
  consistency, 186
  distributed bounded contexts, 165
  gateways
    fault tolerance and, 208–209
    implementation, 209–212
    message retries, 212–215
  message bus, 182–184

message versioning
backward compatibility, 228–229
NServiceBus polymorphic handlers, 229–233
monitoring
errors, 233–234
scaling, 235
SLAs, 234–235
NServiceBus, 195–196
reliability, 184
store-and-forward pattern, 184–185
messaging bridge, Mass Transit
configuration, 236–238
declaring messages for use, 238
event publishing, 242–243
installation, 236–238
message handler, 239
subscribing to events, 239–240
system linking, 240–242
testing, 243
micro types, 347–349
micro-ORM, repositories
ADO.NET implementation, 583–592
application service, 581–583
configuration, 592–593
infrastructure, 578–581
solution setup, 577–578
Model Exploration Whirlpool, 29
Model-Driven Design, 41
analysis model, 43
application of, 56–57
code model, 43–44
team modeling, 45–47
upfront design, problems with, 44–45
Modeling, 29
modeling, continuous, 142
models
analysis, 43
boundaries, 38–39
defining, 82–85
business model, 27–28
code, 43–44
complexity growth, 74
concept naming, 21–22
deliberate discovery and, 28–29
domain concept applicability, 76–78
domain model *versus* enterprise model, 79
Event Storming, 25
existing, 24–29
impact mapping, 25–27
intent, 24–25
language ambiguity, 75–76
legacy code, 78–79
Model Exploration Whirlpool, 29
team modeling, 45–47
teams, multiple, 74–75
third-party code, 78–79
wrong, 142–143
modules, 318
MSA (Micro Service Architecture), 178–179

N

NHibernate, repositories
application service, 525–529
configuration, 540–541
database schema, 535–536
model, 511–525
presentation, 541–543
queries, 536–539
repository implementation, 529–535
solution setup, 509–510
NoSQL, persistence and, 351–352
NServiceBus, 186–187, messages
commands, 200–204
messages
defining, 192–195
sending commands, 197–199
send-only client, 195–196
web application creation, 192–196

O

object graphs
associations, 430–431
IDs *versus* object references, 431–434
single traversal direction, 428–430
objects. *See* value objects
owned UIs, 646–648

P

partnerships, 98
patterns, application service patterns, domain events, emerging, life cycle
Active Record pattern, 67
Anemic Domain Model, 67–71
application service patterns
async/wait, 708–709
command processor, 701–704
publish/subscribe, 704–706
request/reply, 706–708
domain events
asynchrony, 407–408
external, 408–409
internal, 408–409
reacting to events, 407

patterns, application service patterns (*continued*)
domain model, 62–65
domain services, 317–318
emerging
domain events, 324–326
event sourcing, 326–327
entities and, 310–314
implementation patterns, domain models, 60–71
life cycle
aggregates, 318–322
factories, 322–323
repositories, 323–324
Memento, 385–386, 488
micro types, 347–349
modules, 318
state, entities, 382–385
static factory methods and, 345–346
store-and-forward, 184–185
table model pattern, 67
tactical, 310
Transaction Script, 65–67
unit of work, 493–497
value objects, 314–317
persistence, framework, value objects
aggregates, event-sourced, 605–609
framework
change tracking, 497–499
mapping domain model, 487–491
mapping to data model, 486–487
value objects
NoSQL, 351–352
SQL, 353–359
persistence model, *versus* domain model, 482–483
Pottermore web site, 36
problem domains, 4
analysis model, 5
core problem, focusing, 33–37
decomposing, 31–32
development and, 5–6
distilling, 34–35
essence of the problem, 32–33
focus, 6
organization, 5
problem space, 9–10
product entity, 312–313
projections
Event Store, 635–637
reporting projections, 728–729
state and, 599
prototyping, rapid, 23–24
publish/subscribe pattern, 704–706

