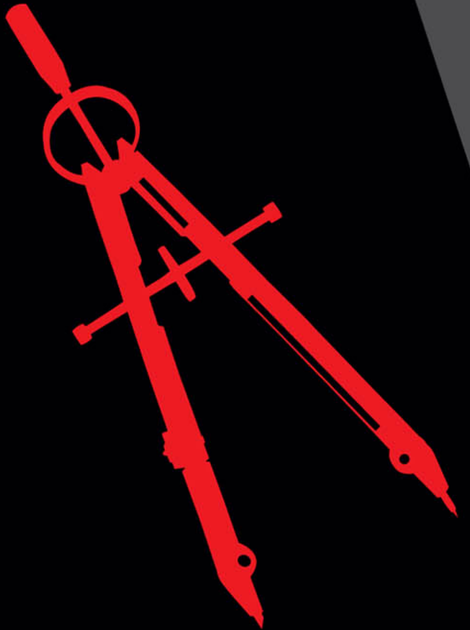
Microsoft

Microsoft .NET: Architecting Applications for the Enterprise

Second Edition



Dino Esposito
Andrea Saltarello

Microsoft

Microsoft .NET: Architecting Applications for the Enterprise, Second Edition

Dino Esposito
Andrea Saltarello

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2014 by Dino Esposito and Andrea Saltarello

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2014940680
ISBN: 978-0-7356-8535-2

Printed and bound in the United States of America.

Second Printing: January 2015

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at http://aka.ms/tellpress.

Microsoft and the trademarks listed at http://www.microsoft.com/en-us/legal/intellectualproperty/Trademarks/EN-US.aspx are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions and Developmental Editor: Devon Musgrave
Project Editor: Carol Dilllingham
Editorial Production: Waypoint Press, www.waypointpress.com
Peer Reviewer: Cesar De la Torre Llorente
Copyeditor: Roger LeBlanc
Indexer: Christina Yeager
Cover: Twist Creative • Seattle and Joel Panchot

To my wife Silvia. You make me feel sandy like a clepsydra. I get empty and filled all the time; but it's such a thin kind of sand that even when I'm full, without you, I just feel empty.

—Dino

To Laura, mum, and Depeche Mode. Moved, lifted higher. Moved, by a higher love.

—Andrea

This page intentionally left blank

Contents at a glance

PART I	**FOUNDATION**	
CHAPTER 1	Architects and architecture today	3
CHAPTER 2	Designing for success	27
CHAPTER 3	Principles of software design	53
CHAPTER 4	Writing software of quality	85
PART II	**DEVISING THE ARCHITECTURE**	
CHAPTER 5	Discovering the domain architecture	113
CHAPTER 6	The presentation layer	137
CHAPTER 7	The mythical business layer	167
PART III	**SUPPORTING ARCHITECTURES**	
CHAPTER 8	Introducing Domain Model	191
CHAPTER 9	Implementing Domain Model	217
CHAPTER 10	Introducing CQRS	255
CHAPTER 11	Implementing CQRS	291
CHAPTER 12	Introducing event sourcing	311
CHAPTER 13	Implementing event sourcing	325
PART IV	**INFRASTRUCTURE**	
CHAPTER 14	The persistence layer	353

This page intentionally left blank

Contents

Introduction xiii

PART I FOUNDATION 1

Chapter 1 Architects and architecture today 3

What's software architecture, anyway? 4
Applying architectural principles to software 4
Acknowledging requirements 7
What's architecture and what's not 11
The architecture process 14
Who's the architect, anyway? 17
An architect's responsibilities 18
The role of the architect 20
Common misconceptions about architects 21
Summary 24
Finishing with a smile 25

Chapter 2 Designing for success 27

The "Big Ball of Mud" 28
Causes of the "Big Ball of Mud" 28
Symptoms of the "Big Ball of Mud" 32
Using metrics to detect a BBM 34
Mechanics of software projects 35
Organizational culture 35
Helping the team write better code 38
Getting out of the mess 43
That odd thing called "legacy code" 44
Checkmate in three moves 45
Deciding whether or not to add manpower 49
Summary 50
Finishing with a smile 51

What do you think of this book? We want to hear from you!
Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Chapter 3 Principles of software design 53

Universal principles of software design 54
From *spaghetti* code to *lasagna* code 54
Separation of concerns 57
Isolation 57

Object-oriented design 57
Pertinent classes 58
Program to an interface 59
Composition vs. inheritance 61
A second pass at object-orientation 63

Development and design vectors 64
SOLID principles 64
Patterns for handling dependencies 69
Coding vectors 72
Use of patterns 75
Refactoring 77

Defensive programming 78
The If-Then-Throw pattern 78
Software contracts 79

Summary 83
Finishing with a smile 84

Chapter 4 Writing software of quality 85

The art of writing testable code 86
What is testability, anyway? 86
Testing your software 88
Common practices of software testing 96

The practice of code extensibility 101
Interface-based design 102
Plugin architecture 103
State machines 103

Writing code that others can read 104
Readability as a software attribute 105
Some practical rules for improving readability 107

Summary 109
Finishing with a smile 110

PART II DEVISING THE ARCHITECTURE 111

Chapter 5 Discovering the domain architecture 113

The real added value of domain-driven design 114
What's in DDD for me? 114

Conducting analysis using DDD 115
Strategic model design 116
The ubiquitous language 118
Purpose of the ubiquitous language 118
Structure of the ubiquitous language 119
How to define the ubiquitous language 119
Keeping language and model in sync 121
Bounded contexts 122
Discovering contexts 122
Context mapping 125
Giving each context its own architecture 127
The layered architecture 129
Origins of the layered architecture 129
Presentation layer 131
Application layer 132
Domain layer 134
Infrastructure layer 134
Summary 135
Finishing with a smile 135

Chapter 6 The presentation layer 137

User experience first 138
Focus on interactions 138
UX is not UI 140
How to create an effective experience 143
Realistic scenarios 147
ASP.NET websites 147
Web Forms vs. ASP.NET MVC 152
Adding device support to websites 155
Single-page applications 160
Desktop rich client 164
Summary 166
Finishing with a smile 166

Chapter 7 The mythical business layer 167

Patterns for organizing the business logic 167
The fairytale of CRUD and an architecture Prince Charming 168
The Transaction Script pattern 169
The Domain Model pattern 172
The Anemic Domain Model (anti-)pattern .174
Moving the focus from data to tasks 176
Task orchestration in ASP.NET MVC 177
Orchestrating tasks within the domain 180

Moving data across the boundaries 182
Data flow in layered architecture 182
Sharing the Domain Model entities 184
Using data-transfer objects 185
Summary. 187
Finishing with a smile 188

PART III SUPPORTING ARCHITECTURES

Chapter 8 Introducing Domain Model 191

The data-to-behavior shift 191
Rationale behind models and domains 192
Database is infrastructure 195
Inside the domain layer 196
Domain model 196
Aggregates 199
Domain services 205
Domain events 209
Cross-cutting concerns 212
Summary. 215
Finishing with a smile 215

Chapter 9 Implementing Domain Model 217

The online store sample project. 218
Selected use-cases 218
Selected approach 219
Structure of the I-Buy-Stuff project 220
Selected technologies 222
Bounded contexts for an online store 222
Context map of the I-Buy-Stuff application 224
The noble art of pragmatic domain modeling 225
Behavior is a game-changer 225
Entities scaffolding. 228
Value objects scaffolding 231
Identifying aggregates 235
Persisting the model 243
Implementing the business logic 248
Finding an order. 248
Placing an order 249
Fidelity card (or customer loyalty program) 253
Summary. 253
Finishing with a smile 254

Chapter 10 Introducing CQRS 255

Separating commands from queries .256
Generalities of the CQRS pattern .256
Benefits of CQRS .258
Fitting CQRS in the business layer .260
CQRS always pays the architecture bill .262

The query stack .264
The read domain model .264
Designing a read-model façade .266
Layered expression trees .269

The command stack .274
Getting back to presentation .274
Formalizing commands and events .277
Handling commands and events .280
Readymade storage .286

Summary .288
Finishing with a smile .289

Chapter 11 Implementing CQRS 291

CQRS implementations .291
Plain and simple CQRS .292
CQRS with command architecture .294

Implementing a query stack .296
Creating a read façade .296
Packaging data for callers .298

Implementing a command stack .302
Laying the groundwork .302
Orchestrating use-cases via commands .305

Summary .309
Finishing with a smile .310

Chapter 12 Introducing event sourcing 311

The breakthrough of events .312
The next big thing (reloaded) .312
The real world has events, not just models .312
Moving away from "last-known good state" .313
The deep impact of events in software architecture .315

Event-sourcing architecture .318
Persisting events .318
Replaying events .321
Summary .323
Finishing with a smile .324

Chapter 13 Implementing event sourcing 325

Event sourcing: Why and when .325
Why event sourcing is a resource .326
When event sourcing is appropriate .327
Event sourcing with replay .329
A live-scoring system .329
Implementation of the system .331
Event sourcing with aggregate snapshots .342
A mini enterprise resource planning system .342
Implementation of the system .344
Summary .348
Finishing with a smile .349

PART IV INFRASTRUCTURE 351

Chapter 14 The persistence layer 353

Portrait of a persistence layer .353
Responsibilities of the persistence layer .354
Design of a Repository pattern .355
Implementing repositories .359
The query side of repositories .359
Persisting aggregates .363
Storage technologies .364
Why should you consider nonrelational storage? .368
Familiarizing yourself with NoSQL .369
What you gain and what you lose .370
Planning a sound choice .375
Summary .377
Finishing with a smile .378

Index 379

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Introduction

> *Good judgment comes from experience, and experience comes from bad judgment.*
>
> *—Fred Brooks*

We find that the preceding quote contains the essence of software architecture and the gist of the architect's role. Software architecture requires judgment because not all scenarios are the same. To exercise sound judgment, you need experience, and in this imperfect world, experience mostly comes from making some mistakes and bad choices—from bad judgment.

However, the world we live in often doesn't give you the opportunity (or even the time) to form your own experience-based knowledge from which good judgment is developed. More often than not, all that executives want from architects is the right architecture right away.

We've written this book primarily to endow you with a solid, reusable, and easily accessible base of knowledge about software architecture. In past years, we've completed projects using technologies like Microsoft Windows DNA, Distributed COM, multitier CRUD, SOA, DDD, CQRS, and event sourcing. We've used Microsoft Visual Basic 6 as well as C#, C++, Java, and JavaScript. We've seen technical solutions change frequently and perspectives about these approaches also evolve.

In the end, we came to the same conclusion as Fred Brooks. We don't wear white coats, we're not doctors, and we're not writing prescriptions. Our purpose here is to aggregate various positions, add our own annotations and comments to those positions, and generate an honest summary of facts and perspectives.

In these times in which developers and architects are asked to do it right and right away, we offer a snapshot of knowledge—a readymade software architect's digest for you to use as the starting point for further investigation and to use to build up your own judgment. If software architecture were a theorem, this book (we hope) would provide a collection of necessary lemmas.

Organization of this book

Software architecture has some preconditions (design principles) and one post-condition (an implemented system that produces expected results). Part I of this book, titled "Foundation," lays the foundation of software architecture and focuses on the role of the architect, the inherent mechanics of software projects, and aspects—like testability and readability—that turn software into top-quality software.

Part II, "Devising the architecture," focuses on the topmost layers that form a typical enterprise system: the presentation layer and business layer. We left for later the canonical third layer: the data access layer. We push a relatively new approach for designing a system and call it *UX-first*. It is a task-based methodology that leads to commands and domain events starting from agreed-upon mockups and screens. In a task-based design philosophy, the role of the domain model is much less central and the same data access layer is just part of the infrastructure, and it's not necessarily based on canonical relational tables. However, the most forceful chapter in Part II—the one we recommend everybody read—is Chapter 5, "Discovering the domain architecture." In a nutshell, the chapter makes the point that only a deep understanding of the domain can lead to discovering an appropriate architecture. And, maybe more importantly, the resulting architecture doesn't have to be a single, top-level architecture for the entire application. As you recognize subdomains, you can model each to subapplications and give each the most effective architecture. As weird as it might sound, this is the core lesson of Domain-Driven Design (DDD).

Part III, "Supporting architectures," covers three supporting architectures you can use to build the various subdomains you recognized. For each architecture, we have a couple of chapters—an introduction and an implementation. The first supporting architecture we consider is the Domain Model. Next we head to Command/Query Responsibility Segregation (CQRS) and event sourcing.

Finally, Part IV, "Infrastructure," contains a single chapter—it deals with infrastructure and the persistence layer. This is interesting because it's not simply a chapter about SQL, Entity Framework, and relational databases. We primarily talk polyglot persistence, NoSQL data stores, and services used to hide storage details.

So, in the end, what's this book about?

It's about what you need to do and know to serve your customers in the best possible way as far as the .NET platform is concerned. The patterns, principles, and

techniques we describe are valid in general and are not specific to complex line-of-business applications. A good software architecture helps control the complexity of the project. And controlling complexity and favoring maintainability are the best strategies we have for fighting the canonical Murphy's Law of technology: "Nothing ever gets built on schedule or within budget." To get there, there's just one thing you're not allowed to fail on: understanding (deeply) the business domain.

Who should read this book

Software architects are the ideal audience for this book, but lead developers and developers of any type of .NET applications will find this book beneficial. Everyone who wants to be an architect should find this book helpful and worth the cost.

Is this book only for .NET professionals? Although all chapters have a .NET flavor, most of the content is readable by any software professional.

Assumptions

Strong object-oriented programming skills are a requirement for using this book. A good foundation in using the .NET platform and knowledge of some data-access techniques will also help. We put great effort into making this book read well. It's not a book about abstract design concepts, and it's not a classic architecture book either, full of cross-references and fancy strings in square brackets that hyperlink to some old paper listed in a bibliography at the end of the book.

This book might not be for you if...

This book might not be for you if you're seeking a reference book to pick up to find out how to use a given pattern. Instead, our goal is sharing and transferring knowledge so that you know what to do at any point. Or, at least, you know what two other guys—Dino and Andrea—would do in an analogous situation. This is (hopefully) a book to read from cover to cover and maybe more than once. It's not a book to keep on the desk for random reference only.

Downloads: Code samples

In the book, we present several code snippets and discuss sample applications with the primary purpose of illustrating principles and techniques for readers to apply in their own projects. In a certain way, we tried to teach fishing but we aren't providing sample fish to take home. However, there's a CodePlex site we want to point you to:

http://naa4e.codeplex.com/

There you find a few Visual Studio 2013 projects, one for each of the supporting architectures we describe in the book. A sample online store system—the I-Buy-Stuff project—is written according to the Domain Model architecture and then ported to CQRS. Two more projects complete the set: a live-scoring application and a mini-ERP system that illustrates event sourcing.

We invite you to follow the project because we plan to add more demos in the future.

The sample code has a few dependencies on common technologies such as Visual Studio 2013 and SQL Server. Projects make use of Entity Framework, ASP.NET MVC, RavenDB, Bootstrap and WURFL. Everything is linked to the project through Nuget. Refreshing the packages ensures you're able to reproduce the demo. In particular, you don't need a full installation of SQL Server; SQL Express will suffice.

Acknowledgments

When Andrea and I wrote the first edition of this book back in the summer of 2008 it was a completely different world. The huge economic downturn that hit the United States and other parts of the world, and still bites Europe, was just on the horizon. And Entity Framework was still to come. We covered patterns and technologies that are much less relevant today, and there was no cloud, mobile, or NoSQL. Still, at several times we caught the book ranked among Amazon's Top 10 in some category as long as four or five years after publication. For a technical book, lifespan of five years is like a geological era. We've been asked several times to work on a second edition, but the right astral conjunction never arrived until the spring of 2014. So here we are. Thanks to Devon Musgrave, Steve Sagman, Roger LeBlanc, and Carol Dillingham—a wonderful team.

As emphatic and partisan as it may sound, in over 20 book projects so far, I never left a lot of work for technical reviewers to do. And I hardly learned much from peer reviewers—for whatever reason. Well, this time it was different. Cesar De la Torre

Llorente—our peer reviewer—did a fantastic job. He promptly caught issues with the outline and content, even deep issues that I missed entirely and Andrea just perceived as glitches that were hard to explain in detail and fix. Cesar convinced us to restructure the content several times, reshaping the book to what it needed to be, and leading it to become what, we think, it should be.

Finally, I wish to reserve a word or two for some people that shared—sometimes without even realizing it—insights and remarks as valuable as gold. One is Hadi Hariri for his constantly updated vision of the IT world. Another is Jon Smith for reminding us of the many facets of the architect's role. Yet another is Giorgio Garcia-Agreda for conveying to me some of the innate attitude for problem solving (especially in harsh conditions). Last, but not least, my thanks also go to Roberto Raschetti. He may wonder why he deserves this accolade, but without doubt, he showed me the way, from the time I was a freshly graduated student to that huge project we had in store for months to come.

Finally, Mom, Dad—this is another one for your bookshelf! Sure you don't need a bigger one?

PS: Follow us on Facebook (facebook.com/naa4e) and tweet using #naa4e.

—Dino

This book would not exist without Dino. Dino is the one who caused me to accept the daunting task of writing a sequel to a book that, to my dismay, revealed itself to be a huge hit and applied a lot of pressure on me.

Never again is what you swore the time before.

Dino approached me several times to ask my feelings about writing a second edition, had the patience to accept a bunch of refusals, and then, after getting me committed, understood that this writing had to be a "two-paces process" because it would take time for me to write what he could, in just a few hours, put into elegant, insightful words.

Not only am I slow at writing, but I'm quite fussy. But Dino has always been very supportive in my struggle to make sure that this book would be at least as good as the previous edition, and then some.

I'm taking a ride with my best friend.

Being as fussy as I am, I was really pleased to have Cesar De la Torre Llorente as our peer reviewer because he did a fantastic job, not only at reviewing the contents,

but also at giving us valuable advice about how to restructure our content. Thank you Cesar. We really owe you a lot.

He knows where's he's taking me, taking me where I want to be.

But for this book to exist and, in my opinion, be a good one, we still needed a wonderful team behind us, and that's where the support we got from Devon Musgrave, Steve Sagman, Roger LeBlanc, and Carol Dillingham really made a difference. Thank you guys!

This is real fun, this is fun.

Writing a book while being a full-time consultant meant devoting a lot of time to being in front of your PC instead of being free and with the people you love, and that could be quite frustrating for both sides. Nevertheless, Laura and mum understood how important this book was for me and bestowed me with terrific support. And love.

You're like an angel and you give me your love, and I just can't seem to get enough of.

Finally, I want to thank all the guys at Managed Designs: without the experience gained because of all our endeavors, this book would not be half as good.

My secret garden's not so secret anymore!

And last, but not least, thank you Helen and Maruska for having been there when I struggled for words. Thank you from the bottom of my heart, miladies.

Welcome to my world, step right through the door.

PS: Follow us on Facebook (facebook.com/naa4e) and tweet using #naa4e.

—Andrea

Errata, updates, & book support

We've made every effort to ensure the accuracy of this book and its companion content. If you discover an error, please submit it to us via *mspinput@microsoft.com*. You can also reach the Microsoft Press Book Support team for other support via the same alias. Please note that product support for Microsoft software and hardware is not offered through this address. For help with Microsoft software or hardware, go to *http://support.microsoft.com*.

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

http://aka.ms/mspressfree

Check back often to see what is new!

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://aka.ms/tellpress

We know you're busy, so we've kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

The authors will be maintaining a Facebook page at *facebook.com/naa4e*.

Please precede comments, posts, and tweets about the book with the #naa4e hashtag.

This page intentionally left blank

CHAPTER 5

Discovering the domain architecture

Essentially, all models are wrong, but some are useful.
—George P. O. Box

Software definitely stems from mathematics and is subject to two opposing forces: the force of just doing things and the force of doing things right. We like to think that software is overall a big *catch-me-if-you-can* game. Visionary developers riding on the wings of enthusiasm quickly build a prototype that just works. The prototype then becomes a true part of the business; sometimes what originally was just a prototype changes and expands the business. Next, more down-to-earth developers join in to analyze, stabilize, consolidate or, in some cases, rewrite the software as it should have been done the first time, in accordance to theoretical principles.

Software, however, cannot be constrained into too-formal and rigid theorems.

Software mirrors real life, and life follows well-known rules defined in the context of some model. Unfortunately, at some given time, *T1*, we find that our understanding of the model is in some way limited. At some later time, *T2*, our understanding might deepen and we become aware of extended rules that better explain the overall model.

That's how things go in the real world out there; but it's not always how we make things go in real software.

Software architects tend to restrict the solution within the boundaries of a fixed, top-level architecture. We have done it ourselves many times. If we look back, we find it is a common error to start defining some top-level architecture (for example, Client/Server, Layered Architecture, Hexagonal) and then use it everywhere in our business application. That approach might work in the end, but the working solution you reach descends from the wrong approach. It's a sort of technical debt you are going to end up paying at some point.

Because software is expected to mirror real life, as a software architect you should first understand the segment of the real world you are modeling with software. That segment of the real world is the business domain, and it might contain multiple business contexts, each calling for its own ideal architecture.

The real added value of domain-driven design

Domain-Driven Design, or DDD for short, is a particular approach to software design and development that Eric Evans introduced over a decade ago. The essence and details of DDD are captured in the book *Domain-Driven Design*, which Evans wrote for Prentice Hall back in 2003. The book subtitle transmits a much clearer message about the purpose of DDD: *Tackling Complexity in the Heart of Software*.

When it debuted, DDD was perceived as an *all-or-nothing* approach to application design. You were given a method, some quite innovative guidelines, and the promise that it would work. Using DDD was not actually cheating, and we dare say that it really fulfilled the promise of "making it work"—except that it works only if you do it right. Doing it right is not immensely hard; but it's also immensely easy to do it wrong. (See Figure 5-1.)

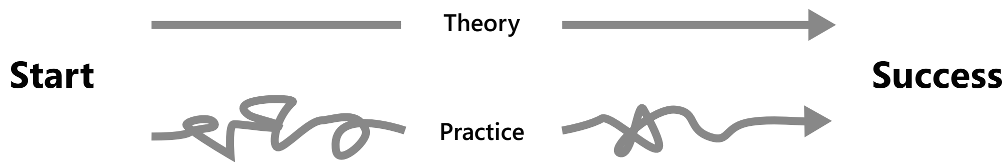


FIGURE 5-1 The DDD road to success is not always as smooth and easy as you expect.

What makes DDD so powerful but also so error prone? We think it's the *context*.

DDD is about crunching knowledge about a given business domain and producing a software model that faithfully mirrors it. The business domain is how a company conducts its own business: it's about organization, processes, practices, people, and language. The business domain lives in a context. Even very similar businesses may live in different contexts.

DDD is easy and powerful to use if you crunch enough knowledge and can model faithfully. DDD is painful and poor if you lack knowledge or fail to turn your knowledge into a model that fits the business domain.

What's in DDD for me?

However you want to frame it, DDD represents a significant landmark in software development. Not coincidentally, DDD initially was developed in the Java space, where the adoption of advanced design techniques (in the beginning of the last decade) was much faster and more widespread than in the .NET space. For many years, the scope and relevance of DDD was not really perceived in the .NET space.

Do we really need it? That was the question we are often asked and have asked ourselves many times.

DDD is not right for every project because it requires mastery and might have high startup costs. At the same time, nothing in DDD prevents you from using it in a relatively simple system. As we see it, the crucial point to using DDD is understanding where its real value lies and learning techniques to take advantage of it. The two biggest mistakes you can make with DDD are jumping on the DDD bandwagon just because it sounds cool, and stubbornly ignoring DDD because in the end you think your system is only a bit more complex than a plain CRUD.

In summary, we think DDD has two distinct parts. You always need one and can sometimes happily ignore the other.

DDD has an *analytical* part that sets out an approach to express the top-level architecture of the business domain in terms of bounded contexts. In addition, DDD has a *strategic* part that relates to defining a supporting architecture for the identified bounded contexts.

The real added value of DDD lies in using the analytical part to identify bounded business contexts. Next, the strategic design might or might not be leveraged to implement any of the bounded contexts.

Conducting analysis using DDD

The analytical part of DDD consists of two correlated elements: the *ubiquitous language* and *bounded contexts*.

The ubiquitous language is a vocabulary shared by all parties involved in the project and thoroughly used throughout the projects, ideally in all forms of spoken and written communication. As an architect, you typically populate the vocabulary of verbs and nouns as you acquire knowledge about the domain. This is the most common approach to starting to populate the vocabulary. More generally, you should also carefully look into adverbial phrases you find in requirements, because they might reveal a lot about the domain, such as events, processes, and triggers of processes.

The ubiquitous language is also the template that inspires the names and structure of the classes you end up writing. The ubiquitous language serves to improve and speed up the acknowledgment of requirements and simplify communication between parties so that they avoid misunderstandings, flawed assumptions, and botched translations when moving from one set of jargon to another.

Initially, there's just one ubiquitous language and a single business domain to understand and model. As you come to understand the requirements and explore the domain further, you might discover some overlap between nouns and verbs and find that they have different meanings in different areas of the domain. This might lead you to think the original domain should be split into multiple subdomains.

Bounded context is the term used with DDD to refer to areas of the domain that are better treated independently because of their own ubiquitous language. Put another way, you recognize a new bounded context when the ubiquitous language changes. Any business domain is made of contexts, and each context is shaped by logical contours. The primary responsibility of a software architect is identifying business contexts in a domain and defining their logical contours.

Context mapping is an expression often used to refer to the analytical part of DDD. Context mapping is a universal technique that can be applied to nearly any software scenario. Context mapping builds a high-level view of the domain from the perspective of a software architect. It shows subdomains and their relationships and helps you make strategic decisions.

Strategic model design

Coupled with context mapping is *strategic model design*. Once you identify the various bounded contexts, your next problem is determining the best architecture for each. DDD offers a recommended architecture in the form of the layered architecture and Domain Model. The term *domain model* here is subject to interpretation and deserves a bit of attention.

In the definition of DDD that Evans gives in his seminal book, the term *domain model* gives a nod to the Domain Model pattern formalized by Martin Fowler: *http://martinfowler.com/eaaCatalog/domainModel.html*. It consists of special flavor of an object model (also known as the domain model or entity model) and a set of domain service classes. More recently, the internal structure of the domain model is being reconsidered within the community. While seeing the domain model as an ad hoc collection of objects is still the most common perspective, a functional vision of it is gaining ground. Functional programming is, in fact, in many ways preferable to object-orientation for implementing tasks and expressing business concepts.

In the end, we can rephrase the whole thing today by saying that DDD suggests a layered architecture designed around a model of the domain. The model is mostly an object model, but it can be other things too—for example, a collection of functions. The persistence of data also depends on the structure of the model. It might require an O/RM tool if the model is a collection of objects; it might even be based on stored procedures invoked from idiomatic wrapper components if the model is, for example, function-based.

In the next chapters, we'll explore in depth the most common scenario for the domain model—when it takes the form of a special object model.

Note According to the original definition given by Evans, DDD is in a way the next natural step for developers versed in object-oriented design (OOD). The first principle of OOD recommends finding "pertinent classes," as you saw in Chapter 3, "Principles of software design." DDD recommends that you model the domain carefully—and that you model the domain carefully by discovering pertinent classes.

The phase of strategic model design consists of evaluating the various architectural options and choosing the architecture for each bounded context. Beyond the layered architecture, with a domain model there are usually other options such as a plain CRUD, a CMS (when the bounded context is expected to be a website), or even more sophisticated things, such as event sourcing (which we'll talk about in upcoming chapters.

Which parameters should drive your choice?

Overall, we think that today there's only one guiding rule, and it's based on the (carefully) estimated lifetime of the software you are about to write. Let's go through a few scenarios.

Fast-food applications

Suppose you are writing a short-term, one-off application such as a survey web application or some analogous set of pages aimed at collecting raw data for your analysts. You know that the expected lifetime is very short and that after the expected data has been collected the app will be bluntly dismissed.

Does it really make sense to invest more than the least amount of time that could possibly make it work? It probably doesn't.

So you can go with the quickest possible CRUD you can arrange, whether it is by using Web Forms, Silverlight, or plain HTML, depending on the skills and the target audience. If you are about to think something like, "Hey, I'm a senior architect, and no boss would pay my time for such trivial problems," well, you are probably just experiencing the power of bounded contexts already. Taken out of context, a fast-food application is undoubtedly a very basic—even silly—example. But it might be just one bounded context of a much larger and more complex domain that you, as a senior architect, helped to map.

Front-end websites

The project you're on requires a web front end. It has a sufficiently complex back end, where a lot of business rules must be taken into account and a bunch of external web services must be coordinated, but at the end of the day the front end is a plain set of read-only pages with zero or limited forms. The most important requirement you have for it is, "It must be shockingly cool and engage people."

Web Forms can be immediately ruled out because of its limited flexibility due to server controls. ASP.NET MVC is a much better option because it allows full control of HTML and can be effectively styled with CSS. Should you really go with an ASP.NET MVC solution from scratch?

Couldn't a CMS be a quicker and equally cool solution?

We can probably hear the same objections—it's a silly example for a book that claims to target software architects. Yes, but a software architect recognizes complexity where she sees it and doesn't create any unnecessary complexity.

You might know that plugins can extend a CMS, like WordPress, to do almost anything you can think of. It's not a far-fetched idea to just get a cool WordPress theme and a bunch of plugins, including custom plugins, to do the job.

Again, it's a matter of opportunity and skills. It's a matter of context.

Any other types of applications

As Thomas Edison used to say, the value of an idea lies in the using of it. So *make-the-code-just-work* is a common approach, especially in these hectic days of emerging ideas and startups. The *make-the-code-just-work* motto is fine if you don't need to touch the code once it's done and it works.

No matter what the customer might say and no matter what the current plans are, there's just one reason that any software avoids further changes: it gets dismissed. If the software is not expected to be dismissed in just a few months, as an architect you better consider a more thoughtful approach than just fast-food code. And Domain Model and the other supporting architectures we're slated to discuss in the upcoming chapters are the best options available for simplifying code maintenance.

Note This is simply a gentle reminder that, not coincidentally, maintainability is both an OOD design goal and a class of requisites according to ISO 9126.

The ubiquitous language

Requirements are always communicated, but we all know that making sense of user requirements is sometimes hard. In addition, the inability to completely comprehend user requirements is probably the primary cause of misunderstandings between business and development teams.

As a way to mitigate the risks of misunderstandings at any time, Eric Evans suggested the use of a common language that he called the *ubiquitous language*.

Purpose of the ubiquitous language

We write software for a specific business, but we're software architects in the end and we might not be black-belt experts of the specific business domain. Likewise, domain experts might have some working knowledge of software development but probably not enough to avoid misunderstandings and incorrect assumptions.

Developers and domain experts often just speak different languages, and each group has its own jargon. Furthermore, it is not unlikely that different business people involved—say, from different departments—might use different jargon and give the same term different meanings. The language barrier might not preclude business from taking place, but it certainly makes any progress much slower than expected and acceptable. Translation from one language to another must be arranged; jargon expressions must be adjusted and put into context. This not only takes time, but it also introduces the risk of losing details along the way.

The purpose of the ubiquitous language is to define a common terminology shared by all involved parties—managers, end users, developers, and stakeholders in general—and at all levels—spoken and written communication, documentation, tests, and code. A common language reduces the need for translating concepts from the business to the development context, promotes clarity, and minimizes assumptions.

Once defined, the ubiquitous language becomes the official, all-encompassing language of the project.

Structure of the ubiquitous language

The ubiquitous language is not the raw language of the business, nor is it the language of the development teams. Both business and development language are forms of jargon, and both languages, if taken literally, might lack or skim over essential concepts, as well as generate misunderstandings and communication bottlenecks.

Figure 5-2 shows the canonical diagram used in literature to indicate the relationship between ubiquitous language and native languages spoken by domain experts and development teams.

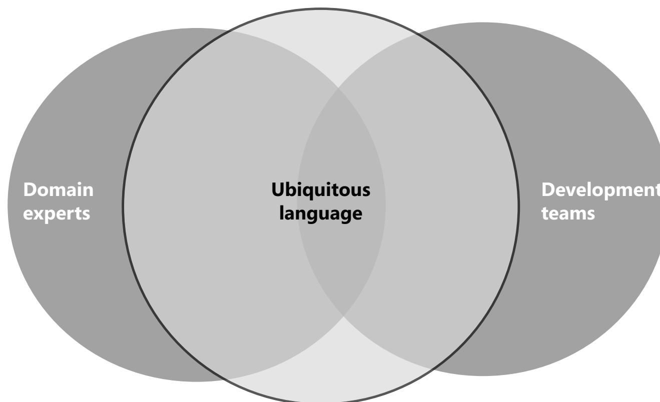


FIGURE 5-2 The canonical diagram that illustrates how the ubiquitous language unifies domain and technical concepts and extends them.

The figure shows that the ubiquitous language is a combination of domain and technical jargon. However, the ubiquitous language is expected to contain, for the most part, words and verbs that reflect the semantics of the business domain rather than technology terms.

It is not limited to the business jargon, however. While technical concepts like caching data, invoking a service, and deleting records of a database should not be part of the language, terms that indicate persistent actions, the response of the system, or notifications sent or received might be necessary to make the resulting language faithfully express the final behavior of the system.

The ubiquitous language exists in the subsoil of the domain. The architect must dig it out at the beginning of the project.

How to define the ubiquitous language

The ubiquitous language is not artificially created in a lab and then submitted for approval to involved parties. Quite the reverse—the language emerges out of interviews and meetings and gets its final shape iteratively along the way. It might take several steps of refinement and adjustment before the language flows as expected and faithfully expresses the reality of the system being built.

The first draft of the language commonly results from acknowledging requirements as architects rewrite and make sense of raw requirements collected during elicitation.

As a technical person, you should expect the language to be rigorous (for example, strictly unambiguous and consistent), fluent, and made of simple elements that can be combined to compose more sophisticated concepts and actions.

As a domain expert, you should hesitate to accept any terms and concepts the language might contain that are unknown in the domain and that are not clearly referring to a process or business concept. Also, as a domain expert, you should ensure that all relevant business terms are defined in the language and, more importantly, are given the right meaning. For example, if the language contains the term *account*, the term must refer to the meaning that *account* has in the domain space.

Note In general, the ubiquitous language contains business terms (nouns and verbs) plus new terms (mostly verbs) that more or less directly map to technical actions, such dealing with databases, cache and security, services and so forth. The number of nonbusiness concepts, however, should be kept to a minimum.

The ubiquitous language is the official language of the project, and the vocabulary of terms is inspired and then validated by domain experts. Everything in the project, from documentation to actual code, is permeated by the language.

How would you physically express and save the vocabulary of the ubiquitous language? In practical terms, it consists of a glossary of terms and expressions saved to a Microsoft Word or Microsoft Excel document or even some UML diagrams. Each term is fully explained in a way that makes it understandable to both domain experts and developers.

It should be the responsibility of the team to keep the glossary up to date throughout the project. The ubiquitous language, in fact, is anything but static. It can change and evolve over time to reflect new insights gained about the domain.

Important There are two main scenarios where the analytical part of DDD excels. One is when there's really a lot of domain logic to deal with that is tricky to digest, distill, and organize. Having a ubiquitous language here is key because it ensures that all terms used are understood and that no other terms are used to express requirements, discuss features, and write code.

Another scenario is when the business logic is not completely clear because the actual business is being built and the software is just part of the initial effort. Startups are an excellent example of this scenario. In this case, the domain logic is being discovered and refined along the way, making the availability of a ubiquitous language a great benefit to understand where one is and where the business can move forward.

Keeping language and model in sync

Naming and coding conventions used in the domain model should reflect naming conventions set in the ubiquitous language. This relationship should not vary during the lifetime of the project.

If the language changes because of a different level of understanding, or a new requirement, then the naming and coding conventions in the domain model should be updated. Also, the opposite is true to a large extent, in the sense that renaming a class or a method is always possible but doing so should require approval if the class or method is pervasive and the change affects a key term of the language. At the same time, it nearly goes without saying that if a given class or method exists only to serve implementation purposes, the constraint doesn't apply and you can rename it without restrictions.

Let's briefly look at an example that refers to the code we'll be examining in more detail in Chapter 8, "Introducing the domain model," and beyond. The domain is an online store, and the use-case we focus on is the placement of an order.

Each order is reasonably associated with a record in a table and with a column that indicates the current state. The order ID is used to track ordered items from another table. Processing the order requires first a check to see if there are pending or delayed payments from the same customer. Next, the process requires a check on goods in store and, finally, a call to the shipping and payment web services and the creation of a new order record.

Here's a more domain-driven way of expressing the same use-case. As you can see, the description is more concise and uses fewer technical details. The terms used should also reflect the jargon used within the organization. Here's an example.

As a registered customer of the I-Buy-Stuff online store, I can redeem a voucher for an order I place so that I don't actually pay for the ordered items myself.

There are a few business terms here—registered customer, order, order items, and voucher. There are also actions, such as placing an order and redeeming a voucher. All these belong to the ubiquitous language glossary. In particular, the term *voucher* here is the term used in the business, and once it is added to the ubiquitous language, nobody will ever think of using synonyms (such as coupon, gift card, credit note and so forth).

When coding the use-case, a developer will likely create a class to access the database table of orders, and instances of the order class will be materialized from the database and saved back there. However, these are just technical details that don't belong in the business context. As such, those details should be buried in the folds of the implementation, limited to technical meetings between developers, and never surface in official communication with business people.

This is the essence of the ubiquitous language.

Bounded contexts

In the beginning, you assume one indivisible business domain and start processing requirements to learn as much as possible about it and build the ubiquitous language. As you proceed, you learn how the organization works, which processes are performed, how data is used and, last but not least, you learn how things are referred to.

Especially in a large organization, the same term often has different meanings when used by different people, or different terms are used to mean the same thing. When this happens, you probably crossed the invisible boundaries of a subdomain. This probably means that the business domain you assumed to be one and indivisible is, in reality, articulated in subdomains.

In DDD, a subdomain in the problem space is mapped to a bounded context in the solution space.

A bounded context is an area of the application that requires its own ubiquitous language and its own architecture. Or, put another way, a bounded context is a boundary within which the ubiquitous language is consistent. A bounded context can have relationships to other bounded contexts.

Important Subdomains and bounded contexts are concepts that sometimes appear to be similar and can be confusing. However, both concepts can be easily understood by looking at the difference between a domain and domain model, which is probably easier to grasp. The *domain* represents the problem to solve; the *domain model* is the model that implements the solution to the problem. Likewise, a *subdomain* is a segment of the domain, and a *bounded context* is a segment of the solution.

Discovering contexts

Without flying too high conceptually, consider a simple booking system. The front-end web site is certainly a subdomain. Is it the only one? Most likely, the system needs a back-office panel to put content on the site and perhaps extract statistics. This probably makes for another subdomain.

In the current draft of the top-level architecture, we have two candidate bounded contexts.

There are two additional aspects vital to investigate: the boundaries of each bounded context and their relationships.

Marking boundaries of contexts

Sometimes it's relatively easy to split a business domain into various subdomains, each representing a bounded context to render with software.

But is it splitting or is it partitioning? There is a huge difference between the two.

In the real world, you don't often see business domains that can be easily partitioned in child domains with nearly no overlapping functions and concepts. So in our experience, it is more a case of

splitting than just partitioning. The problems with splitting a business domain are related to marking the boundaries of each context, identifying areas of overlap, and deciding how to handle those areas.

As mentioned, the first concrete clue that you have a new subdomain is when you find a new term used to express a known concept or when the same term is found to have a second meaning. This indicates some overlapping between subdomains. (See Figure 5-3.)

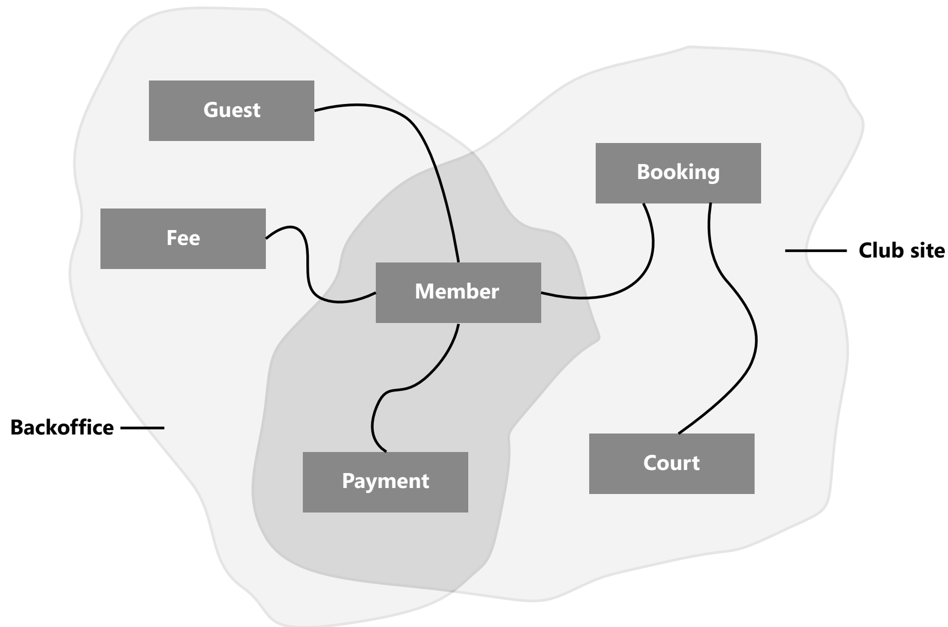


FIGURE 5-3 Two business contexts with some overlapping.

The business domain is made of a subdomain (say, *club site*) that, among other features, offers the booking of courts. The booking of courts involves members and payments. The back office is a distinct but related subdomain. Both subdomains deal with members and payments, even though each has a different vision of them.

The first decision to be made is whether you need to treat those subdomains separately and, if so, where you draw the boundaries.

Splitting a domain into bounded contexts

Working on a single, all-encompassing model is always dangerous, and the level of complexity grows as the number of entities and their relationships grow. The resulting graph can be crowded; entities and related code can become quite coupled, and it doesn't take much to serve up the perfect Big Ball of Mud.

Splitting is always a good idea, especially when this leads you to creating software subsystems that reflect the structure of the organization. The back-office system, for example, will be used by different people than the club site.

Let's say you go for distinct bounded contexts.

How would you deal with overlapping logic? The concept of "club member" exists in both contexts, but in the back-office context the club member has nearly no behavior and is a mere container of personal and financial data. In the club-site context, on the other hand, the member has some specific behaviors because she can book a court or add herself to an existing booking. For doing so, the *Member* entity will just need an ID, user name, and possibly an email address.

In general, having a single, shared definition of an entity will have the side effect of padding the definition with details that might be unnecessary in some of the other contexts. With reference to Figure 5-3, family members are not necessary to book a court from the club site, but they are relevant to calculating the yearly fee.

The fundamental point to resolve when conceptual overlapping is detected is which of the following options is more appropriate:

- A single bounded context that includes all entities
- A distinct bounded context with a shared kernel of common entities
- A distinct bounded context with distinct definitions of common entities

The options are graphically summarized in Figure 5-4.

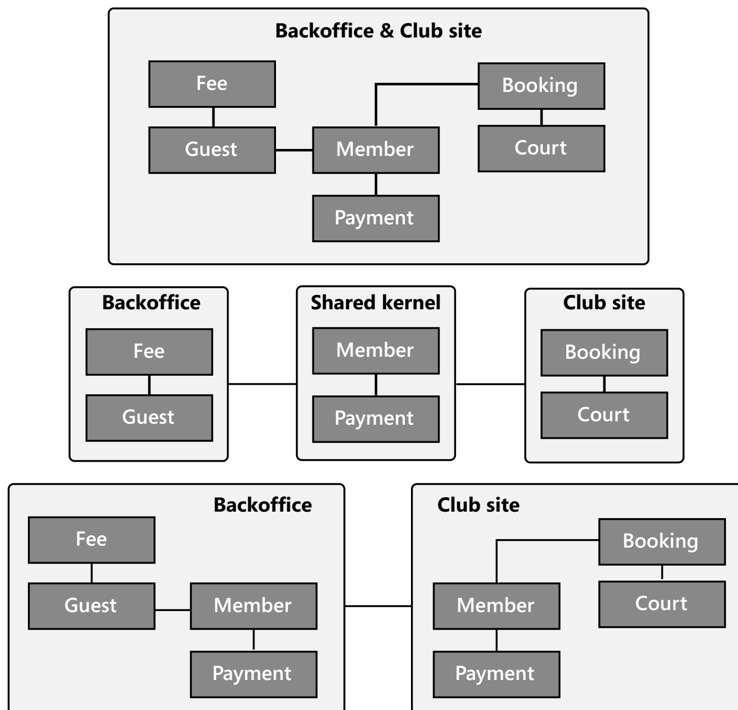


FIGURE 5-4 Resolving the conceptual overlapping of contexts.