Q

queries, event streams, 615–616

R

rapid prototyping, 23–24
RavenDB, repositories, 543–544
application service, 548–551
configuration, 555–557
implementation, 553–555
model, 546–548
queries, 551–553
solution setup, 544–546
reactive DDD, SOA and, 174–179
read models, 116–117
rehydrating aggregates, 605–609
reliability, 161
reporting, 713–714, across bounded contexts, within bounded contexts
across bounded contexts
composed UI and, 733–734
separate reporting context, 734–736
within bounded contexts
datastore, 720–726
event streams, projections, 726–733
reports from domain objects, 714–720
repositories, 323–324, 479–481, antipatterns, Entity Framework, micro–ORM, NHibernate, persistence model
as anticorruption layer, 499–500
as antipattern, 481–482
antipatterns
ad hoc queries, 506–507
lazy loading, 507
reporting and, 507–508
audit trails, 506
collections, summaries, 502
concurrency, 503–506
domain model, *versus* persistence model, 482–483
entities, ID generation, 500–502
Entity Framework
application service, 571–574
configuration, 576–577
implementation, 566–571
model, 559–565
queries, 574–576
solution setup, 558–559
event sourcing, 605–607
as explicit contract, 492–493
generic, 483–486

micro-ORM
ADO.NET implementation, 583–592
application service, 581–583
configuration, 592–593
infrastructure, 578–581
solution setup, 577–578
NHibernate
application service, 525–529
configuration, 540–541
database schema, 535–536
model, 511–525
presentation, 541–543
queries, 536–539
repository implementation, 529–535
solution setup, 509–510
persistence model
change tracking, 497–499
versus domain model, 482–483
RavenDB, 543–544
application service, 548–551
configuration, 555–557
implementation, 553–555
model, 546–548
queries, 551–553
solution setup, 544–546
transaction management, unit of work pattern, 493–497
request/reply pattern, 706–708
resilience, event-driven reactive DDD, 171–172
REST (REpresentational State Transfer), 165, bounded contexts
ASP.NET, 273
bounded contexts
DDD, 268–269
problems with, 304–305
SOA, 269
HTTP and, 266–268
hypermedia and, 265, 271–272
metrics, 303–304
monitoring, 303–304
resources, 264–265
statelessness and, 265–266
versioning, 303
[RG1]architecture
[RG2]Should this be reusing?
[RG3]across
[RG4]persistence
ROI (return on investment), 36
RPC (remote procedure calls)
distributed bounded contexts, 164–165
coupling and, 168
resiliency and, 167
scalability and, 167–168
flavor, 263–264
over HTTP, 248
JSON, 259–263
SOAP, 249–259
WCF, 250–258
XML, 259–263

S

scalability, 161
distributed transactions, 169–170
Scenario Exploring, 29
server-side orchestration, 226
service locators, domain services and, 400–401
shared kernel, 96–97
side effects, 386–388
snapshots
Event Store, 616–618
SQL Server-based, 625–627
persistence, 607–609
state, 599–600
SOA (Service Oriented Architecture), 152
bounded contexts and, 175–178
disambiguation, 175
MSA (Micro Service Architecture), 178–179
reactive DDD and, 174–179
SOAP (Simple Object Access Protocol), 164, 248
decline, 258–259
RPC and, 249–259
software
complexity in, 4–6
organization lack, 5
solution space, 9–10
SQL (Structured Query Language), persistence and, SQL Server–based Event Store
persistence and
denormalization, 353–357
normalization, 357–359
SQL Server-based Event Store
loading events, 624–625
saving events, 623–624
schema, 621–622
snapshots, 625–627
streams, 622–623
SRP (Single Responsibility Principle), 78
stakeholders, role, 18
state, storing
projections, 599
snapshots, 599–600
storing
as snapshot, 596–597
as stream of events, 597–600
temporal queries, 597–598
state pattern, entities, 382–385

static factory methods, 345–346
storage of bounded contexts, 216–224
store-and-forward pattern, 184–185
strategic patterns, 6–9
stream of events, state, 597–600
subdomains, 37
  versus bounded contexts, 85
  for replacement, 39
supporting domains, 37