There's also a fourth option. Is the entire model entirely inadequate and in need of refinement so that in the end you can have partitions instead of subsets?

That's what it means to mark the boundaries of bounded contexts.

By the way, it's not us dodging the issue by not taking a clear stand on a particular option. It's that, well, it just depends. It depends on other information about the domain. It depends on time and budget. It depends on skills. It also depends on your personal view of the domain.

That's what makes it so fun to mark the boundaries of bounded contexts.

Bounded context and the organization

The number of contexts and relationships between bounded contexts often just reflect the physical organization of the enterprise. It is common to have a bounded context for each business department such as human resources, accounting, sales, inventory, and the like.

Different development teams are typically assigned to each bounded context, and different artifacts are generally produced, scheduled, and maintained.

The overlapping of concepts is quite natural in business domains; speaking in general, the best way to handle such overlapping is to use different bounded contexts, as shown in the third option of Figure 5-4.

Just sharing entities between development teams, as a common kernel, might prefigure risky scenarios, where changes of team 1 might break the code of team 2 and compromise the integrity of the model. Shared kernels work great if an effective shared kernel exists—such as different organizations just using the same entities.

Otherwise, it's the first step toward a true mess.

Context mapping

Bounded contexts are often related to each other. In DDD, a *context map* is the diagram that provides a comprehensive view of the system being designed. In the diagram, each element represents a bounded context. The diagrams in Figure 5-4 are actually all examples of a context map.

Relational patterns

Connections between elements of a context map depict the relationship existing between bounded contexts. DDD defines a few relational patterns.

Relational patterns identify an *upstream* context and *downstream* context. The upstream context (denoted with a *u*) is the context that influences the downstream and might force it to change. Denoted with *d*, the downstream context is passive and undergoes changes on the upstream context. Table 5-1 lists DDD relational patterns.

TABLE 5-1 DDD relational patterns

DDD relational pattern	Description
Anticorruption layer (ACL)	Indicates an extra layer of code that hides to the downstream context any changes implemented at some point in the upstream context. More on this later.
Conformist	The downstream context just passively conforms to whatever model the upstream context comes up with. Typically, the conformist pattern is a lighter approach than ACL and the downstream context also receives data it might not need.
Customer/Supplier	Two contexts are in a classic upstream/downstream relationship, where the supplier is the upstream. The teams, however, work together to ensure that no unnecessary data is sent. This aspect marks the difference with Conformist.
Partnership	Two contexts are developed independently; no code is shared, but both contexts are upstream and downstream at the same time. There's a sort of mutual dependency between the two, and one can't just ignore the other for delivery and change.
Shared kernel	Two contexts share a subset of the model. Contexts are therefore tightly coupled, and no team can change the shared kernel without synchronizing with the other team.

Figure 5-5 is the graphical representation of a context map. Each block represents a bounded context. The Sales block is connected to the upstream External Service block, and an ACL ensures that changes in the service don't force changes in the Sales context. The upstream and downstream contexts are labeled with the *u* and *d* marks.

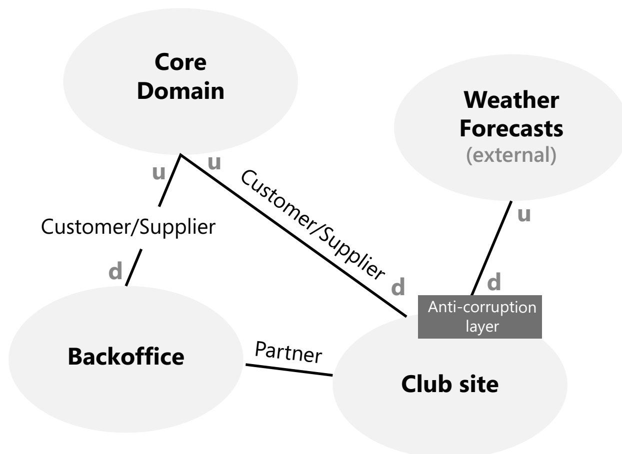


FIGURE 5-5 A sample context map showing some of the DDD relational patterns.

Context mapping is part of the strategic design of the solution. It doesn't produce code or deployable artifacts, but it can be immensely helpful to grab a better understanding of the system.

Note Many DDD experts advise that because software ultimately mirrors the structure of business organizations, a context map should ideally reflect the organization of the enterprise. Sometimes, it turns out that the ideal context map for the system to build doesn't actually reflect the real organization. When this happens—and it does happen—well, things are not going to be easy!

Anticorruption layers

Relationships between bounded contexts pose the problem of how the development of one context influences the other over time. The safest way of dealing with related contexts is by creating an anticorruption layer (ACL).

It's the safest way because all the changes required to keep the contexts in sync when one undergoes changes are isolated in the anticorruption layer, as shown in Figure 5-6.

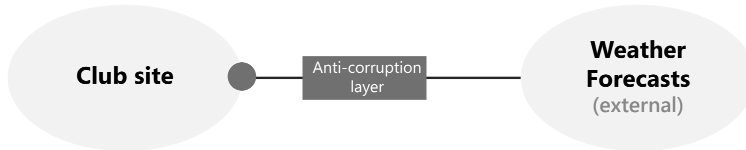


FIGURE 5-6 The anticorruption layer is an interfacing layer that separates two connected contexts.

The interface that the ACL exposes to the downstream context (the club site in this case) is an invariant. The ACL, in fact, absorbs the changes in the upstream context (Weather Forecasts service in this case) and does any conversion work that might be required. Updating the ACL when the upstream context changes usually requires less work and is less obtrusive than updating the club-site context.

The ACL is particularly welcome when one of the bounded contexts encapsulates a chunk of legacy code or just an external service that none of the teams building the system has control over.

Giving each context its own architecture

Each bounded context is a separate area of the overall application. You are forced to use DDD strategic modeling to implement each bounded context, and not only because you identified the bounded context using a DDD methodology. As an architect, you should validate the context map and then focus on each context separately.

For example, the Core Domain area of the application might be implemented using a Domain Model approach. The club-site context can be an ASP.NET MVC application with a layered back end that uses an application layer on top of MVC controllers. The application layer uses services in the Core Domain context for changing the state of the application. Finally, a simpler subsystem like Back Office can be efficiently given a data-driven design and result in a simple two-layer architecture with only presentation and data access. (Concretely, this could be a Web Forms application using DataGrids.)

Another option might be separating the front end of the club site from, say, the booking module. You could use ASP.NET MVC for the booking module and a CMS (for example, WordPress) for the few pages with news, photos, and static content.

Mixing multiple supporting architectures in the realm of a single system is far from wrong.

Common supporting architectures

The process of identifying business contexts already reveals a lot about the nature of the domain and subdomains. To an expert eye that knows about technologies and frameworks, a good candidate solution appears immediately for a given context.

Just as a quick glossary, Table 5-2 lists the most commonly used supporting architectures you might find in the industry.

TABLE 5-2 A list of supporting architectures.

Supporting architecture	Brief description
Multilayer architecture	Canonical segmentation based on presentation, business, and data layers. The architecture might come in slightly different flavors, such as an additional application layer between the presentation and business layers and with the business layer transformed into a domain layer by the use of a DDD development style. Layered architecture is just another name for a multilayer architecture. We'll be using the term layered architecture instead of multilayer in the rest of this chapter and throughout the book.
Multitier architecture	Segmentation that is in many ways similar to that of a multilayer architecture except that now multiple tiers are involved instead of layers. (More on the possible downsides of a layer-to-tier mapping in a moment.)
Client/server architecture	Classic two-layer (or two-tier) architecture that consists only of presentation plus data access.
Domain Model	Layered architecture based on a presentation layer, an application layer, a domain layer, and an infrastructure layer, designed in accordance with the DDD development style. In particular, the model is expected to be a special type of object model.
Command-Query Responsibility Segregation (CQRS)	Two-fold layered architecture with parallel sections for handling command and query sides. Each section can be architected independently, even with a separate supporting architecture, whether that is DDD or client/server.
Event sourcing	Layered architecture that is almost always inspired by a CQRS design that focuses its logic on events rather than plain data. Events are treated as first-class data, and any other queryable information is inferred from stored events.
Monolithic architecture	The context is a standalone application or service that exposes an API to the rest of the world. Typical examples are autonomous web services (for example, Web API host) and Windows services. Yet another example is an application hosting a SignalR engine.

As we write this chapter, another architectural style is gaining in popularity: *micro-services*. At first, micro-services don't sound like a completely new idea and are not really presented like that. There's a lot of service-oriented architecture (SOA) in micro-services, such as the fact that services are autonomous and loosely coupled. However, micro-services also explicitly call out for lightweight HTTP mechanisms for communication between processes. For more information on micro-services, you can check out the Martin Fowler's site at *http://martinfowler.com/articles/microservices.html*.

The reason why we mention micro-services here is that, abstractly speaking, the overall idea of micro-services weds well with identifying business contexts, discovering relationships, and giving each its own architecture and autonomous implementation. Micro-services, therefore, can be yet another valid entry in Table 5-2.

Layers and tiers might not be interchangeable

Layers and tiers are not the same. A *layer* is a logical container for different portions of code; a *tier* is a physical container for code and refers to its own process space or machine. All layers are actually deployed to a physical tier, but different layers can go to different tiers.

That's precisely the point we want to raise here.

In Table 5-2, we listed multilayer architecture and multitier architecture. Admittedly, they look the same except that one separates blocks of code logically and the other physically. We suggest, however, that you consider those architectures as different options to be evaluated individually to see if they fit in the solution.

The error that many system integrators made in the past was to deploy a multilayer architecture as a multitier architecture. In doing so, they matched layers to tiers one-to-one. This led to segregating in different tiers the presentation layer of a Web Forms application and the business layer using WCF, Web services or even, in the old days, .NET Remoting. It apparently looked like a better architecture, but it created latency between tiers and had a deep impact on the performance of the system. In addition, system maintenance (for example, deploying updates) is harder and more expensive in a multitier scenario.

Tiers are heavy but can be used to scale the application. However, just having tiers doesn't automatically ensure your application is faster. Generally speaking, we tend to prefer the deployment of the entire application stack on a single tier, if that's ever possible.

The layered architecture

In the rest of this chapter, we'll provide an overview of the layered architecture—the multilayer architecture introduced in Evans' book about DDD. The layered architecture is probably the most common type of architecture that results from DDD analysis.

Origins of the layered architecture

In the 1990s, most computing scenarios consisted of one insanely powerful server (at least for the time it was) and a few far slower personal computers. Software architecture was essentially client/server: the client focused on data presentation and commands, and the server mostly implemented persistence. Any business logic beyond basic CRUD (Create, Read, Update, Delete) was stuffed in stored procedures to further leverage the capacity of the insanely powerful server machine.

Over the years, we all managed to take larger and larger chunks of the business logic out of stored procedures and place them within new components. This originated the classic (up to) three-segment model, which is shown in Figure 5-7. Note that the figure also shows a direct connection between the presentation and data layers in memory of SQL data binding.

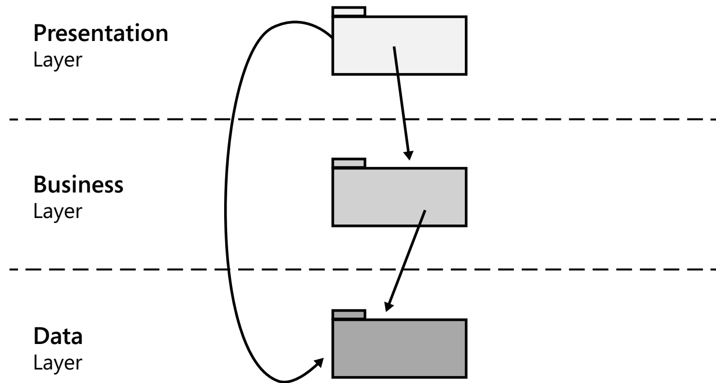


FIGURE 5-7 The classic three-segment architecture.

Note that we're using the term *segment* here as a general way to interchangeably refer to both tiers and layers.

From what we have seen, learned, and done ourselves, we'd say that the largest share of systems inspired by the three-segment architecture is actually implemented as a layered system deployed on two physical tiers. For a website, for example, one tier is the ASP.NET application running within the Internet Information Services (IIS) process space and another was the Microsoft SQL Server service providing data. (See Figure 5-8.)

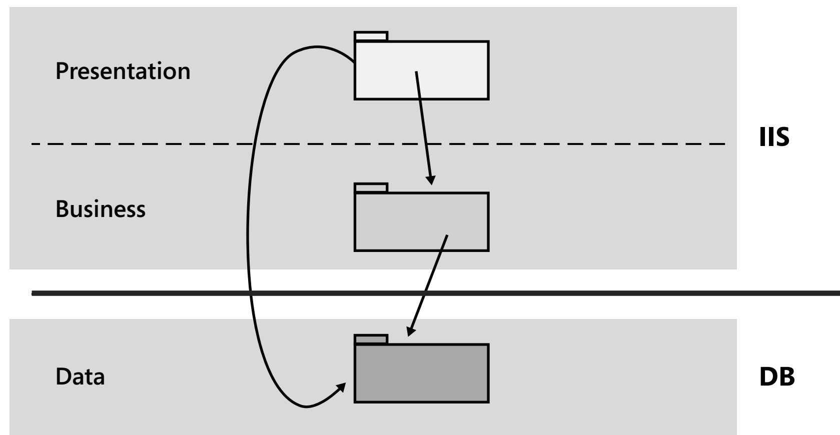


FIGURE 5-8 Common deployment of a multilayer ASP.NET application.

For the most part, the data model of any three-segment architecture is the relational data model of the data store. The growing complexity of applications has led developers to a more conceptual view of that data. As patterns like the *Domain Model* and approaches like Domain-Driven Design (DDD) were developed and exercised, the internal structure of a layered architecture evolved quite a bit. (See Figure 5-9.)

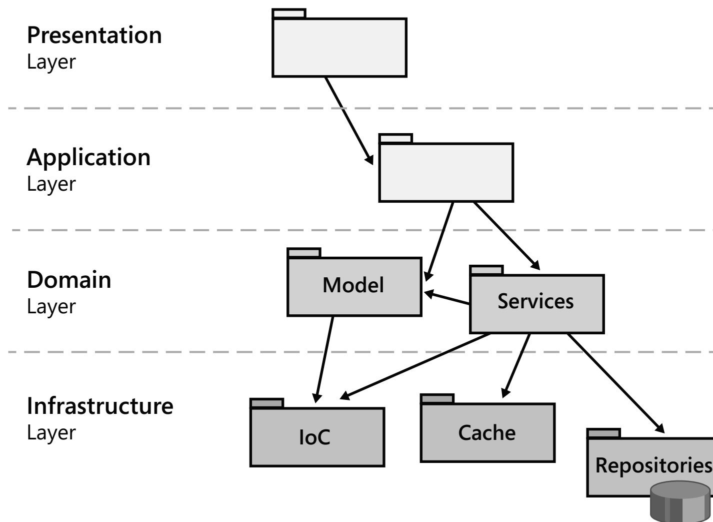


FIGURE 5-9 A more modern version of a layered architecture.

Roughly speaking, the presentation layer is the same in both architectures and the infrastructure layer includes the data layer of Figure 5-7 but is not limited to that. The infrastructure layer, in general, includes anything related to any concrete technologies: data access via O/RM tools, implementation of IoC containers, and the implementation of many other cross-cutting concerns such as security, logging, caching, and more.

The business layer exploded into the application and domain layer. Upon a more thoughtful look, the layered architecture of Figure 5-9 results from a better application of the *separation of concerns* (SoC) principle. In systems inspired by the schema of Figure 5-7, the actual business logic is sprinkled everywhere, mostly in the business logic but also in the presentation and data layers.

The layered architecture of Figure 5-9 attempts to clear up such gray areas.

Note Repositories are generally placed in the domain layer as far as their interfaces are concerned. The actual implementation, however, usually belongs to the infrastructure layer.

Presentation layer

The presentation layer is responsible for providing some user interface (UI) to accomplish any tasks. Presentation is a collection of screens; each screen is populated by a set of data and any action that starts from the screen forwards another well-defined set of data.

Generally speaking, we'll refer to any data that populates the presentation layer as the *view model*. We'll refer to any data that goes out of the screen triggering a back-end action as the *input model*. Although a logical difference exists between the two models, most of the time the view model and input model coincide. (See Figure 5-10.)

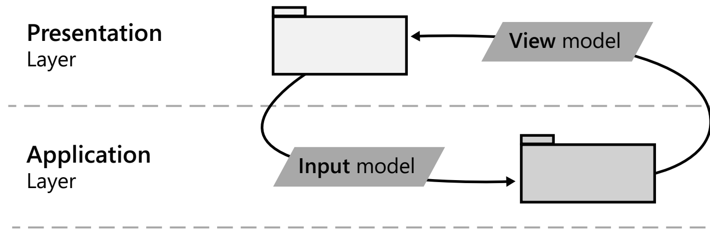


FIGURE 5-10 Describing the data that goes into and out of presentation screens.

Application layer

As we see it, the application layer is an excellent way to separate interfacing layers such as presentation and domain. In doing so, the application layer contributes immensely to the clarity of the entire design. In the past, a typical gray area of many architectures was the placement of the part of the business code that needed to be aware of the presentation.

The application layer is the additional layer that reports to the presentation and orchestrates any further business action. The application layer is where you orchestrate the implementation of use-cases.

Entry point in the system's back end

Each interactive element of the user interface (for example, buttons) triggers an action in the back end of the system. In some simple scenarios, the action that follows some user's clicking takes just one step to conclude. More realistically, instead, the user's clicking triggers something like a workflow.

According to Figure 5-9, the application layer is the entry point in the back end of the system and the point of contact between the presentation and back end. The application layer consists of methods bound in an almost one-to-one fashion to the use-cases of the presentation layer. Methods can be grouped in any way that makes sense to you.

For example, in an ASP.NET MVC application, we expect the application layer classes to go hand in hand with controllers. The *HomeController* class, therefore, will have injected some *HomeControllerService* worker class. Here's a quick sample:

```
public class HomeController
{
    private readonly IHomeControllerService _service;
    public HomeController(IHomeControllerService service)
    {
       _service = service;
    }
    public ActionResult Index()
    {
       var model = _service.FillHomePage( /* input model */ );
       return View(model);
    }
    ...
}
```

The mechanism of injection can happen at your leisure. It can happen via Unity or any other Inversion of Control (IoC) container, or it can be done through poor man's dependency injection, as shown here:

```
public class HomeController
{
    private readonly IHomeControllerService _service;
    public HomeController() : this(new HomeControllerService())
    {
    }
    public HomeController(IHomeControllerService service)
    {
       _service = service;
    }
}
```

In a nutshell, the application layer is responsible for the implementation of the application's use-cases. All it does is orchestrate tasks and delegate work to other layers down the stack.

We think there can be two flavors of an application layer: inside or outside the business logic. Neither is preferable to the other; it's all about how you envision the system.

Orchestrating the business logic

In general, the application layer is bound one-to-one to the presentation with the notable exception of unattended systems. The structure of the layer is driven by the actionable controls in the various user interface screens. With reference to Figure 5-5, this flavor of application layer lives in the club-site context and orchestrates workflows involving components in the Core Domain and external services. Key aspects of this application layer are these:

- It might or might not be consumable by different front ends because, for example, a mobile front end might have slightly different use-cases than the web front end or ends.
- It can be stateful at least as far the progress of a UI task is concerned.
- It gets its input from the presentation and sends a view model back as shown in Figure 5-10.

The application layer holds references to the domain layer and infrastructure layer. Finally, the application layer has no knowledge of business rules and doesn't hold any business-related state information.

As the name suggests, the application layer is just application specific. If our aforementioned booking system must be consumed by a website and a mobile application, we're going to have two distinct application layers—one on the site and one either within the mobile application or exposed as an HTTP service. Are these layers actually different?

Well, they might be different; and if they're different, different layers should be implemented. To continue with the booking example, the application layer of the website might redirect to the credit card site to pay and then proceed with the persistence of the booking data. The application layer of

the mobile app might use in-app payment features or just use stored payment information and pass them along in some way to the domain layer.

Note The DDD jargon uses the term *application services* to refer to services that sit atop the domain layer and orchestrate business use-cases.

Domain layer

The domain layer hosts the entire business logic that is not specific to one or more use-cases. In other words, the domain layer contains all business logic that remains once you have compiled the application layer.

The domain layer consists of a model (known as the *domain model*) and possibly a family of services. The nature of the model can vary. Most of the time, it is an entity-relationship model, but it can be made of functions too. Let's stick to what appears to be the most common scenario. So let's say that at the end of the day an *entity model* is an object model.

However, in an entity model, constituent classes usually follow certain conventions. We'll return in detail to domain modeling and DDD conventions for entities in upcoming chapters. For now, it suffices to say that entities in the model are expected to expose both data and behavior. A model with entities devoid of any significant behavior—that is, merely data structures—form an *anemic domain model*.

The ultimate goal of a domain model is to implement the ubiquitous language and express the actions that business processes require. In this regard, exposing some behavior tends to be more relevant than holding some data.

Along with an entity model, the domain model layer features domain services.

Domain services are pieces of domain logic that, for some reason, don't fit into any of the existing entities. A domain service is a class, and it groups logically related behaviors that typically operate on multiple domain entities. A domain service often also requires access to the infrastructure layer for read/write operations. In the aforementioned club-site context, a domain service can be the code to book a court:

```
void BookCourt(Court court, ClubMember member)
```

Court and *ClubMember* are domain entities, and the method *BookCourt* knows how to retrieve and apply due policies.

Infrastructure layer

The infrastructure layer is anything related to using concrete technologies, whether it is data persistence (O/RM frameworks like Entity Framework), specific security API, logging, tracing, IoC containers, caching, and more.

The most prominent component of the infrastructure layer is the persistence layer—which is nothing more than the old-faithful data access layer, only possibly extended to cover a few data sources other than plain relational data stores. The persistence layer knows how to read and/or save data.

The data can reside on a relational server as well as in a NoSQL data store or in both. The data can be accessible through web services (for example, CRM or proprietary services) or live in the file system, cloud, or in-memory databases such as Memcached, ScaleOut, or NCache.

Summary

We dare say that the software industry moved from one extreme to the other. Decades ago, writing software was inspired by the slogan "model first, code later." This led to considerable efforts to have a big comprehensive design up front. There's nothing wrong with an upfront design, except that it is like walking on water. It's definitely possible if requirements, like water, are frozen.

Maybe because of global warming, requirements hardly ever freeze these days. Subsequently, whomever embarks in an upfront design risks sinking after only a few steps.

Mindful of failures of upfront design, architects and developers moved in the opposite direction: code first, model later. This philosophy, although it is awkward, moves things ahead. It just works, in the end. Adopting this approach makes it hard to fix things and evolve, but it delivers working solutions as soon as possible, and if something was wrong, it will be fixed next. Like it or not, this model works. As our friend Greg Young used to write in his old posts, you should never underestimate the value of working software.

What we're trying to say, however, is that some middle ground exists, and you get there by crunching knowledge and deeply understanding the domain. Understanding the domain leads to discovering an appropriate architecture. However, it doesn't have to be a single, top-level architecture for the entire application. As you recognize subdomains, you can model each to subapplications, each coded with the most effective architecture. If it still sounds hard to believe, consider that it's nothing more than the old motto *Divide-et-Impera* that—historians report—helped Julius Caesar and Napoleon to rule the world.

Finishing with a smile

Developers sometimes enter into God mode and go off on a tangent, thus losing perspective on their code. We try to see models everywhere and to see each model as a special case. This makes software design cumbersome at times, and funny. For more tongue-in-cheek examples of Murphy's law beyond the following ones, have a look at *http://www.murphys-laws.com*:

- All generalizations are false, including this one.
- The weakest link is the most stable one.
- Never underestimate the value of working software.

This page intentionally left blank

CHAPTER 10

Introducing CQRS

Beware of false knowledge; it is more dangerous than ignorance.
—George Bernard Shaw

As discussed in Chapter 5, "Discovering the domain architecture," there are two distinct but interoperating parts in Domain-Driven Design (DDD). The analytical part is about discovering the top-level architecture, using the ubiquitous language to dig out bounded contexts and their relationships. The strategic part is about giving each bounded context the most appropriate architecture. A decade ago, the standard architecture for a bounded context was a layered architecture, with a domain layer made of an object-oriented, all-encompassing model and domain services. The effort of developers was then to crunch knowledge about the domain and render it through a web of interconnected objects with state and behavior. The model was unique and intended to fully describe the entire business domain.

It didn't look, in the beginning, like a thing that's far easier to say than do.

Some projects that embraced DDD eventually worked; other projects failed. Success stories can be told too, but many people still believe that DDD is hard to do even though it can possibly deliver significant benefits. The point is that, for many people, the perception that DDD holds benefits is much less concrete than the perception of the damage that might result from using DDD and failing.

Is there anything wrong with DDD?

The analytical part of DDD has little to do with code and software design. It's all about figuring out the top-level architecture while using ad hoc tools like the ubiquitous language. This is an excellent approach to take for just about any project. In complex scenarios, it helps to understand the big picture and lay out modules and services. In simple scenarios, it boils down to having just one context and a single module to build.

The critical part of the original vision of DDD is the suggested architecture for a bounded context. First and foremost, the layered architecture with an object-oriented model and services is just one option, and simpler solutions—for example, Content Management System (CMS), Customer Relationship Management (CRM), coupled CRUD, and two-layer systems—certainly are not banned as long as they fit the needs. Second, even when the layered architecture with a domain layer appears to be the ideal solution for a bounded context, the model doesn't have to be object-oriented, nor does it have to be an all-encompassing model for the entire context.

In this chapter, we introduce a pattern that splits the domain model in two, actually achieving much more than just separation of concerns.

Separating commands from queries

Most of the difficulties that early adopters of DDD faced were in designing a single model to take care of all aspects of the domain. Generally speaking, any actions performed on a software system belong to one of the following two categories: query or command. In this context, a *query* is an operation that doesn't alter in any way the state of the system and just returns data. The *command*, on the other hand, does alter the state of the system and doesn't return data, except perhaps for a status code or an acknowledgment.

The logical separation that exists between queries and commands doesn't show up clearly if the two groups of actions are forced to use the same domain model. For this reason, a new supporting architecture emerged in the past few years called CQRS, which is short for *Command/Query Responsibility Segregation*.

Generalities of the CQRS pattern

Since the days of ancient Rome, *Divide et Impera* has been an extremely successful approach to actually getting things done. Roman ruler Julius Caesar won a number of battles fighting against the entire enemy army, but when things got more complicated, he implemented a strategy of leading his enemy into dividing forces across the battlefields so that he could fight against a smaller army.

Similarly, the CQRS pattern is based on a simple, almost commonplace, idea: queries and commands (sometimes also referred to as *reads* and *writes*) are very different things and should be treated separately. Yet, for a long time, developers—like short-sighted commanders—insisted on having the same conceptual model for both queries and commands in their systems.

Especially in complex business scenarios, a single model soon becomes unmanageable. It doesn't just grow exponentially large and complex (and subsequently absorb time and budget), it also never ends up working the way it should.

Note We wouldn't be too surprised to find out at some point that developers insisted for years on having a single model because having two distinct models might seem like a negative statement about their ability to work out a single, all-encompassing model. Our egos grow big sometimes and obscures a proper perspective on things!

From domain model to CQRS

In a way, CQRS is a form of lateral thinking resulting from the difficulty of finding a well-conceived model for complex domains. If the Domain Model turns out to be expensive and objectively complex, are we sure we're approaching it right? That was probably the question that led to investigating and formalizing a different pattern.

At the end of the day, CQRS uses two distinct domain layers rather than just one. The separation is obtained by grouping operations that are queries in one layer and operations that are commands in another. Each layer, then, has its own architecture and its own set of services dedicated to only queries and commands, respectively. Figure 10-1 captures the difference.

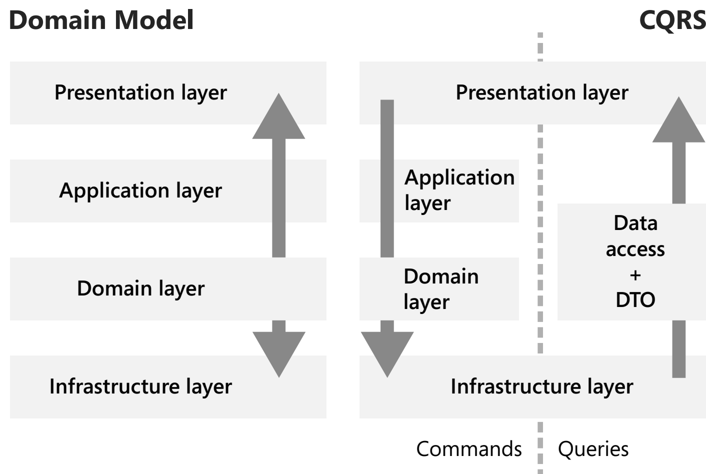


FIGURE 10-1 Visual comparison between Domain Model and CQRS.

In CQRS, it is not a far-fetched idea to have the query stack based exclusively on SQL queries and completely devoid of models, an application layer, and a domain layer. Having a full domain-model implementation in the query stack is not common. In general, the query stack should be simplified to the extreme. In addition, typically a CQRS approach has a different database for each side.

Structure of the query and command domain layers

As surprising as it might sound, the simple recognition that commands and queries are two different things has a deep impact on the overall architecture of the system. In Figure 10-1, we split the domain layer into two blocks.

Are they just two smaller and simpler versions of a domain layer like we discussed in the past two chapters?

The interesting thing is that with the architecture of the system organized as two parallel branches as shown in Figure 10-1, the requirement of having a full-fledged domain model is much less strict. For one thing, you might not need a domain model at all to serve queries. Queries are now just data to be rendered in some way through the user interface. There's no command to be arranged on queried data; as such, many of the relationships that make discovering aggregates so important in a classic domain model are unnecessary. The model for the domain layer of the query side of a CQRS system can be simply a collection of made-to-measure data-transfer objects (DTOs). Following this consideration, the domain services might become just classes that implement pieces of business logic on top of an anemic model.

Similar things can be said for the domain layer of the command side of the system. Depending on the commands you actually implement, a classic domain model might or might not be necessary. In general, there's a greater chance you might need a domain model for the command side because here you express business logic and implement business rules. At any rate, the domain model you might have on the command side of a CQRS system is likely far simpler because it is tailor-made for the commands.

In summary, recognizing that queries and commands are different things triggers a chain reaction that sets the foundation for domain modeling, as discussed in the past two chapters. We justified domain models as the ideal way to tackle complexity in the heart of software. Along the way, we ended up facing a good deal of complexity and thought it was, for the most part, complexity that is inherent to the business domain. Instead, most of that complexity results from the Cartesian product of queries and commands. Separating commands from queries can reduce complexity by an order of magnitude.

> **Note** Just in case you were wondering, a *Cartesian product* is a mathematical operation that, given two or more sets, returns a new and larger set made of all ordered pairs (or tuples), where each element belongs to a different set. The cardinality of the resulting set is the product of the cardinalities of all input sets.

CQRS is not a top-level architecture

Unlike DDD, CQRS is not a comprehensive approach to the design of an enterprise-class system. CQRS is simply a pattern that guides you in architecting a specific bounded context of a possibly larger system. Performing a DDD analysis based on a ubiquitous language and aimed at identifying bounded contexts remains a recommended preliminary step.

Next, CQRS becomes a valid alternative to Domain Model, CRUD, and other supporting architectures for the implementation of a particular bounded context.

Benefits of CQRS

The list of benefits brought about by using CQRS to implement a bounded context is not particularly long. Overall, we think that there are essentially two benefits. Their impact on the solution, though, is dramatic.

Simplification of the design

As our interactions with the Domain Model taught us, most of the complexity you face in a software system is usually related to operations that change the state of the system. Commands should validate the current state and determine whether they can run. Next, commands should take care of leaving the system in a consistent state.

Finally, in a scenario in which reading and writing operations share the same representation of data, it sometimes becomes hard to prevent unwanted operations from becoming available in reading or writing. We already raised this point in the last chapter when we pointed out that it's nearly impossible to give, say, a list of order items a single representation that fits in both the query and command scenarios. Anyway, we'll return to this aspect in a moment with a detailed example.

We stated in an earlier note that the complexity of the Domain Model results from the Cartesian product of queries and commands. If we take the analysis one step further, we can even measure by a rule of thumb the amount of reduced complexity. Let's call *N* the complexity of queries and commands. In a single domain model, where requirements and constraints of queries affect commands and vice versa, like in a Cartesian product, you have a resulting complexity of *NxN*. By separating queries from commands and treating them independently, all you have is N+N.

Potential for enhanced scalability

Scalability has many faces and factors; the recipe for scalability tends to be unique for each system you consider. In general, scalability defines the system's ability to maintain the same level of performance as the number of users grows. A system with more users performs certain operations more frequently. Scalability, therefore, depends on the margins that architects have to fine-tune the system to make it perform more operations in the same unit of time.

The way to achieve scalability depends on the type of operations most commonly performed. If reads are the predominant operation, you can introduce levels of caching to drastically reduce the number of accesses to the database. If writes are enough to slow down the system at peak hours, you might want to consider switching from a classic synchronous writing model to async writes or even queues of commands.

Separating queries from commands gives you the chance to work on the scalability aspects of both parts in total isolation.

Note A good example of what it means to treat reads and writes separately is given by a cloud platform such as Microsoft Azure. You deploy query and command layers as distinct web or worker roles and scale them independently, both in terms of instances and the size of each instance. Similarly, when reads are vastly predominant, you can decide to offload some pages to a distinct server with a thick layer of caching on top. The ability to act on query and command layers separately is invaluable. Note also that the investment in Microsoft Azure is focused more on websites, WebJobs, and even mobile services rather than the original web roles and worker roles. In particular, WebJobs is more of a lightweight approach than web roles and worker roles.

Pleasant side effects of CQRS

A couple of other pleasant side effects of CQRS are worth noting here. First, CQRS leads you to a deep understanding of what your application reads and what it processes. The neat separation of modules also makes it safe to make changes to each without incurring some form of regression on one or the other.

Second, thinking about queries and commands leads to reasoning in terms of tasks and a task-based user interface, which is very good for end users.

Fitting CQRS in the business layer

Honestly, we don't think there are significant downsides to CQRS. It all depends in the end on what you mean exactly by using CQRS. So far we just defined it as a pattern that suggests you have two distinct layers: one filled with the model and services necessary for reading, and one with the model and services for commands. What a *model* is—whether it is an object model, a library of functions, or a collection of data-transfer objects—ultimately is an implementation detail.

With this definition in place, nearly any system can benefit from CQRS and coding it doesn't require doing things in a different way. Neither does it mean learning new and scary things.

Noncollaborative vs. collaborative systems

The point, however, is that CQRS also induces some deeper architectural changes that maximize the return in terms of scalability and reduced complexity but that require some investment in learning and performing a preliminary analysis.

In the end, CQRS was discovered by looking for more effective ways to tackle complex systems in which multiple actors—both end users and software clients—operate on the data concurrently and sophisticated and ever-changing business rules apply. The major proponents of CQRS—Udi Dahan and Greg Young—called these systems *collaborative systems*.

Let's try to formalize the landmarks of a collaborative system a bit better.

In a collaborative system, the underlying data can change at any time from the effect of the current user, concurrent users connected through various front ends, and even back-end software. In a collaborative system, users compete for the same resources, and this means that whatever data you get can be stale in the same moment that it is read or even long before it is displayed. One of the reasons for this continuous change is that the business logic is particularly complex and involves multiple modules that sometimes need to be loaded dynamically. The architect has two main options:

- Lock the entire aggregate for the time required to complete any operation.
- Keep the aggregate open to change at the cost of possibly showing out-of-sync data that eventually becomes consistent. (This is often referred to as *eventual consistency*.)

The first option is highly impractical for a collaborative system—the back end would be locked while serving a single request at nearly any time, and the throughput would be very low. The second option might be acceptable but, if the system is not properly fine-tuned, it can end up giving inaccurate results and taking too long a time to respond.

This is the scenario that led to formalizing CQRS.

CQRS to the rescue

CQRS is not simply about using different domain layers for queries and commands. It's more about using distinct stacks for queries and commands architected by following a new set of guidelines. (See Figure 10-2.)

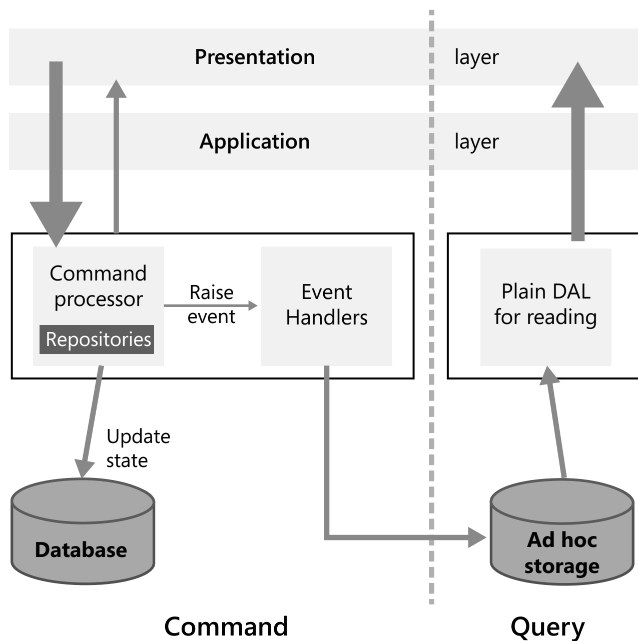


FIGURE 10-2 The big picture of the CQRS implementation of a collaborative system.

In the command pipeline, any requests from the presentation layer become a command appended to the queue of a processor. Each command carries information and has its own handler that knows about the logic. In this way, each command is a logical unit that can thoroughly validate the state of the involved objects and intelligently decide which updates to perform and which to decline. The command handler processes the command just once. Processing the command might generate events handled by other registered components. In this way, other software can perform additional tasks. One of the common tasks is performing periodical updates of a database cache that exists for the sole purpose of the query pipeline.

When the business logic is extremely sophisticated, you can't afford to handle commands synchronously. You can't do that for two reasons:

- It slows down the system.
- The domain services involved become way too complex, perhaps convoluted and subject to regression, especially when rules change frequently.

With a CQRS architecture, the logic can be expressed through single commands that result in distinct, individual components that are much easier to evolve, replace, and fix. In addition, these commands can be queued if necessary.

Note In a CQRS scenario, one-way commands that do not return any response do not conceptually exist. They should be modeled as events fired for one or more event handlers to handle.

The query pipeline is quite simple, on the other hand. All it has is a collection of repositories that query content from ad hoc caches of denormalized data. The structure of such database cache tables (most of the time, plain Microsoft SQL Server tables) closely reflects the data required by the user interface. So, for example, if a page requires the customer name while displaying the order details, you can arrange to have the ID and name readymade in a cache without having to JOIN every time. Furthermore, because the query pipeline is separated, it can offload to a dedicated server at any time.

CQRS always pays the architecture bill

Many seem to think that outside the realm of collaborative systems, the power of CQRS diminishes significantly. On the contrary, the power of CQRS really shines in collaborative systems because it lets you address complexity and competing resources in a much smoother and overall simpler way. There's more to it than meets the eye, we think.

In our opinion, CQRS can sufficiently pay your architecture bills even in simpler scenarios, where the plain separation between query and command stacks leads to simplified design and dramatically reduces the risk of design errors. You don't need to have super-skilled teams of developers to do CQRS. Quite the opposite: using CQRS enables nearly any team to do a good job in terms of scalability and cleanliness of the design.

Transaction script in the command stack

CQRS is a natural fit in a system dependent on collaboration. However, the benefits of command/query separation can apply to nearly all systems.

Most systems out there can be summarized as "CRUD with some business logic around." In these cases, you can just use the Transaction Script (TS) pattern (as discussed in Chapter 7, "The

mythical business layer") in the implementation of the command stack. TS is an approach that has you partition the back end of the system—overall, business logic—in a collection of methods out of a few container classes. Each method essentially takes care of a command and provides a full implementation for it. The method, therefore, takes care of processing input data, invoking local components or services in another bounded context, and writing to the database. All these steps take place in a single "logical" transaction.

As Fowler said, the glory of TS is in its simplicity. TS is a natural fit for applications with a small amount of logic. The major benefit of TS is there's only minor overhead for development teams in terms of learning and performance.

Note CQRS suggests—or just makes it reasonable sometimes—to use distinct databases for reading and writing. When this happens, the adoption of TS in the organization of the business logic raises the problem of figuring out the ideal way to handle eventual consistency. We'll return to this point in the next chapter about CQRS implementation.

EDMX for the read model

What's the easiest way to build a data access layer that serves the purposes of the presentation layer with no extra whistles and bells? Once you know the connection string to the database to access, all you do is create an Entity Framework wrapper in the form of an EDMX designer file in Microsoft Visual Studio.

Running the Entity Framework designer on the specified connection string infers an object model out of the database tables and relationships. Because it comes from Entity Framework, the object model is essentially anemic. However, the C# mechanism of partial classes enables you to add behavior to classes, thus adding a taste of object orientation and domain modeling to the results.

Arranging queries—possibly just LINQ queries—on top of this object model is easy for most developers, and it's effective and reliable. Expert developers can work very quickly with this approach, and junior developers can learn from it just as quickly.

The pragmatic architect's perspective

Taking the point of view of an architect, you might wonder what the added value of CQRS is in relatively simple systems with only a limited amount of business logic.

You set the architecture to define the boundaries of command and query stacks. You pass each developer an amount of work that is commensurate to that developer's actual skills. You still have distinct stacks to be optimized independently or even rewritten from scratch if necessary.

In a word, an expert architect has a far better chance to take on the project comfortably, even with only junior developers on the team.

The query stack

Let's delve a bit deeper into the two pipelines that make up the CQRS architecture. In doing so, another key aspect that drives the adoption of CQRS in some highly collaborative systems will emerge clearly—the necessity of dealing with stale data.

The read domain model

A model that deals only with queries would be much easier to arrange than a model that has to deal with both queries and commands. For example, a prickly problem we hinted at in Chapter 8, "Introducing the domain model," is brilliantly and definitely solved with the introduction of a read-only domain model.

Why you need distinct models

The problem was summarized as follows. The *Order* class has an *Items* property that exposes the list of ordered products. The property holds inherently enumerable content, but which actual type should you use for the *Items* property? The first option that probably comes to mind is *IList<T>*. It might work, but it's not perfect. So let's put ourselves in a Domain Model scenario and assume we want to have a single model for the entire domain that is used to support both queries and commands. Also, let's say we use a plain list for the *Items* property:

```
public IList<OrderItem> Items { get; private set; }
```

The private setter is good, but it prevents only users of an *Order* from replacing it. Any code that gets an instance of *Order* can easily add or remove elements from it. This might or might not be a legitimate operation; it depends on the use-case. If the use-case is managing the order, exposing order items through a list is just fine. If the use-case is showing the last 10 orders, a list is potentially dangerous because no changes to the order are expected.

Important The domain model is the API of the business domain. Once publicly exposed, an API can be invoked to perform any action it allows. To ensure consistency, the API should not rely on developers to use it only the right way. If Murphy (of "Murphy's laws") were a software engineer, he would say something like, "If a developer can call an API the wrong way, he will."

On the other hand, if you expose the list as a plain enumeration of order items, you have no way to create an order and add items to it. In addition, individual items are still modifiable through direct access:

```
public IEnumerable<OrderItem> Items { get; private set; }
```

Things don't change even if you use *ReadOnlyCollection<T>* instead of *IEnumerable*. A Microsoft .NET Framework read-only collection is read-only in the sense that it doesn't allow changes to the

structure of the collection. Furthermore, if the read-only collection is created as a wrapper for a regular list, changes to the underlying list do not affect the read-only wrapper. Here's an example where order items are exposed as a read-only collection but methods still make it possible to populate the collection:

```
public class Order
{
    private readonly IList<OrderItem> _items;
    public Order()
    {
        _items = new List<MOrderItem>();
    }
    public ReadOnlyCollection<OrderItem> Items
    {
        get
        {
            return new ReadOnlyCollection<OrderItem>(_items);
        }
    }

    public void Add(int id, int quantity)
    {
        _items.Add(new OrderItem(id, quantity));
    }
}
public class OrderItem
{
    public OrderItem(int id, int quantity)
    {
        Quantity = quantity;
        ProductId = id;
    }
    public int Quantity { get; /*private*/ set; }
    public int ProductId { get; /*private*/ set; }
}
```

However, direct access to elements in the collection is still possible—whether it is gained during a *for-each* loop, out of a LINQ query, or by index:

```
foreach (var i in order.Items)
{
    i.Quantity ++;
    Console.WriteLine(i);
}
```

To prevent changes to the data within the collection, you have to make the setter private.