T

table model pattern, 67
tactical patterns, 9, 310
  misconceptions, 12–13
TDD (Test-Driven Development), 22
team modeling, 45–47
temporal queries, Event Store
  Event Store
    multiple streams, 634–635
    single stream, 632–634
  state and, 597–598
testing, 201–202
  aggregates, event-sourced, 610–611
  application service layer, domain events, 424–425
  application services, 709–712
  architecture, 109
  unit testing, domain events, 422–423
third-party code, 78–79
transaction management, unit of work pattern, 493–497
Transaction Script pattern, 65–67
transactional models, 116–117

U

UI (user interface), composition, examples
  aggregates, boundaries, 448
  APIs, HTML *versus* data, 649
  authoritative applications, 647–648
  autonomous applications, 646–647
  client side aggregation/coordination, 649–651
  composed, 646–648
  composite, 88
  composition
    AJAX data and, 226
    AJAX HTML and, 226–227
  examples
    data API-based, 658–666
    HTML API-based, 651–658
  owned, 646–648
  server side aggregation/coordination, 649–651
UL (ubiquitous language), 17
  boundaries, 82
  code model analysis, 47–48
  developing, 48–52
unit of work pattern, transactions and, 493–497
unit testing, domain events, 422–423
upfront design, problems with, 44–45
upstream downstream relationships, 98–99
  conformist, 100
  customer-supplier, 99
use cases, knowledge crunching and, 20

V

validation, entities, 380–382
value objects, 314–317, 329–330, persistence
  attribute-based equality, 333–336
  behaviors, 337
  cohesiveness, 337
  collection and, 349–351
  combinability, 339–341
  concepts with no identity, 330–331
  explicitness, 331–333
  identity, 333
  identity-less concepts, 330–331
  immutability, 337–339
  persistence
    NoSQL, 351–352
    SQL, 353–359
  testing, 344–345
  validation, 341–344

W

WCF (Windows Communication Foundation), 165
  RPC implementation over HTTP, 250–258
  service clients, 255–258
  service contracts, 252–253
  services, 251–252
  testing services, 253–255
web application, commands from, 192–199
working backwards, Amazon, 33

X

XML (eXtensible Markup Language), 163, 246

Try Safari Books Online FREE for 15 days and take 15% off for up to 6 Months*

Gain unlimited subscription access to thousands of books and videos.



With Safari Books Online, learn without limits from thousands of technology, digital media and professional development books and videos from hundreds of leading publishers. With a monthly or annual unlimited access subscription, you get:

- Anytime, anywhere mobile access with Safari To Go apps for iPad, iPhone and Android
- Hundreds of expert-led instructional videos on today's hottest topics
- Sample code to help accelerate a wide variety of software projects
- Robust organizing features including favorites, highlights, tags, notes, mash-ups and more
- Rough Cuts pre-published manuscripts

START YOUR FREE TRIAL TODAY!
Visit: **www.safaribooksonline.com/wrox**

*Discount applies to new Safari Library subscribers only and is valid for the first 6 consecutive monthly billing cycles. Safari Library is not available in all countries.

Safari Books Online

wrox
An Imprint of WILEY
Now you know.



Programmer to Programmer™

Connect with Wrox.

Participate

Take an active role online by participating in our P2P forums @ **p2p.wrox.com**

Wrox Blox

Download short informational pieces and code to keep you up to date and out of trouble

Join the Community

Sign up for our free monthly newsletter at **newsletter.wrox.com**

Wrox.com

Browse the vast selection of Wrox titles, e-books, and blogs and find exactly what you need

User Group Program

Become a member and take advantage of all the benefits

Wrox on twitter

Follow @wrox on Twitter and be in the know on the latest news in the world of Wrox

Wrox on facebook

Join the Wrox Facebook page at **facebook.com/wroxpress** and get updates on new books and publications as well as upcoming programmer conferences and user group events

Contact Us.

We love feedback! Have a book idea? Need community support?
Let us know by e-mailing wrox-partnerwithus@wrox.com

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.