This would work beautifully if it weren't for yet another possible issue. Is it worthwhile to turn the *OrderItem* entity of the domain model into an immutable object?

Classes in the domain model are modified and made more and more complex because they can be used interchangeably in both query and command scenarios. Using the read-only wrapper, ultimately, is the first step toward making a read version of the *Order* entity.

Note We are not trying to say that having *Items* coded as a list is dangerous; instead, we just want to point out a consistency hole and a sort of violation of the syntax rules of the ubiquitous language. The order displayed for review is not the order created out of a request. This is what CQRS is all about.

From a domain model to a read model

When your goal is simply creating a domain model for read-only operations, everything comes easier and classes are simpler overall. Let's look at a few varying points.

The notion of *aggregates* becomes less central, and with it the entire notion of the domain model as explained in Chapter 8. You probably still need to understand how entities aggregate in the model, but there's no need to make this knowledge explicit through interfaces.

The overall structure of classes is more similar to data-transfer objects, and properties tend to be much more numerous than methods. Ideally, all you have are DTOs that map one-to-one with each screen in the application. Does that mean that model becomes anemic? Well, the model is 100 percent anemic when made of just data. An *Order* class, for example, will no longer have an *AddItem* method.

Again, there's no issue with CQRS having a 100 percent anemic read model. Methods on such classes can still be useful, but only as long as they query the object and provide a quick way for the presentation or application layer to work. For example, a method *IsPending* on an *Order* class can still be defined as follows:

```
public bool IsPending()
{
   return State == OrderState.Pending;
}
```

This method is useful because it makes the code that uses the *Order* class easier to read and, more importantly, closer to the ubiquitous language.

Designing a read-model façade

The query stack might still need domain services to extract data from storage and serve it up to the application and presentation layers. In this case, domain services, and specifically repositories, should be retargeted to allow only read operations on the storage.

Restricting the database context

In the read stack, therefore, you don't strictly need to have classic repositories with all CRUD methods and you don't even need to expose all the power of the *DbContext* class, assuming you're in an Entity Framework Code-First scenario, as described in Chapter 9, "Implementing the domain model," and as it will be used in future chapters.

In Chapter 9, we had a class wrapping the Entity Framework *DbContext* and called it *DomainModelFacade*. The structure of the class is shown here:

```
public class DomainModelFacade : DbContext
{
   public DomainModelFacade() : base("naa4e-09")
   {
      Products = base.Set<Product>();
      Customers = base.Set<Customer>();
      Orders = base.Set<Order>();
   }

   public DbSet<Order> Orders { get; private set; }
   public DbSet<Customer> Customers { get; private set; }
   public DbSet<Product> Products { get; private set; }
   ...
}
```

The *DbSet* class provides full access to the underlying database and can be used to set up queries and update operations via LINQ-to-Entities. The fundamental step toward a query pipeline is limiting the access to the database to queries only. Here are some changes:

```
public class ReadModelFacade : DbContext
{
   public ReadModelFacade() : base("naa4e-09")
   {
      Products = base.Set<Product>();
      Customers = base.Set<Customer>();
      Orders = base.Set<Order>();
   }

   public IQueryable<Customer> Customers
   {
       get { return _customers; }
   }

   public IQueryable<Order> Orders
   {
       get { return _orders; }
   }

   public IQueryable<Product> Products
   {
       get { return _products; }
   }
   ...
}
```

Collections to query from the business logic on are now exposed via *IQueryable* interfaces. We said that the notion of aggregates loses focus in a read model. However, queryable data in the read-model façade mostly corresponds to aggregates in a full domain model.

Adjusting repositories

With a read-model façade, any attempt to access the database starts with an *IQueryable* object. You can still have a set of repository classes, populate them with a bunch of *FindXxx* methods, and use them from domain services and the application layer.

In doing so, you'll certainly run into simple situations such as just needing to query all orders that have not been processed two weeks after they were placed. The *FindXxx* method can return a collection of *Order* items:

```
IEnumerable<Order> FindPendingOrderAfter(TimeSpan timespan);
```

But there are also situations in which you need to get all orders whose total exceeds a threshold. In this case, you need to report order details (like ID, date of creation, state, payment details) as well as customer details (at least the name and membership status). And, above all, you need to report the total of the order. There's no such type in the domain; you need to create it. OK, no big deal: it's just a classic DTO type:

```
IEnumerable<OrderSummary> FindOrdersBeyond(decimal threshold);
```

All is good if the *OrderSummary* DTO is general enough to be used in several repository queries. If it is not, you end up with too many DTO classes that are also too similar, which ultimately also poses a problem with names. But beyond the name and quantity of DTOs, there's another underlying issue here: the number of repository methods and their names and implementation. Readability and maintainability are at stake.

A common way out is leaving only common queries as methods in the repositories that return common DTOs and handling all other cases through predicates:

```
public IEnumerable<T> Find(Expression<Func<T, Boolean>> predicate)
```

In this case, though, you're stuck with using type T, and it might not be easy to massage any queried data into a generic DTO within a single method.

Important We decided to introduce relevant aspects of CQRS starting from a DDD perspective and then discuss issues that arise from using it, as well as what has been done to smooth out the rough spots according to the key guidelines of CQRS.

As far as repositories are concerned, the bottom line is that you don't likely need them in the query stack. The entire data access layer can be articulated through LINQ queries on top of some Object/Relational Mapper (O/RM) classes placed directly in the application layer. Also, a full-fledged O/RM like Entity Framework sometimes might be overkill. You might want to consider a micro O/RM for the job, such as PetaPoco. (See *http://www.toptensoftware.com/petapoco*.)

Looking ahead to .NET, a better option probably is the upcoming Entity Framework 7, which will be a lot more lightweight and aligned with ASP.NET vNext.

Layered expression trees

Over the past 20 years of developing software, we have seen a recurring pattern: when a common-use solution gets overwhelmingly complex and less and less manageable over time, it's probably because it doesn't address the problem well. At that point, it might be worth investigating a different approach to the problem. The different approach we suggest here to reduce the complexity of repositories and DTOs in a read model leverages the power of LINQ and expression trees.

Realistic scenarios

Let's focus first on a few realistic scenarios where you need to query data in many different ways that are heavily dependent on business rules:

- **Online store** Given the profile of the user, the home page of the online store will present the three products that match the profile with the highest inventory level. It results in two conceptual queries: getting all products available for sale, and getting the three products with the highest inventory level that might be interesting to the user. The first query is common and belongs to some domain service. The second query is application specific and belongs to the application layer.
- **ERP** Retrieve all invoices of a business unit that haven't been paid 30 days after their due payment terms. There are three conceptual queries here: getting all invoices, getting all invoices for the business unit, and getting all invoices for the business unit that are unpaid 30 days later. The first two queries are common and belong to some domain services. The third query sounds more application specific.
- **CMS** Retrieve all articles that have been published and, among them, pick those that match whatever search parameters have been specified. Again, it's two conceptual queries: one domain-specific and one application-specific.

Why did we use the term *conceptual* query?

If you look at it conceptually, you see distinct queries. If you look at it from an implementation perspective, you just don't want to have distinct queries. Use-cases often require queries that can be expressed in terms of filters applied over some large sets of data. Each filter expresses a business rule; rules can be composed and reused in different use-cases.

To get this, you have two approaches:

- Hide all filters in a repository method, build a single super-optimized query, run it, and return results. Each result is likely a different DTO. In doing this, you're going to have nearly one method for each scenario and new or modified methods when something changes. The problem is not facing change; the problem is minimizing the effort (and risk of regression) when change occurs. Touching the repository interface is a lot of work because it might have an impact on upper layers. If you can make changes only at the application level, it would be much easier to handle and less invasive.
- Try LINQ and expression trees.

Let's see what it takes to use layered expression trees (LET).

Using *IQueryable* as your currency

The idea behind LET is enabling the application layer to receive *IQueryable<T>* objects wherever possible. In this way, the required query emerges through the composition of filters and the actual projection of data is specified at the last minute, right in the application layer where data is being used to generate the view model for the presentation to render.

With this idea in mind, you don't even need repositories in a read model, and perhaps not even as a container of common queries that return direct and immediately usable data that likely will not be filtered any more. A good example of a method you might still want to have in a separate repository class is a *FindById*.

You can use the public properties of the aforementioned read façade as the starting point to compose your queries. Or, if necessary, you can use ad hoc components for the same purpose. In this way, in fact, you encapsulate the read-model façade—still a point of contact with persistence technology—in such components. Here's what the query to retrieve three products to feature on the home page might look like. This code ideally belongs to the application layer:

```
var queryProducts = (from p in CatalogServices.GetProductsAvailableForSale()
                        orderby p.UnitsInStock descending
                        select new ProductDescriptor
                        {
                           Id = p.Id,
                           Name = p.Name,
                           UnitPrice = p.UnitPrice,
                           UnitsInStock = p.UnitsInStock,
                        }).Take(3);
```

Here's another example that uses the recommended async version of LINQ methods:

```
var userName = _securityService.GetUserName();
var currentEmployee = await _database
         .Employees
         .AsNoTracking()
         .WhereEmployeeIsCurrentUser(userName)
         .Select(employee =>
             new CurrentEmployeeDTO
             {
                 EmployeeId = employee.Id,
                 FirstName = employee.PersonalInformation.FirstName,
                 LastName = employee.PersonalInformation.LastName,
                 Email = employee.PersonalInformation.Email,
                 Identifier = employee.PersonalInformation.Identifier,
                 JobTitle = employee.JobTitle,
                 IsManager = employee.IsTeamManager,
                 TeamId = employee.TeamId,
             }).SingleOrDefaultAsync();
currentEmployee.PictureUrl = Url.Link("EmployeePicture",
                                new { employeeId = currentEmployee.EmployeeId });
```

As you might have noticed, the first code snippet doesn't end with a call to *ToList*, *First*, or similar methods. So it is crucial to clarify what it means to work with *IQueryable* objects.

The *IQueryable* interface allows you to define a query against a LINQ provider, such as a database. The query, however, has deferred execution and subsequently can be built in multiple steps. No database access is performed until you call an execution method such as *ToList*. For example, when you query all products on sale, you're not retrieving all 200,000 records that match those criteria. When you add *Take(3)*, you're just refining the query. The query executes when the following code is invoked:

```
var featuredProducts = queryProducts.ToList();
```

The SQL code that hits the database has the following template:

```
SELECT TOP 3 ... WHERE ...
```

In the end, you pass the *IQueryable* object through the layers and each layer can add filters along the way, making the query more precise. You typically resolve the query in the application layer and get just the subset of data you need in that particular use-case.

Isn't LET the same as an in-memory list?

No, LET is not the same as having an in-memory list and querying it via LINQ-to-Objects. If you load all products in memory and then use LINQ to extract a subset, you're discarding tons of data you pulled out of the database.

LET still performs a database access using the best query that the underlying LINQ provider can generate. However, *IQueryable* works transparently on any LINQ provider. So if the aforementioned method *GetProductsAvailableForSale* internally uses a static list of preloaded *Product* instances, the LET approach still works, except that it leverages LINQ-to-Objects instead of the LINQ dialect supported by the underlying database access layer.

Using LET is not the same as having a static list, but that doesn't mean having a static list is a bad thing. If you see benefits in keeping, say, all products in memory, a static list is probably a good approach. LET is a better approach if the displayed data is read from some database every time.

Note Crucial to CQRS is the fact that the database you query might not be the core database where commands write. It can easily be a separate database optimized for reading and built to denormalize some of the content in the core database. This approach is often referred to as the "pure CQRS approach."

Upsides of LET

The use of LET has several benefits. The most remarkable benefit is that you need almost no DTOs. More precisely, you don't need DTOs to carry data across layers. If you let queries reach the application layer, all you do is fetch data directly in the view model classes. On the other hand, a view model is unavoidable because you still need to pass data to the user interface in some way.

Note As you might have noticed, we're using different names—DTO and view model classes—for two software entities that can be described using the same words: classes that just carry data. ASP.NET MVC view model classes are actually DTOs, and the reason we're using different names here is to emphasize that one of the benefits of LET is that you can forget about intermediate classes you might need in the classic Domain Model to carry data across layer. In CQRS with LET, all you need is LINQ to query data and a DTO to return data to the presentation. There are no other intermediaries—just LINQ queries and, in ASP.NET MVC, view model classes.

Another benefit is that the code you write is somehow natural. It's really like you're using the database directly, except that the language is much easier to learn and use than plain-old T-SQL.

Queries are DDD-friendly because their logic closely follows the ubiquitous language, and sometimes it seems that domain experts wrote the queries. Among other things, DDD-friendly queries are also helpful when a customer calls to report a bug. You look into the section of the code that produces unexpected results and read the query. You can almost read your code to the customer and quickly figure out whether the reason unexpected data is showing up on the screen is logical (you wrote the wrong query) or technical (the implementation is broken). Have a look at the following code:

```
var db = new ReadModelFacade();
var model = from i in db.IncomingInvoices
                        .ForBusinessUnit(buId)
                        .Expired()
            orderby i.PaymentDueDate
            select new SummaryViewModel.Invoice
                       {
                           Id = i.ID,
                           SupplierName = i.Party.Name,
                           PaymentDueDate = i.PaymentDueDate.Value,
                           TotalAmount = i.TotalPrice,
                           Notes = i.Notes
                       };
```

The code filters all invoices to retrieve those charged to a given business unit that haven't been paid yet. Methods like *ForBusinessUnit* and *Expired* are (optional) extension methods on the *IQueryable* type. All they do is add a WHERE clause to the final query:

```
public static IQueryable<Invoice> ForBusinessUnit(this IQueryable<Invoice> query, int buId)
{
    var invoices = from i in query
                   where i.BusinessUnit.OrganizationID == buId
                   select i;
    return invoices;
}
```

Last but not certainly least, LET fetches all data in a single step. The resulting query might be complex, but it is not necessarily too slow for the application. Here we can't help quoting the timeless wisdom of Donald Knuth: "Premature optimization is the root of all evil." As Andrea repeats in every class he teaches, three things are really important in the assessment of enterprise architecture: measure, measure, and measure. We're not here to say that LET will always outperform any other solution, but before looking for alternative solutions and better SQL code, first make sure you have concrete evidence that LET doesn't work for you.

Downsides of LET

Overall LET is a solution you should always consider, but like anything else it is not a silver bullet. Let's see which factors might make it less than appealing.

The first point to consider is that LET works beautifully on top of SQL Server and Entity Framework, but there's no guarantee it can do the same when other databases and, more importantly, other LINQ providers are used.

LET sits in between the application layer and persistence in much the same way repositories do. So is LET a general abstraction mechanism? The *IQueryable* interface is, in effect, an abstraction layer. However, it strictly depends on the underlying LINQ provider, how it maps expression trees to SQL commands, and how it performs. We can attest that things always worked well on top of Entity Framework and SQL Server. Likewise, we experienced trouble using LET on top of the LINQ provider you find in NHibernate. Overall, the argument that LET is a leaky abstraction over persistence is acceptable in theory.

In practice, though, not all applications are really concerned about switching the data-access engine. Most applications just choose one engine and stick to that. If the engine is SQL Server and you use Entity Framework, the LET abstraction is not leaky. But we agree that if you're building a framework that can be installed on top of your database of choice, repositories and DTOs are probably a better abstraction to use.

Finally, LET doesn't work over tiers. Is this a problem? Tiers are expensive, and we suggest you always find a way to avoid them. Yet sometimes tiers provide more scalability. However, as far as scalability is concerned, let us reiterate a point we made in a past chapter: if scalability is your major concern, you should also consider scaling out by keeping the entire stack on a single tier and running more instances of it on a cloud host such as Microsoft Azure.

Note When you use LET, testing can happen only on top of the LINQ-to-Objects provider built into the .NET Framework or any other LINQ provider that can be used to simulate the database. In any case, you're not testing LET through the real provider. For the nature of LET, however, this is the barrier that exists between unit and integration tests.

The command stack

In a CQRS scenario, the command stack is concerned only about the performance of tasks that modify the state of the application. As usual, the application layer receives requests from the presentation and orchestrates their execution. So what's going to be different in a CQRS scenario?

As shown in Figure 10-1, CQRS is about having distinct domain layers where the business logic—and objects required to have it implemented—is simpler to write because of the separation of concerns. This is already a benefit, but CQRS doesn't stop here. Additionally, it lays the groundwork for some more relevant design changes.

In the rest of the chapter, we drill down into concepts that slowly emerged as people increasingly viewed query and command stacks separately. These concepts are still evolving and lead toward the *event-sourcing* supporting architecture we'll discuss thoroughly in the next couple of chapters.

Getting back to presentation

A *command* is an action performed against the back end, such as registering a new user, processing the content of a shopping cart, or updating the profile of a customer. From a CQRS perspective, a task is monodirectional and generates a work flow that proceeds from the presentation down to the domain layer and likely ends up modifying some storage.

Tasks are triggered in two ways. One is when the user explicitly starts the task by acting on some UI elements. The other is when some autonomous services interact asynchronously with the system. As an example, you can think of how a shipping company interacts with its partners. The company might have an HTTP service that partners invoke to place requests.

The command placed updates the state of the system, but the caller might still need to receive some feedback.

Tasks triggered interactively

Imagine a web application like the I-Buy-Stuff online store we presented in Chapter 9. When the user clicks to buy the content of the shopping cart, she triggers a business process that creates an order, places a request for delivery, and processes payment—in a nutshell, it modifies the state of multiple systems, some interactively and some programmatically.

Yet the user who originally triggered the task is there expecting some form of feedback. That's no big deal when commands and queries are in the same context—the task modifies the state and reads

it back. But what about when commands and queries are in separated contexts? In this case, you have two tasks: one triggered interactively and one triggered programmatically. Here's some code from an ASP.NET MVC application:

```
[HttpPost]
[ActionName("AddTo")]
public ActionResult AddToShoppingCart(int productId, int quantity=1)
{
    // Perform the requested task using posted input data
    var cart = RetrieveCurrentShoppingCart();
    cart = _service.AddProductToShoppingCart(cart, productId, quantity);
    SaveCurrentShoppingCart(cart);

    // Query task triggered programmatically
    return RedirectToAction("AddTo");
}

[HttpGet]
[ActionName("AddTo")]
public ActionResult DisplayShoppingCart()
{
    var cart = RetrieveCurrentShoppingCart();
    return View("shoppingcart", cart);
}
```

The method *AddToShoppingCart* is a command triggered interactively, as evidenced by the *HttpPost* attribute. It reads the current state of the shopping cart, adds the new item, and saves it back. The command Add-to-Shopping-Cart ends here, but there's a user who still needs some feedback.

In this specific case—an ASP.NET application—you need a second command triggered programmatically that refreshes the user interface by placing a query or, like in this case, performing an action within the realm of the application layer. This is the effect of *RedirectToAction*, which places another HTTP request—a GET this time—that invokes the *DisplayShoppingCart* method.

What if you have a client-side web application—for example, a Single-Page application? In this case, you use some JavaScript to trigger the call to a Web API or SignalR endpoint. The task completes, but this time there's no strict need for the web back end to execute a second task programmatically to get back to the presentation. The nature of the client makes it possible to display feedback in the form of an acknowledgment message:

```
$("#buttonBuy").click(function() {
   // Retrieve input data to pass
   ...
   $.post(url, { p1: ..., p2: ... })
    .done(function(response) {
        // Use the response from the task endpoint to refresh the UI
        ...
    });
});
```

A similar mechanism applies when the client is a desktop or mobile application, whether it's Microsoft Windows, Windows Store, Windows Phone, Android, iOS, or something else.

Important In the case of an ASP.NET front end, the use of a redirect call to refresh the user interface is doubly beneficial because it defeats the notorious F5/Refresh effect. Browsers usually keep track of the last request and blindly repeat it when the user presses F5 or refreshes the current page. Especially for tasks that update the state of the system, reiterating a post request might repeat the task and produce unwanted effects—for example, the same item can be added twice to the shopping cart. A page refreshed after the task through a redirect leaves a GET operation in the browser memory. Even if it is repeated, no bad surprises can show up.

Tasks triggered programmatically

A system that exposes a public HTTP API is subject to receive calls from the outside. The call merely consists of the invocation of a method through an HTTP request, and the response is just an HTTP response. Here, the fundamentals of HTTP rule over all. Here's a sample Web API template for tasks to be invoked programmatically:

```
public class ExternalRequestController : ApiController
{
    public HttpResponseMessage PostDeliveryRequest(DeliveryRequest delivery)
    {
       // Do something here to process the delivery request coming from a partner company
       ...

       // Build a response for the caller:
       // Return HTTP 201 to indicate the successful creation of a new item
       var response = Request.CreateResponse<String>(HttpStatusCode.Created, "OK");

       // Add the location of new item for their reference
       var trackingId = ...;
       var path = "/delivery/processed/" + delivery.PartnerCode + "/" + trackingId;
       response.Headers.Location = new Uri(Request.RequestUri, path);
       return response;
   }
}
```

In this example, you have a Web API controller that receives delivery requests from a partner company. The request is processed and generates a tracking ID that must be communicated back to indicate the success of the operation.

There are various ways you can do this, and it mostly depends on your personal perspective regarding Web APIs. If you're a REST person, you would probably go with the code shown earlier. If you're more inclined toward remote procedure calls (RPCs), you can just return the tracking ID as a plain string in a generic HTTP 200 response.

Formalizing commands and events

All software systems receive input from some front-end data source. A data source can be any number of things, like a sensor connected to a hardware device that pumps real-time data, a feed asynchronously provided by a remote service, or—the most common scenario—a presentation layer equipped with a comfortable user interface. The input data travels from the front end to the application layer, where the processing phase of the input data is orchestrated.

Abstractly speaking, any front-end request for input processing is seen as a *message* sent to the application layer—the recipient. A message is a data transfer object that contains the plain data required for any further processing. In such an architecture, it is assumed that messages are fully understood by the recipient. Such a definition of a message leaves room for a number of concrete implementations. In most cases, you might want to start with a *Message* base class that acts as a data container:

```
public class Message
{
    // Optionally, a few common properties here.
    // The class, however, can even be a plain marker with no properties.
    ...
}
```

The front end can deliver the message to the application layer in a number of ways. Commonly, the delivery is a plain method invocation—for example, an application service invoked from within an ASP.NET MVC controller method. In more sophisticated scenarios, such as where scalability is the top priority, you might want to have a service bus in your infrastructure that also supports brokered messaging. In this way, you ensure delivery of the message to the intended recipient under any conditions, including when the recipient is not online.

Events vs. commands

There are two types of messages: commands and events. In both cases, messages consist of a packet of data. Some subtle differences exist, however, between events and commands.

A *command* is an imperative message that sounds like an explicit request made to the system to have some tasks performed. Here are some other characteristics of a command:

- A command is directed at one handler.
- A command can be rejected by the system.
- A command can fail while being executed by some handler.
- The net effect of the command can be different depending on the current state of the system.
- A command doesn't generally trespass the boundaries of a given bounded context.
- The suggested naming convention for commands says that they should be imperative and specify what needs to be done.

An *event* is a message that serves as a notification for something that has already happened. It has the following characteristics:

- An event can't be rejected or canceled by the system.
- An event can have any number of handlers interested in processing it.
- The processing of an event can, in turn, generate other events for other handlers to process.
- An event can have subscribers located outside the bounded context from which it originated.

Note The key difference between CQRS and the Event Sourcing architecture we'll cover in Chapter 12, "Introducing Event Sourcing," is this: in an Event Sourcing scenario, messages can be persisted to form a detailed and exhaustive audit log. This gives you a great chance at any later time to look back at what has happened within the system. Once you have the record of all that happened, you can set up a *what-if* elaboration, replay events to figure out the current state, and extrapolate models of any kind.

Writing an event class

In terms of source code, commands and events are both classes derived from *Message*. Dealing with commands and events through different classes makes the design of the system more logical and simpler overall. Here's a sample command class:

```
public class CheckoutCommand : Message
{
    public string CartId { get; private set; }
    public string CustomerId { get; private set; }

    public CheckoutCommand(string cartId, string customerId)
    {
        CartId = cartId;
        CustomerId = customerId;
    }
}
```

Conversely, here's the layout of an event class.

```
public class DomainEvent : Message
{
   // Common properties
   ...
}

public class OrderCreatedEvent : DomainEvent
{
    public string OrderId { get; private set; }
    public string TrackingId { get; private set; }
    public string TransactionId { get; private set; }
```

```
    public OrderCreatedEvent(string orderId, string trackingId, string transactionId)
    {
        OrderId = orderId;
        TrackingId = trackingId;
        TransactionId = transactionId;
    }
}
```

As you can see, the structure of event and command classes is nearly the same except for the naming convention. A fundamental guideline for designing domain events is that they should be as specific as possible and clearly reveal intent.

As an example, consider the form through which a customer updates the default credit card he uses in transactions. Should you fire a rather generic *CustomerUpdated* event? Or is a more specific *CreditCardUpdated* event preferable? Both options lead to a working solution, but which option works for you should be evident and stand on its own. It depends on the ubiquitous language and the level of granularity you have in place. We believe that a finer granularity here is a significant asset.

We generally recommend that the intent of the event be made clear in the name and all ambiguity be removed. If some ambiguity around a single event surfaces, you'll probably find that it's safer to have two distinct events.

Which properties should you have in the base class *DomainEvent*?

We'd say that at a minimum you want to have a timestamp property that tracks the exact time at which the event was fired. Moreover, you might want to have a property containing the name (or ID) of the user who caused the firing of the event. Another piece of data you might want to have is a version number that handlers can use to determine whether they can or cannot handle the event. The point here is that the definition of an event might change over time. A version number can help in this regard.

The implementation of the version number is completely up to the team. It can be a Version object as well as a string or a number. It can be bound to the application build number, or it can even be referred to the version of the event class:

```
public class DomainEvent : Message
{
    public DateTime TimeStamp { get; private set; }
    public DomainEvent()
    {
        TimeStamp = DateTime.Now;
    }
}
```

An event class should be considered immutable for the simple reason that it represents something that has already happened. *Immutable* here means that there should be no way to alter the value of properties. The combination of private setters, no write methods, and a plain constructor will do the trick.

Handling commands and events

Commands are managed by a processor that usually is referred to as a *command bus*. Events are managed by an *event bus* component. It is not unusual, however, that commands and events are handled by the same bus. Figure 10-3 presents the overall event-based architecture of a CQRS solution. This architecture is more standard and is an alternative to the one we mentioned earlier that was based on the TS pattern.

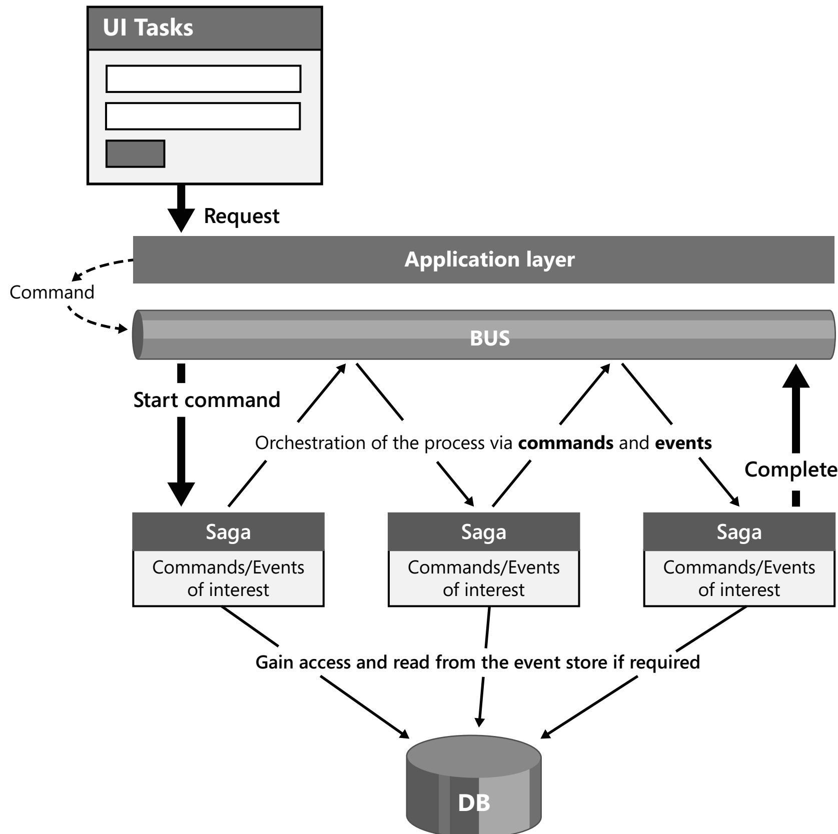


FIGURE 10-3 The command stack of an event-based CQRS architecture.

Any interaction that takes place in the user interface generates some requests to the system. In an ASP.NET MVC scenario, these requests take the form of controller actions and methods in the application layer. In the application layer, a command is created and pushed to some machinery for actual processing.

The bus component

The command bus holds a list of known business processes that can be triggered by commands. Active instances of such processes can be further advanced by commands. Processing a command can sometimes generate an event within the domain; the generated event is published to the same

command bus or to a parallel event bus, if any. Processes that handle commands and related events are usually referred to as *sagas*.

The command bus is a single class that receives messages (requests of executing commands and notifications of events) and finds a way to process them. The bus doesn't actually do the work itself; instead, it selects a registered handler that can take care of the command or event. Here's a possible template for a bus class that handles commands and events. We use the interface *IHandles* as a placeholder for actions. The interface has a single void method:

```
public interface IHandles
{
   void Handle(T message);
}
```

The bus uses the interface to handle both commands and events:

```
public class Bus
{
    private static readonly Dictionary<Type, Type> SagaStarters =
             new Dictionary<Type, Type>();
    private static readonly Dictionary<string, object> SagaInstances =
             new Dictionary<string, object>();

    public static void RegisterSaga<TStartMessage, TSaga>()
    {
        SagaStarters.Add(typeof(TStartMessage), typeof(TSaga));
    }

    public static void Send<T>(T message) where T : Message
    {
        // Publish the event
        if (message is IDomainEvent)
        {
            // Invoke all registered sagas and give each
            // a chance to handle the event.
            foreach(var saga in SagaInstances)
            {
               var handler = (IHandles<T>) saga;
               if (handler != null)
                   handler.Handle(message);
            }
        }

        // Check if the message can start one of the registered sagas
        if (SagaStarters.ContainsKey(typeof (T)))
        {
            // Start the saga creating a new instance of the type
            var typeOfSaga = SagaStarters[typeof (T)];
            var instance = (IHandles<T>) Activator.CreateInstance(typeOfSaga);
            instance.Handle(message);

            // At this point the saga has been given an ID;
            // let's persist the instance to a (memory) dictionary for later use.
            var saga = (SagaBase) instance;
```

```
            SagaInstances.Add(saga.Data.Id, instance);
            return;
        }

        // The message doesn't start any saga.
        // Check if the message can be delivered to an existing saga instead
        if (SagaInstances.ContainsKey(message.Id))
        {
            var saga = (IHandles<T>) SagaInstances[message.Id];
            saga.Handle(message);

            // Saves saga back or remove if completed
            if (saga.IsComplete())
                SagaInstances.Remove(message.Id);
            else
                SagaInstances[message.Id] = saga;
        }
    }
}
```

The bus has two internal dictionaries: one to map start messages and saga types, and one to track live instances of sagas. In the latter dictionary, you can typically have multiple instances of the same saga type that are bound to different IDs.

The saga component

In general, a saga component looks like a collection of logically related methods and event handlers. Each *saga* is a component that declares the following information:

- A command or event that starts the process associated with the saga
- Commands the saga can handle and events the saga is interested in

Whenever the bus receives a command (or an event) that can start a saga, it creates a new saga object. The constructor of the saga generates a unique ID, which is necessary to handle concurrent instances of the same saga. The ID can be a GUID as well as a hash value from the starter command or anything else, like the session ID. Once the saga is created, it executes the command or runs the code that handles the notified event. Executing the command mostly means writing data or executing calculations.

At some point in the lifetime of the saga instance, it might be necessary to send another command to trigger another process or, more likely, fire an event that advances another process. The saga does that by pushing commands and events back to the bus. It might also happen that a saga stops at some point and waits for events to be notified. The concatenation of commands and events keeps the saga live until a completion point is reached. In this regard, you can also think of a saga as a workflow with starting and ending points.

Events raised by sagas pass through the bus, and the bus passes them as messages to whomever subscribed to that event. Raising an event from within the saga requires code like that shown here:

```
// Raise the PaymentCompleted event
var theEvent = new PaymentCompletedEvent( /* add transaction ID */ );
theEvent.SagaId = message.Cart.Id;
Bus.Send(theEvent);
```

From the application layer, you invoke the bus as shown in the following example. This example simulates a scenario in which an ASP.NET MVC application is called back from the payment page on a bank service gateway:

```
public ActionResult CompleteProcessOrder(String transactionId)
{
    // Retrieve shopping cart from session state
    var cart = RetrieveCurrentShoppingCart();

    // Prepare and queue a Process-Order command
    var command = new ProcessOrderCommand(transactionId, cart.CartId);
    Bus.Send(command);

    // Refresh view: in doing so, results of previous command might be captured, if ready.
    return RedirectToAction("Done");
}
```

As you might have noticed, the command bus doesn't return a response. This is not coincidental. The refresh of the user interface—wherever necessary—is left to a subsequent read command that queries data from any place—storage, cache, or whatever—where output is expected to be.

What we mostly want from a command bus is to decouple the application layer from the domain services. Doing so opens up new opportunities, such as handling domain service calls asynchronously. Other scenarios that the command bus can simplify are adding cross-cutting filters along the way, such as transactions, logging, and general injection points for optional logic. In any of these cases, all you need to do is change the command-bus class—no changes are required to the application layer and domain services.

Important As battlefield experience grows around CQRS, some practices consolidate and tend to become best practices. Partly contrary to what we just stated about async domain services, it is a common view today to think that both the command handler and the application need to know how the transactional operation went. Results must be known, and if the command needs to run asynchronously, it should be designed as an event rather than as a command.

The combined effect of commands and events

When you write systems based on very dynamic business rules, you might want to seriously consider leaving the door open to making extensions and changes to certain workflows without patching the system.

Think, for example, of an e-commerce website that launches a marketing campaign so that any user performing certain actions through the site will accrue points on some sort of a customer-loyalty program. Because of the huge success of the campaign, the company decides to extend the incentives to more actions on the site and even double the points if a given action (say, buying suggested products) is repeated over time. How would you effectively handle that?

You can certainly fix and extend the application layer, which turns out to be the nerve center that governs the workflow. Anyway, when the business logic is expressed through the composition of small, independent and even autonomous commands and events, everything is much easier to handle. This is a classic scenario for collaborative systems. Let's say you use the flowchart in Figure 10-4 to implement the process of an order being submitted.

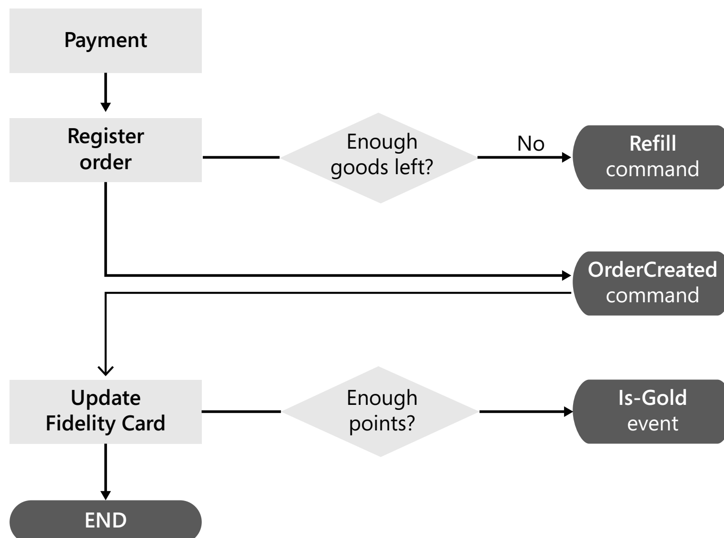


FIGURE 10-4 A sample workflow that handles the purchase of a few products.

Having the entire business logic of the workflow in a single method might make maintenance and testing problematic. However, even in the simple case of having split blocks in the workflow in simple commands, it might be difficult to do things properly. Let's consider the update of the customer's status in the fidelity program after an order is processed.

Instead of having the Register-Order command invoke the Update-Fidelity-Card command, isn't it more flexible if the Register-Order command just fires an event saying that a new order has been created? With this approach, a handler for that event can kick off and check whether the customer is eligible for Gold status and fire another event for other handlers to jump in and update databases.

This would isolate each action, keep each form of validation small and in reusable pieces, and keep the amount of logic in each command to the bare minimum.

Note When it comes to events, we highly recommend that the naming convention reflect the occurrence of something. For example, the generic Not-In-Store and Is-Gold events should be implemented through classes with more evocative names, such as *ItemWasNotInStoreWhenPayingEvent* and *CustomerReachedGoldStatusEvent*.

Sagas and transactions

Looking back at Chapter 8 where we introduced the Domain Model architecture, we can say that a saga is the evolution of the application service concept.

In an application service you orchestrate the tasks that serve a user request. The application service takes care of setting up transactions--possibly distributed transactions--and like a state-machine it coordinates the next step until the end of a workflow is reached. Abstractly speaking, a saga does the same except that it removes the need for a single component to know about the entire workflow and uses events to break an otherwise monolithic workflow into the combined effect of smaller command handlers raising events to coordinate with others.

This makes the entire solution a lot more flexible and reduces the surface of code subject to changes when business changes.

In addition, a saga reduces the need of distributed transactions when long-running processes are involved that span across multiple bounded contexts. The structure of a saga--a collection of rather independent command and event handlers--lends to easily define compensating behavior for each command executed. Thus if at some point a saga command fails, it can execute the compensating behavior for what it knows has happened and raise an event at the saga level to undo or compensate the work completed.

Note In the case of failures, saga methods might receive the same event twice or more. It might not be easy to figure out the event is a repeat. If it can be figured out, methods should avoid repeating action (idempotency)

Downsides of having a command bus

There are contrasting feelings about the command bus. A common criticism is that it just adds an extra layer and contributes to making the code less easy to read. You need to go through some more classes to figure out what happens.

This aspect can be mitigated by using proper naming conventions so that the name of the handler matches the name of the command class being queued to the bus. In this way, you know exactly where to look, and looking into the implementation of methods takes only a few more clicks than with application services, as discussed in past chapters.

With classic application services, you see the call to the method right from controllers and can use a tool like ReSharper to jump directly to the source code. With a command bus, you see the command and know by convention the name of the class in which you need to look.

Readymade storage

Most real-world systems write data and read it back later. Before CQRS appeared on the horizon, reads and writes took place within the same stack and often within the same transactional context. CQRS promotes the neat separation of queries and commands and pushes the use of different stacks. As you saw in this chapter, though, the internal architecture of the two stacks can be significantly different. You typically have a domain layer in the command stack and a far simpler data-access layer in the query stack.

What about databases, then? Should you use different databases too: one to save the state of the system and one to query for data?

Optimizing storage for queries

Many real-world systems use a single database for reading and writing purposes—and this happens regardless of CQRS architectures. You can have distinct command and query models and still share the same database. It's all about the scenarios you deal with.

Our thought is that using CQRS—at least in its lightweight form of using different models—is mostly beneficial for every system because it splits complexity, enables even different teams to work in parallel, and can suggest using something even simpler than a Domain Model for each stack. So CQRS is also about simplification.

However, for most practical purposes, using a single relational database is still the best option. When CQRS is used for scalability in a highly collaborative system, however, you might want to think about a couple of other aspects. (CQRS with a single database is sometimes referred to as *hybrid-CQRS*.)

In the query stack, what you query for has nearly the same schema as the view model. Most likely, these view models come from tables that are largely denormalized and that have very few links to other tables. In other words, the tables you really need are specialized for the queries you need to run.

A question that naturally arises is this: do you actually need a relational database for queries? We say you don't strictly need a relational database, but it's probably the best option you have, even though NoSQL solutions might be handier at times. More importantly—we'd say—you don't strictly need the same database where the state of the system is persisted.

Creating a database cache

If you're using one database to store the state of the system and another database to store data in a format that is quick and effective to query for the purposes of presentation, who's in charge of keeping the two databases in sync?

This seems to be the final step of nearly any command whose action affects the query database. To avoid giving too many responsibilities to each command, the most obvious approach is that any command whose action affects the query database fires an event at the end. The event handler will then take care of updating the query database. The query database doesn't fully represent the business behind the application; it only contains data—possibly denormalized data—in a format that matches the expectations of the user interface. For example, if you're going to display pending orders, you might want to have a query database just for the information you intend to display for orders. In doing so, you don't store the customer ID or product ID, just the real name of the customer, a full description of the product, and a calculated total of the order that includes taxes and shipping costs.

Stale data and eventual consistency

When each stack in CQRS owns its database, the command database captures the state of the system and applies the "official" data model of the business domain. The query database operates as a cache for data readymade for the presentation's needs. Keeping the two databases in sync incurs a cost. Sometimes this cost can be postponed; sometimes not.

If you update the query database at the end of a command, just before the end of the saga or at any point of the workflow when it makes sense, you automatically keep the command and query databases in sync. Your presentation layer will constantly consume fresh data.

Another approach that is sometimes used in the concrete implementations consists of delaying the synchronization between the command and query databases. The command database is regularly updated during the execution of the command so that the state of the application is consistent. Those changes, though, are not replicated immediately to the query side. This typically happens for performance reasons and to try to keep scalability at the maximum.

When the query and command databases are not in sync, the presentation layer might show stale data and the consistency of the entire system is partial. This is called *eventual consistency*—at some point, the databases are consistent, but consistency is not guaranteed all the time.

Is working with stale data a problem? It depends.

First and foremost, there should be a reason for having stale data; often, the reason is to speed up the write action to gain greater scalability. If scalability is not a concern, there's probably no reason for stale data and eventual consistency. Beyond that, we believe that very few applications can't afford displaying stale data for a short amount of time (with "short amount of time" being defined by the context). In many write-intensive systems, writes are sometimes performed in the back end and only simulated on the presentation to give the illusion of full synchronicity. The canonical example is when you post on a social network and the following two things happen:

- The back end of the system is updated through a command.
- At the end of the command, the DOM of the page is updated via JavaScript with the changes.

In other words, what appears as the real status of the system is simply the effect of a client-side command. At some point, however, in a few seconds (or minutes or days) something happens that

restores full consistency so that when the page is displayed from scratch reading from the server, it presents aligned data.

Eventual consistency is commonly achieved through scheduled jobs that run periodically or via queues that operate asynchronously. In this case, the saga ends by placing an event to a separate bus, possibly on a persistent queue. The queue will then serve events one at a time, causing the handler to fire and update the query database.

Summary

A decade ago, DDD started an irreversible, yet slow, process that is progressively changing the way many approach software architecture and development. In particular, DDD had the positive effect of making it clear that a deep understanding of the domain is key. And above everything else, DDD provided the tools for that: a ubiquitous language and bounded context.

Initially, DDD pushed the layered architecture with an object-oriented model as the recommended architecture for a bounded context. Years of experience suggested that a single domain model to handle all aspects of the business logic—specifically, commands and queries—was probably taking the complexity up to a vertical asymptote. From that point, there was an effort in the industry to find a different approach—one that could retain the good parts of DDD while making the implementation simpler and more effective.

CQRS was one of the answers.

CQRS propounds the separation between domain layers and the use of distinct models for reading and writing—a brilliantly simple idea that, however, had a far broader scope than first imagined. CQRS is perfect for highly collaborative systems, but it also can serve well in simpler scenarios if you apply it in a lighter form. That's a fundamental point and—as we see it—also the source of a lot of confusion.

CQRS is primarily about separating the stack of commands from the stack of queries. Realizing this architectural separation is not necessarily complex or expensive. However, if you take CQRS one or two steps further, it can really deliver a lot of added value in terms of the ability to manage much more business complexity while making your system much easier to scale out. To achieve this, you need to restructure and largely rethink the command and query sides and introduce LET, command buses, handlers, domain and integration events, and (likely) distinct storage for reads and writes. This flavor of architecture is ideal for collaborative systems, but it can be expensive to use in some systems. In the end, full-blown CQRS is a good solution, but not for every problem.

Finishing with a smile

To generate a smile or two at the end of this chapter, we selected three popular but largely anonymous quotes in the spirit of Murphy's laws:

- To understand what recursion is, you must first understand recursion.
- Before software can be reusable, it first has to be usable.
- Ninety percent of everything is CRUD.

In addition, we saved two funny but deep pearls of wisdom for last, from two people who contributed a lot to the software industry and computer science as a whole:

- If debugging is the process of removing software bugs, then programming must be the process of putting them in. (E. W. Diijkstra)
- C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg. (Bjarne Stroustrup)

This page intentionally left blank

Index

A

Abelson, Hal, 349
abstraction, 273
acceptance tests, 31, 94–96
accessibility, 10
actions
  in Behavior-Driven Design, 95
  in Domain-Driven Design, 121
  execution through components, 177
  implementing as classes, 174
  objects representing, 171
Active Record pattern, 168
adapters, 186
ADO.NET, 192
aggregates, 199–205, 266
  aggregate root objects, 200, 204–205, 344–346
  boundaries of, 202–203
  cross-aggregate behavior in domain services, 206
  data snapshots, 323, 340–348
  event handling, 347–348
  identifying, 235–243
  locking vs. keeping open to change, 260–261
  modeling, 199–200
  persistence, 199, 347, 357, 363–364
  rebuilding state, 328, 339
  repository for, 346–347. *See also* repositories
  snapshots vs. replaying events, 348
Agile development, 15–17
  Extreme Programming, 17
  Scrum methodology, 17
  training and practice, 41
  unforeseen events, 37
Agile Project Management with Scrum (Schwaber), 17
agility in architecture, 4, 11–12
Ajax, 152–153
Alexander, Christopher, 75
American National Standards Institute/Institute of Electrical and Electronics Engineers (ANSI/IEEE) standard 1471, 5
analysts, 18, 22
Anemic Domain Model (ADM), 63, 134, 174–176, 226–227
anticorruption layers (ACLs), 126–127
anti-patterns, 175–176
Apple's organizational culture, 35
application layer, 132–134, 177–180, 257
  connecting to data-access layer, 179–180
  connecting to presentation layer, 178–179
  in CQRS, 257
  event notification from, 336
  extending, 284
  front ends, 178
  messages delivered to, 277
  security concerns, 213
application logic, 198
application persistence, 195–196
application services, 134, 149–151, 248–251, 285
architects, 17–24
  as analysts, 21–22
  code writing, 23–24
  and developers, boundary between, 12
  flowcharts, use of, 305
  misconceptions about, 21–24
  openness and transparency, 32
  as project managers, 22
  responsibilities, 18–20
  roles, 4, 20–21
architectural patterns, 19. *See also specific pattern names*
architecture, 5
  agility in, 4, 11–12
  of bounded contexts, 127–129
  classic three-segment, 129–130, 167
  early decisions, 12–14
  emerging, 16–17
  event-based, 315–318
  implementation, 7, 12
  layered, 130–131, 167. *See also* layered architecture
  process of, 14–17
  requirements, 7. *See also* requirements
  single-tier, 129
  tasks, responsibility for, 14
  upfront, 15
  vision of, 7

Ariadne 5 disaster, 8–9, 78, 99
Arrange-Act-Assert (AAA) pattern, 93
arrays, as modeling tools, 245
aspects, 57
ASP.NET
  display modes, 222
  Identity facilities, 237–238
  websites, 147–152
ASP.NET MVC, 10, 177, 222
  controllers, 177–179
  display modes, 158–159
  presentation logic, 147–149
  task triggering, 275
  TempData facility, 252
  vs. Web Forms, 152–155
ASP.NET Web Forms
  vs. MVC, 152–155
  presentation logic, 149
  user interface, 151–152
async database operations, 361
asynchronous query methods, 361
async writes, 259
atomic methods, 364
audit logs, 278, 318
auto-completion, 39
AutoMapper, 186–187
automated testing, 86, 94–95, 101
Azure, 142

B

back ends
  commands against, 274
  entry point, 132–133
  partitioning, 263
bad code, 39–41, 44–48
Ballmer, Steve, 36
Balsamiq, 142
behavior
  black box of, 12
  domain vs. application logic, 198
  flowcharts of, 305
  as focus in modeling, 225–228
Behavior-Driven Design (BDD), 95–96
best practices, 14
Big Ball of Mud (BBM), 9, 27
  causes, 28–32
  detecting, 34–35
  requirements churn, 43
  symptoms of, 32–34
Big Bang pattern, 93
black box of behavior, 12
black-box testing, 94, 101
blocks of code, 45–48
Boolean conditions, 81–82
Boolean guards, 80
bounded contexts, 47, 121–129, 193
  adding, 224
  architecting, 116, 127–129, 255, 258. *See also* Command/Query Responsibility Segregation (CQRS)
  boundaries, marking, 122–123
  business domain, decomposing into, 125, 222–223
  communication between, 223
  context mapping, 116, 125–127
  contextual overlaps, 123–124
  defined, 122
  discovering, 115, 122–125
  integration events, 302
  in layered architecture, 255
  modules, 197
  number of, 218
  relationships between, 125–126
  supporting architecture for, 196
  ubiquitous language of, 122
Box, George E. P., 51
breakdown process, 11–12, 19
Brooks, Frederick P., 25, 49, 349
Brooks law, 38
Buschmann, Frank, 76
business domain
  bounded contexts, decomposing into, 222–223
  context of, 114, 117
  entities, state of, 321–323
  entity snapshots, 323
  jargon, translating to ubiquitous language, 119
  partitioning or splitting, 122–124
  subdomains of, 122
  understanding, 113
business layer, 167–188
  CQRS in, 260–262
  data flow across boundaries, 182–187
  patterns for organizing, 167–176
  task-based modeling, 176–182
  testing, 100–101
business logic
  in domain entities and domain services, 180, 205
  events, flow of, 319
  hosting, 134
  implementing, 167, 248–253. *See also* business layer
  location of, 227–228
  modeling, 58
  orchestrating, 133–134, 294
  patterns for organizing, 13, 167–176
  single commands, expressing as, 262
  in stored procedures, 129
  workflow of, 284
business people, language of, 10–11
business requirements, 28. *See also* requirements

business rules
  filters, expressing with, 269
  modeling with aggregates, 200

C

C#, common conventions for, 107
caching, 259
  performance costs, 215
  placement of code, 214
  in repositories, 366–367
cardinality, 201, 203, 258
Cartesian product, 258–259
case studies, 14
Cassandra, 374
check-in policies, 41, 108
Churchill, Winston, 34
churn, requirement, 43
clarity in code, 108–109
classes. *See also* modules
  anemic domains, 63
  base and derived, 62
  breakage, 33
  cohesion, 55–56
  coupling, 56–57
  domain models, 63, 173–174
  extension, 66
  inheritance of, 61
  moving, 78
  pertinent, 58–59
  renaming, 78
  sealed and unsealed, 13–14
  single responsibility, 56, 65
  wrapper, 62
client/server architecture, 128–129
client-side device detection, 159–160
cloud platforms, 259
code
  adapting features, 29
  check-in policies, 41
  costs, 39. *See also* costs
  deterioration, 53
  duplication, 74
  hacks, 34
  line length, 109
  maintainability, 53. *See also* maintainability
  quality, 97–98, 110
  requirements, mapping to, 29. *See also* requirements
  spaghetti, 54
  static code analysis, 34–35
  unnecessary, 73–74
  validating, 29
  vertical scrolling, 109
  workarounds, 34
  writing from scratch, 33. *See also* coding
code-analysis tools, 34–35, 45
code-assistant tools, 39–41, 67, 78, 109
codebase
  inspecting, 41
  legacy code, 44–45
  redesigning vs. rewriting, 44
Code Contracts rewriter (Visual Studio), 67, 83
code coverage in testing, 98–99
code fixes, calendar of, 32
code inspections, 39
code kata, 226
code reuse, 61–63, 178
CodeRush (DevExpress), 34, 40
code smell, 88
coding. *See also* development
  bad code, 39–41, 44–48
  code-assistant tools, 39–41, 67, 78, 10939–40
  code katas, 226
  code quality, 38–43. *See also* quality software
  common conventions, 104–109
  continuous change and, 43
  costs of, 39
  deadlines, 42
  and design, simultaneous, 55
  heroism, avoiding, 42
  inspecting code, 41
  practicing, 41–42
  quality. *See* quality software
  standards, 41
  team guidelines, 108
  for use-cases, 121
coding vectors, 72–75
cohesion, 55–56
collaborative systems, 260–262, 284
collections
  data access, 264–265
  exposing to queries, 267
  of persistent data, 355
  read-only, 264–265
columns, mapping properties to, 246–247
column stores, 320
command and query systems, 171
command architecture, 294–296, 302–309
command bus, 280–283
  command processing, 302
  naming conventions, 285–286
  saga management, 305–309
  for workflow implementation in event sourcing, 332–333
command classes, writing, 278
Command pattern, 171

Command/Query Responsibility Segregation (CQRS), 128, 256–288, 291–292
- benefits, 258–260
- bounded contexts, architecting, 258
- in business layer, 260–262
- for collaborative systems, 260–263
- with command architecture. *See* command bus
- command bus components, 285–286
- commands and queries, separating, 256–263
- command stack, 274–288, 302–309
- consistency, 263
- data access layer, 268
- databases in, 286–288, 293–296
- vs. Domain Model, 256–257
- vs. Event Sourcing architecture, 278
- eventual consistency, 287–288
- layers, 257
- packaging data for callers, 298–301
- persistence layer, 355
- plain and simple approach, 292–294
- preliminary analysis for, 258, 260
- pure CQRS approach, 271
- query stack, 264–274, 296–301
- Repository pattern and, 357
- saga components, 285. *See also* saga components
- scalability of systems, 259
- stale data, 287–288
- tasks, 274

commands
- characteristics of, 277
- defined, 256, 274
- granularity of, 309
- handling, 280–286
- processing, 302
- queues of, 259
- in saga components, 305–309
- separating from queries, 241, 256–263. *See also* Command/Query Responsibility Segregation (CQRS)

command stack
- command and event handling, 280–286
- command bus components, 280–283
- command processing, 261–262, 294
- domain model for, 258
- event-based CQRS architecture, 280
- implementing, 302–309
- messages, commands, and events, 277–279
- saga components, 281–283, 302–305
- save operations, 363
- storage, 286–288
- tasks, 274–276
- Transaction Script pattern, 262–263, 292

comments in code, 105–106, 108

communication in development, 29, 42. *See also* ubiquitous language

complexity
- bounded contexts, number of, 218
- Domain Model pattern for, 173
- indicators of, 173
- modeling and, 225
- perception of, 171
- reducing with CQRS, 258–259
- solutions, matching to, 168–169

composition vs. inheritance, 61–63

conceptual queries, 269

concerns, 57
- cross-cutting, 60, 212–215

conditional statements, 78–79

conformist relational pattern, 126

connection strings, 181, 208, 263

consistency, 108
- ACID, 373
- boundaries of, 199
- with CQRS, 263
- eventual. *See* eventual consistency
- transactional, 200

Constantine, Larry, 55

constructors, 230–231, 244

context. *See also* bounded contexts
- design decisions based on, 6, 14

context mapping, 116, 125–127
- of online store sample project, 224–225

continuous change, 43

control, 87

controller factories, 178–179, 222

controllers, 146, 248, 251
- application and presentation layers, connecting, 178–179
- as coordinators, 177–178
- presentation logic in, 147–148
- testability and, 148
- thin, 147

Cook, Rick, 324

costs, 30–31, 38
- of bad vs. good code, 39
- of code maintenance and evolution, 39
- direct and indirect, 49
- of event stores, 337
- of maintenance, 39, 83
- of polyglot persistence, 374
- of refactoring, 43
- of relational storage, 371, 375
- of single-page applications, 163

coupling, 56, 59

Cray, Seymour, 324

crises, recognizing, 32

cross-cutting concerns, 60, 212–215

CRUD (Create, Read, Update, Delete), 168–169
- vs. event sourcing, 326, 328
- interface, 212
- limitations of, 326

Cunningham, Ward, 34, 56
customer objectives, capturing, 28–30
customers entities, 235–238
customer/supplier relational pattern, 126126

D

Dahan, Udi, 260
data
  growth of, 376–377
  homogeneity, 376
  persistence of, 353–377. *See also* persistence
  reading from permanent stores, 355
  schema volatility, 376
  source of, 315, 353
Data Access, 365
data access layer
  connecting to application layer, 179–180
  in CQRS, 268
data annotations, 246
database caches, 286–287
databases
  accessing with layered expression trees, 270–272
  arrays and, 245
  in CQRS systems, 257, 262–263, 286–288, 293–296
  in Domain-Driven Design, 195–196
  event stores, 317–321
  eventual consistency, 287–288, 320
  inferring objects from, 175
  query-only access, 267
  read database context, 298–300
  reading optimized, 271
  stale data, 287–288
  synchronizing data, 295
data clumps, 199, 235
data flow
  across boundaries, 182–187
  in layered architecture, 182–183, 248
data models, 183, 353. *See also* modeling
data operations, capturing and tracking, 328. *See also* event sourcing (ES)
data snapshots, 323, 340–343
data source context classes, 362
data sources, 277
data stores, 376–377. *See also* storage technologies
data-transfer objects (DTOs), 140, 185–186
  building, 186–187
  vs. domain entities, 185–186
  layered expression trees and, 272–273
  name and quantity, 268
  in query stack, 293
  reducing need for, 301
  returning subsets of data, 359–360
data update conflicts, controlling, 328
DbContext class, 266–267, 296, 298, 361
deadlines, unfair and wrong, 42
decision coverage, 98
defensive programming, 62, 78–83
dependencies
  between classes and calling code, 59
  coupling and, 56–57
  graphs of, 34
  ignoring, 91
  isolating in testing, 90–91
  on LINQ providers, 362–363
  number and depth of, 33
  patterns for handling, 69–72
Dependency Injection (DI) pattern, 60, 71–72, 178–180
Dependency Inversion principle (DIP), 69–72, 86
derived classes, substitutability, 66–68
design
  and coding, simultaneous, 55
  idiomatic, 39
  immobility of, 33
  object-oriented vs. procedural, 169
  patterns, 75–77. *See also specific pattern names*
  requirements driven, 19
  simplification with CQRS, 258–259
  task-based, 138–139
  Test-Driven Development aware, 97–98
  up-front, 55, 135
  user experience first, 138–146
Design by Contract, 79
Design for Testability (DfT), 87
Design Patterns: Elements of Reusable Object-Oriented Software (Gamma, Helm, Johnson, and Vlissides), 58
design refactoring, 43. *See also* refactoring
design reviews, 41
desktop rich clients, 164–165
developers. *See also* coding
  and architects, boundary between, 12
  augmenting skills, 40–41
  heroism, 36, 41–42
  objectives, pursuing, 36–37
  perspective of, 226
  respect through readable code, 106
  ubiquitous language use, 118
development
  life cycle of, 6
  responsibilities of, 14
  restarting from scratch, 44
  stopping, 45
development team
  architects, 12, 17–24, 32, 305
  developers. *See* developers
  estimates of projects, 30–31
  firefighters, 37–38
  knowledge silos, 35
  leadership, 38

development team (*continued*)
managers, 36–38
negotiation of goals, 36–37
project managers, 15, 20, 22
roles, defining, 12
device description repositories (DDRs), 157
devices
client-side detection, 159–160
display modes, 158–159
feature detection, 155–156
server-side adaptive solutions, 158–159
server-side detection, 157–158
device support for websites, 155–160
Dijkstra, Edsger W., 57, 64, 289
distributed memory caches, 367–368
document databases for event stores, 320
Document/Object NoSQL store, 370
domain, defined, 122
domain architecture, 113
bounded contexts, 121–129
Domain-Driven Design, 114–118
layered architecture, 129–135
ubiquitous language, 118–121
Domain-Driven Design (DDD), 47, 114–118, 191
analytical part, 115, 120, 194, 255
application services, 134
bounded contexts, 47. *See also* bounded contexts
context mapping, 193
databases as infrastructure, 195–196
domain layer, 191–215. *See also* domain layer
Domain-Driven Design (DDD) (*continued*)
entities, 228. *See also* domain entities
faithfulness to observed processes, 176–177
models and domains, 192–194
relational patterns, 125–126
strategic model design, 116–118
strategic part, 115, 255
subdomains, 122
ubiquitous language, 11. *See also* ubiquitous language
validation, 213
viability, 194
Domain-Driven Design: Tackling Complexity in the Heart of Software (Evans), 114, 116, 193
domain entities
vs. data-transfer objects, 185–186
equality, 229–230
identities, 228–230
operator overloading, 234
private setters, 230
referencing from UI, 185
scaffolding, 228–231
sharing, 184–185
validation, 213
domain events, 209–212. *See also* events
in command execution, 302
as core of architecture, 315–318
design of, 279
handling, 211–212
implementing, 210–211, 253
raising, 210
when to use, 209–210
domain experts, 18, 118
domain layer, 134, 191–215
aggregates, 199–205
in CQRS, 257
cross-cutting concerns, 212–215
data flow to, 248
domain events, 209–212
domain services, 205–209
entities, 198–199
modules, 197
persistence of entities, 199
repositories, 131
security, 213–214
splitting in CQRS, 257–258
structure of, 196–197
testing, 100–101
validation, 213
value objects, 198
domain logic, 198, 205
Domain Model, 116, 128, 134, 172–174
business logic, location of, 227–228
defined, 122
domain entities, 184. *See also* domain entities
logging, 214
for online store sample project, 219–220, 225–247
persisting, 208
Repository pattern and, 357–358
and ubiquitous language, syncing conventions, 121
domain modeling, 58, 63, 65, 168, 225–247, 311. *See also* Domain-Driven Design (DDD)
aggregates, 199–205
behavior focus, 225–228
complexity and, 225
CQRS systems, 257–258
entity-based, 315
evolution of techniques, 176
of real world, 312–313
vs. scripting objects, 170
traditional approach, 191–192
domain models, 63, 172
behavior-based, 194, 225–226
benefits of, 175
commands and queries, separating, 256–263
constructors, 230–231
defined, 196
exceptions, 213
factories, 230–231

domain models (*continued*)
  front ends, 185
  invariant conditions, 200
  object-based, 193
  persistence, 183, 195, 243–247
  plain-old C# objects, 181
  read-only, 264–266, 297
  Special Case pattern, 242–243
domain services, 134, 180–181, 205–209
  cross-aggregate behavior, 206
  naming conventions, 180–181
  read operations, allowing, 266
  repositories, 207–208
downloader and uploader components, 366
downstream contexts, 125–126
DRY (Don't Repeat Yourself), 74

E

Edison, Thomas, 118
EDMX wrappers, 263
efficiency, 8
Einstein, Albert, 73
emerging architecture, 16–17
encapsulation, 78
end-to-end scenarios, 28
enterprise architects (EAs), 20
enterprise resource planning system, 329–342
entities, 198–203. *See also* domain entities
Entity Framework, 174–175, 192, 226, 245, 365
  asynchronous query methods, 361
  Code-First approach, 222, 245–247, 341
  for CQRS data-access layer, 268
  database access, 298
  Database-First approach, 245
  DbContext wrapper class, 246, 267
  EDMX wrappers, 263
  layered expression trees on, 273
  LINQ and, 362
  repository classes, implementing, 363–364
entity models, 134, 202
equality, 229–230, 232–233
estimates of software project, 30–31, 49–50
Evans, Eric, 114, 116, 118, 193, 199, 206
event bus components, 280
event classes, 278–279
event-driven architecture, 167
events, 312, 315–318. *See also* domain events
  in aggregate roots, 344–346
  characteristics of, 278
  as data sources, 315, 325–326
  event replay and what-if scenarios, 316
  extensibility and, 316
  granularity of, 316
  handling, 280–286
  last-known good state and, 313–315
  logging, 343
  materializing objects out of, 339
  naming convention, 285
  persistence of, 311, 318–321, 337–338, 342. *See also* event sourcing (ES)
  raising from commands, 280, 294
  raising from sagas, 282–283
  in real world, 312–313
  replaying, 316, 321–323, 328–342
  repository for, 346
  in saga components, 304–309
  sequence of, 313–315
  state, rebuilding from, 321–323
  storing, 328
  tracking system with, 315
event sourcing (ES), 274, 311–324
  with aggregate snapshots, 342–348
  architecture of, 318–323
  command bus for workflow implementation, 332–333
  vs. CQRS, 278
  data snapshots, creating, 340–342
  drawbacks, 317–318
  event notification, 331, 336
  events as data sources, 315
  events sequence, 313–315
  latency and, 328
  log of events, 326–327
  need for, 325–326
  persisting events, 337–338. *See also* persistence
  real-time systems and, 329
  with replay, 329–342, 348
  tracking and rebuilding events, 326–327
  undo functionality, 336–337, 343
  when to use, 327–329
event stores, 317–321
  cost of, 337
  front end, 340
  performance and, 321
  replaying events, 338–339
  table schema, modeling, 337–338
eventual consistency, 200, 260–261, 287–288, 320, 328
  in NoSQL data stores, 372–373
Ewing, Sam, 349
exceptions, testing parameters of, 89–90
expertise, adding to late projects, 50
expression trees, layered, 269–274
expressivity, 234–235
extensibility, 85, 101–103, 109–110, 185
extension points, 103
external quality characteristics, 8
extracting interfaces, 78
extracting methods, 78

Extreme Programming (Cunningham), 56
Extreme Programming (XP), 17, 56
  acceptance tests, 94–95
  YAGNI principle, 73–74
extreme scalability, 10

F

F5/Refresh effect, 276
factories, 230–231
fake objects, 91–93
Feathers, Michael, 44, 47
feature detection, 155–156
fields, encapsulating, 78
filters
  for business rules, 269
  in queries, 270–271, 362
firefighters, 37–38
flowcharts, 305
fluent Code-First API, 246–247
Foote, Brian, 28
Fowler, Martin, 12, 103, 116, 169, 175, 181, 355
fragility of software, 33
front ends, 164, 178, 185
  data sources, 277
  user interface, refreshing, 276
functionality, 8
  layering, 19
functional programming, 116
functional requirements, 7–9, 28. *See also* requirements
  churn in, 43
  processing, 11
functional unit testing, 93–94. *See also* unit tests
function coverage, 98
functions, defined, 8

G

Gagarin, Yuri, 27
Gamma, Erich, 58
Gang of Four (GoF), 58
gated check-ins, 41
Genome, 365
Given/When/Then statements, 96
global methods, 364
goals of project development, negotiation of, 37
GOTO-based code, 54
Graph NoSQL store, 370

H

hash codes, equality and, 233
healthcare.gov, 28, 31–32, 94
Helm, Richard, 58
Henney, Kevin, 105
Hexagonal Architecture (HA), 19
hidden dependencies, 33
homogeneous data, 376
HTML, 164–165
HTTP endpoints, 153–154
HTTP interfaces, 180
HTTP requests and responses, 276
Hunt, Andy, 74, 101
hybrid-CQRS, 286

I

ICanHandleMessageT interface, 306
identity, 228–230
idiomatic design, 39
If-Then-Throw pattern, 78–79
immobility of design, 33
immutable types, 198, 232, 279
implementation of architecture, 7, 12
indexes, updating, 372–373
information hiding, 57
information silos, 35
infrastructure layer, 134–135, 212–215
inheritance, 61–63, 66
in-memory lists, 271
input models, 131, 140, 144–145, 183
integration, 31–32
integration events, 302
integration tests, 31, 93–94
  Big Bang pattern, 93
  bottom-up, 94
  in refactoring, 47
interactions
  focusing on, 138–140
  views, creating from, 143–144
interfaces
  design based on, 102–103
  extracting, 78
  and implementation, separating, 59–60
  programming to, 69
  thin, 68–69
Interface Segregation principle (ISP), 68–69
internal quality characteristics, 8
International Obfuscated C Code Contest (IOCCC), 104–105
International Organization for Standards/International Electrotechnical Commission (ISO/IEC) standards
  9126, 7–8, 105
  25010, 8
  42010, 5, 18
interviews, deriving raw requirements from, 9
invariants, 82, 200

Inversion of Control (IoC) pattern, 71–72, 178–179
IQueryable interface, 187, 268, 270–271, 273
  extensions, 301
  objections to, 362–363
  returning types, 298–299, 360–362
Isaacson, Walter, 36
isolated code blocks, 46–47
isolation, 57
IStartWithMessageT interface, 306

J

JavaScript in presentation layer, 152
Jobs, Steve, 35–36
Johnson, Ralph, 58
JustCode (Telerik), 40

K

Kay, Alan, 349
key value NoSQL store, 370
KISS (Keep It Simple, Stupid), 73
knowledge silos, 35
Knuth, Donald, 102, 273

L

last-known good state, 313–314
latency, 328
late projects, 49–50
layered architecture, 19, 129–135, 255
  application layer, 132–134. *See also* application layer
  boundaries, 48
  connecting layers, 178–180
  data flow, 182–183
  defined, 48
  domain entities, 184–185. *See also* domain entities
  domain layer, 134, 196. *See also* domain layer
  infrastructure layer, 134–135, 212–215
  for online store sample project, 220–221
  origins, 129–131
  persistence layer, 353–377
  presentation layer, 131–132. *See also* presentation layer
  priority of layers, 137
  vs. tiers, 129
  tight coupling, 180
layered expression tree (LET) expressions, 299–301
layered expression trees (LETs), 269–274, 293
  vs. in-memory lists, 271
  IQueryable objects, 270–271
  testing, 274
lead developers, 21
leaders vs. bosses, 38
legacy code, 44–48
Lehman, Manny, 85
libraries, 13, 197
LINQ providers
  async version, 270
  in CQRS data access layer, 268
  dependencies on, 362–363
  query composition and execution, 362
LINQ-to-Entities, 267
LINQ-to-SQL, 192
Liskov's principle (LSP), 66–68
live-scoring sample project, 342–348
LLBLGen Pro, 365
logging, placement of code, 214
login systems, 236–238

M

maintainability, 8, 53, 55, 83, 105–106, 118
maintenance
  costs, 39, 83
  extensibility and, 101–102
  readability and, 106
Managed Extensibility Framework (MEF), 72
managers, 36–38
manpower, adding to late projects, 49–50
Martin, Robert C., 56, 64, 97
Maslow, Abraham, 378
McKean, Alan, 177
MEF 2, 72
Memcached, 367
memory caches, distributed, 367–368
Mercator projection, 193–194
Message base class, 277
messages, 277–279
  persistence of, 321
methodologies, 14–15
  Agile approach, 15–17
  Extreme Programming, 17
  Scrum, 17
  sprint zero, 17
  Waterfall model, 15
methods
  conditional statements, 78–79
  extracting, 78
  invariants, 82
  length, 109
  naming, 107, 240
  postconditions, 81–82
  preconditions, 80
  private vs. public, 107
metrics of technical debt, 34–35
Meyer, Bertrand, 79

micro-services, 19, 128
Microsoft Azure, 259
Microsoft Entity Framework. *See* Entity Framework
Microsoft .NET Framework. *See* .NET Framework
Microsoft Patterns and Practices Unity, 222
Microsoft Pex add-in, 101
Microsoft's organizational culture, 36
Microsoft SQL Server, 222
Microsoft Team Foundation Server (TFS), 41
Microsoft Unity, 178–179
Microsoft Visual Studio. *See* Visual Studio
mini enterprise resource planning system, 329–342
mobile applications, 165, 366
mobile devices, 155
mobile-first design, 156–157
mock objects, 91–93
mockups, 141–144
modeling, 146. *See also* domain modeling; domain models
  object modeling, 65, 74–75, 116
Model-View-Controller (MVC) pattern, 146, 183
Model-View-Presenter (MVP) pattern, 146
Model-View-ViewModel (MVVM) pattern, 146
Modernizr, 156
modular programming, 57
modules, 197
  closed for modification, 66
  defined, 35
  entities, 198
  open for extension, 66
  separation of, 260
  value objects, 198
monolithic architecture, 128
Moq, 92
MSBuild scripts, 41
MSTest, 89
multilayer architecture, 128. *See also* layered architecture
  data flow, 182–183
  deployment of, 130
multitier architecture, 128, 181–182
multivendor systems, 28, 31–32
Mythical Man Month, The (Brooks), 25, 49

N

namespaces, 197
native mobile applications, 165
NDepend, 34
.NET Framework
  classes, 198
  Code Contracts library, 79–83
  developers, typical perspective of, 226
  hash codes and equality, 233
  immutable types, 198, 232
  namespaces, 197
  operators, overloading, 234
  validation, 213
NEventStore project, 320, 338
new technologies, 14
NHibernate, 174, 245, 365
NMock2, 92
nonfunctional requirements, 7, 9–10, 43. *See also* requirements
nonrelational storage, 368–377
North, Dan, 95
NoSQL, 368–370
  databases for event stores, 320
  eventual consistency, 372–373
  flavors, 370
  planning decisions, 375–377
  use-cases for, 369–370
NUnit, 89

O

OASIS, 366
obfuscated code, 104–105
object composition, 61–63
Object Design: Roles, Responsibilities, and Collaborations (Wirfs-Brock and McKean), 177
object modeling, 65, 116
  Tell-Don't-Ask principle, 74–75
object orientation (OO), 57–58, 63–64
object-oriented design (OOD), 57–64
  cohesion, 56
  composition vs. inheritance, 61–63
  coupling, 56, 59
  defined, 58
  pertinent classes, 58–59
  programming to interface, 59–60
Object/Relational Mapper (O/RM) tools, 174, 196, 365–366
  in CQRS data access layer, 268
  persistence responsibilities, 243–244
  protected constructors, 231
objects
  actions, representing with, 171
  attributes, 198
  behavior associated with, 174
  cardinality, 201, 203
  domain logic in, 177
  fake, 91–93
  inferring from databases, 175
  local instantiation of, 180
  materializing out of events, 339. *See also* event sourcing (ES)
  mock, 91–93

online store sample project, 218–225
- aggregates, 235–243
- bounded contexts, 222–224
- business logic, implementing, 248–253
- context mapping, 224–225
- CQRS command-based architecture, 274–276
- design approach, 219–220
- domain events, 253
- domain modeling, 225–247
- entities, 219–220
- fidelity card functionality, 253
- order processing, 249–253
- structure of, 220–221
- technologies used, 222
- use-cases, 218–219

"On the Role of Scientific Thought" (Dijkstra), 57
Open/Closed principle (OCP), 66
Open Data Protocol (OData), 162–163, 366–367
open for extension, 66
open-source software, 109
Open Web Interface for .NET (OWIN), 155
operator overloading, 234
organizational culture, software project success and, 35–38
over-engineering, 102–103

P

page redirects, 252
partitioning architectural pattern, 19
partnership relational pattern, 126126
path coverage, 98
Pattern-Oriented Software Architecture (Buschmann et al.), 76
patterns, 75–77. *See also specific pattern names*
- anti-patterns, 175–176
- business logic, organizing with, 167–176
- cross-cutting concerns, decomposing with, 60
- dependencies, handling, 69–72
- use of, 75–77
- value in, 77

Patterns of Enterprise Application Architecture (Fowler), 12, 169
payment APIs, 252
peer programming, 41
performance
- caching and, 215
- event stores and, 321
- replay of events and, 322

persistence, 174
- of aggregates, 347
- in applications, 195–196
- concessions to, 196, 244–245
- default constructors, 244
- in domain models, 183, 195, 208, 243–247
- of entities, 199
- of events, 311, 313–315, 318–321, 337–338
- persistence layer. *See* persistence layer
- polyglot, 368, 371, 374
- repository classes and, 181
- of saga components, 308

persistence layer, 135, 246, 353–377
- nonrelational storage, 368–377
- polyglot persistence, 368, 371, 374
- repositories, implementing, 359–368
- Repository pattern, designing, 355–359
- responsibilities of, 354–355

pertinent classes, 58–59
PetaPoco, 268
pet-technology anti-pattern, 168
PhoneGap, 13
plain old CLR objects (POCOs), 173
plain old C# objects (POCOs), 181, 236
plugin-based architecture, 103
Plugin pattern, 103
polyglot persistence, 368, 371, 374
portability, 8, 11
postback handlers, 149
postconditions, 81–82
Post-Redirect-Get pattern, 252
practice in coding, 41–42
Pragmatic Programmer: From Journeyman to Master, The (Hunt and Thomas), 74, 101
preconditions, 80, 95
predefined snippets, 39
predefined templates, 39
presentation layer, 131–132, 137–166
- ASP.NET websites, 147–152
- connecting to application layer, 178–179
- defining, 137
- desktop rich clients, 164–165
- device support for websites, 155–160
- patterns for, 146
- presentation model, 184
- security, 214
- single-page applications, 160–163
- stale data, 287–288
- user experience first, 138–146
- Web Forms vs. ASP.NET MVC, 152–155

presentation logic, 147
- in ASP.NET MVC, 147–149
- in ASP.NET Web Forms, 149
- in controllers, 147–148
- in single-page applications, 160–161

presenter elements, 146
primitive types, 234
private setters, 230, 264–266

procedural designs
  Anemic Domain Model, 175
  Transaction Script pattern, 169–172, 262–263, 292
productivity, code quality and, 110
programming to interface, 59–60
progress tracking, 49
project management, 14
project managers, 18, 20, 22
properties, mapping to columns, 246–247
prototypes, 141, 143–146
public interface, 57
pure CQRS approach, 271. *See also* Command/Query Responsibility Segregation (CQRS)

Q

quality assurance (QA), 14
quality software
  characteristics of, 7–8
  extensibility, 85, 101–103
  readability, 85, 104–109
  SOLID principles, 86
  testability, 85–88
  testing, 88–101
queries
  composing, 270–271
  conceptual, 269
  DDD-friendly, 272
  defined, 256
  separating from commands, 241, 256–263
  storage, 286–288
query databases, 286–288
query methods, 359–362
QueryModelDatabase class, 296, 298–299
Query Object pattern, 244
query stack
  components, 262
  in CQRS, 257
  data-access layer, 263, 293
  database context, restricting, 266–267
  data model, 296–297
  data-transfer objects, 293
  domain model of, 264–266
  eventual consistency, 287–288
  implementing, 296–301
  layered expression trees, 269–274
  packaging data for callers, 298–301
  read façade, 266–268, 270, 296–298
  repositories for, 268, 359–360
  simplification of, 257
  stale data, 287–288

R

Rapid Application Development (RAD), 30
Ratchet2, 222
RavenDB, 321, 337–339, 372–373
readability, 85, 104–109
  constructors and, 230–231
  refactoring for, 77–78
read façade, 266–268, 270, 296–298
read-only domain models, 266
read-only wrappers, 265
read operations. *See also* queries
  external services for, 366
  model for, 241
  servers for, 355
real-time systems, event sourcing and, 329
redesigns, 44
redirect calls, 276
refactoring, 43, 77–78
  continuous, 48
  data clumps, 235
  effectiveness, 85
  operations of, 78
  techniques, 45
  in Test-Driven Development, 97
  testing for, 47–48
regression, 12, 31, 33–34, 47, 260
  maintainability and, 53, 55
  testing for, 85, 88, 93–94, 97–98
relational DBMS for event stores, 320
relational modeling, 176, 311
relational storage, 368, 370–371, 375–377
reliability, 8
replay of events, 321–323. *See also* event sourcing (ES)
  vs. aggregate snapshots, 343, 348
  psuedocode, 338–340
repositories, 131, 179–181, 212
  for aggregates, 199, 346–347, 363–364
  benefits, 356
  caching, 366–367
  connection strings in, 181
  in CQRS query stack, 268
  defined, 208, 355
  downloader and uploader components, 366
  for events, 346
  granularity, 357
  implementing, 359–368
  interface for, 207–208, 358–359
  IQueryable types, returning, 360–362
  query methods, 359–363
  in read models, 270
  read operations, allowing, 266
  storage technologies, 364–368
  for write actions, 358

Repository pattern, 355–359
  CQRS and, 357
  Domain Model and, 357–358
requirements, 7–11
  acknowledging, 18, 29
  analyzing, 29
  changing over time, 15
  churn in, 43
  communicating, 118. *See also* ubiquitous language
  continuous refinement of, 4
  defined, 7
  functional, 7–9
  gathering, 10–11
  grouping raw requirements, 11, 30
  learning about problems to be solved, 11, 30
  levels of, 28
  mapping to features, 29
  new or changed, 29
  nonfunctional, 7, 9–10
  pertinent classes from, 58–59
  processing, 11
  testing software against, 94–95
  ubiquitous language for, 18. *See also* ubiquitous language
  when dimension, 8–9, 209–210, 313
ReSharper (JetBrains), 34, 40–41
Responsibility-Driven Design (RDD), 177
Responsive Web Design (RWD), 156–157
reusability, 33, 61, 178
rewrites, 44
Rhino Mocks, 92
rich clients over the web, 164–165
rich-Model, 184
rigidity of software, 32–33
runtime errors, Liskov's principle and, 68

S

saga components, 281–283, 285
  aggregate and event handling, 347–348
  behavior flowcharts, 305
  commands and events in, 304–309
  designing, 303–305
  implementation code, 334–335
  implementing, 302–303
  in memory, 333
  persistence, 308
  starting, 306
Saint-Exupéry, Antoine de, 73
scaffolding, 221
  domain entities, 228–231
  value objects, 231–235
scalability
  CQRS systems, 259
  defined, 10
  extreme, 10
  tiers, multiple, 182
schedules of software project development, 37
schema volatility, 376
Schwaber, Ken, 17
screens, view model classes for, 140, 144
scripting objects vs. domain modeling, 170
Scrum methodology, 17, 37–38
sealed classes, 13–14
security, 10
  placement of code, 213–214
separation of concerns (SoC), 57, 131, 147, 221, 274, 356
server-side adaptive solutions for devices, 158–159
server-side device detection, 157–158
Service Locator (SL) pattern, 60, 70–71
Service-Oriented Architecture (SOA), 19
services. *See also* domain services
  domain logic in, 174
setters, private, 264–266
shared entities, 125
shared kernel relational pattern, 126
SharpTestex, 90
Silverlight, 164–165
simplicity, 73, 87
single-page applications (SPA), 160–163
  costs, 163
  external services, 366
  presentation logic in, 160–161
  server, transferring data from, 162–163
  task triggering, 275
  workflow, 161
Single Responsibility principle (SRP), 56, 65
sketches, 141–144
snapshot databases, 331, 336, 340–341
snapshots, data, 323, 340–342
snippets, 39
software anemia, 226–227
software architects, 4, 12, 17–24, 32, 305
software architecture, 3–7. *See also* architecture
software components. *See also* classes; methods
  extensible design, 102–103
  responsibilities and roles of, 177
  testing priority, 100–101
software contracts, 79–83
  Design by Contract, 79
software design principles, 53–83
  coding vectors, 72–75
  cohesion, 55–56
  coupling, 56
  defensive programming, 78–83
  isolation, 57

software design principles (*continued*)
maintainability, 8, 53, 55, 83, 105–106, 118
of object-oriented design, 57–64
patterns, 75–77
refactoring, 77–78
separation of concerns, 57. *See also* separation of concerns (SoC)
SOLID principles, 64–69
software development methodologies, 14–15
software disasters, 27, 38. *See also* Big Ball of Mud (BBM)
software engineering, 54
software functionality, 57
Software Requirement Patterns (Withall), 10
software reuse, 33
software systems
aspects and features, 12–14
Behavior-Driven Design aware, 95–96
Big Ball of Mud, threat of, 28
breakdown process, 11–12
building, 4. *See also* architecture
building incrementally, 16–17
code writing process, improving, 38–43
complexity of, 1
context, 6
continuous change, 43
development team. *See* development team
diagram, 6
disasters, recovering from, 43–50
estimates on development costs, 30–31
fixed price vs. time/materials mode, 49–50
growth of, 50
lifespan of, 50
managers and developers, 36
official language of, 120. *See also* ubiquitous language
openness and transparency, 32
organizational culture and, 35–38
organization of, 5
ownership by individuals, 35
ownership of, 31–32
prevention of vs. recovery from disaster, 50
purpose, capturing, 28–30
quality characteristics, 7–8
redesigning, 44
stakeholders, 5, 9–10, 28
successful, 27
successfully designed, 27
unit testing, 93. *See also* unit tests
SOLID principles, 54, 64–69, 86
solution architects, 4
source control systems, 41
spaghetti code, 54
Special Case pattern, 242–243
specifications, 12, 20
Spolski, Joel, 65
sprint zero development methodology, 17
SQL Server, layered expression trees on, 273
StackOverflow user, 106
stakeholders
acceptance testing, 94–95
communicating with, 32, 118. *See also* ubiquitous language
defined, 5
meeting with, 21, 29, 312
representation of, 22
requested system attributes, 9–10
requirements of, 28
stale data, 287–288
state
last-known good state, 313–315
rebuilding, 321–323. *See also* event sourcing (ES)
state machines, 103
statement coverage, 98
static code analysis, 34–35
Steve Jobs (Isaacson), 36
storage solutions, 286–288
planning, 375–377
storage technologies, 364–368
stored procedures, 129
strategic model design, 116–118
Stroustrup, Bjarne, 289
Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design (Constantine and Yourdon), 55
structured programming, 54–57
subdomains, 115–116, 122–124
subsystems, 35
supporting architectures, 127–129
for bounded contexts, 196
CQRS, 255–256. *See also* Command/Query Responsibility Segregation (CQRS)
Domain Model, 191. *See also* Domain Model
event sourcing, 311. *See also* event sourcing (ES)
switch statements, 103

T

Table Module pattern, 226
Tabular NoSQL store, 370
task-based designs, 138–139, 160
task-based modeling, 169, 176–182
task-based user interfaces, 131, 260
tasks
classes built around, 65
in CQRS, 274
implementing, 209
interactively triggered, 274–276
orchestrating, 133, 148–149, 160
in plugin architecture, 103

tasks (*continued*)
programmatically triggered, 276
splitting, 14
in state machines, 103
teams. *See* development team
technical debt, 34–35, 85
technical people, language of, 10–11
technical specifications, 20
technologies
choosing, 20
identifying and evaluating, 19–20
Tell-Don't-Ask principle, 74–75, 177
templates, 39
testability, 85–88, 109
controllers and, 148
principles of, 87
return-on-investment, 88
test coverings, 47
test doubles, 91–93
Test-Driven Development (TDD), 94, 97–98
test harnesses, 88–89
testing, 88–101
assertions, number of, 90
automated, 86, 101
code coverage, 98–99
dependency injection and, 72
domain layer, 100–101
layered expression trees, 274
priority of, 100–101
for refactoring, 47–48
relevancy of, 99
responsibilities of, 14
Test-Driven Development process, 97–98
timing of, 31, 98
Thomas, Dave, 74, 101
Thompson, Ken, 349
tiers, 181–182
data flow, 182, 185
defined, 48
layered expression trees and, 273
vs. layers, 129
tight coupling, 56–57, 180, 202
Tomlin, Lily, 378
tools. *See also* Object/Relational Mapper (O/RM) tools
for coding assistance, 39–41, 67, 78, 109
design pattern books, 76
knowledge silos, 35
for productivity, 72
source-code management, 41
static code analysis, 34–35, 45
for testing, 99, 101
for UX development, 141–142
Torvalds, Linus, 217
training of developers, 41, 50
transactional consistency, 200
transactions
atomic and global methods, 364
handling, 354
Unit of Work, 357
Transaction Script (TS) pattern, 169–172, 262–263, 292
Twitter, architectural refactoring, 43
Twitter Bootstrap, 222
Typemock, 92

U

ubiquitous language, 11, 118–121
commands, granularity of, 309
defining, 119–120
and domain model, syncing, 121
domain services, naming, 180–181
first-class concepts, 209
methods, naming, 240
purpose of, 118–119
for requirements, 18
structure of, 119
vocabulary, populating, 115
UI logic, 147
undo functionality, 336–337, 343
unforeseen events, 37
Unified Modeling Language (UML), 138–139
Unit of Work (UoW) pattern, 244, 357, 363–364
unit tests, 31, 88–93
Arrange-Act-Assert (AAA) pattern, 93
automating generation of, 101
from SpecFlow tests, 96
writing, 101
Unity, 72
Unity.Mvc library, 179
universal application projects, 147
unsealed classes, 13–14
upfront architecture, 15, 135
upstream contexts, 125–126
usability, 8
usability reviews, 141
use-cases
commands, orchestrating with, 294
diagrams of, 138–139
domain-driven expressions of, 121
implementation, 132–133
pertinent classes from, 58–59
user experience (UX), 141
user experience first (UX-first), 138–146
designing, 143–146
interactions, focusing on, 138–140
tools for, 141–142
vs. user interface, 140
UX experts, 141

user interface (UI), 147
data flow from, 248
designing, 138–140
F5/Refresh effect, 276
refreshing with redirect call, 276
screens, view model classes for, 140
task-based, 138–139, 260
in web applications, 151–152
user-oriented quality characteristics, 8
users entities, 235–238
UX architects, 141
UX designers, 141
UXPin, 142

V

validation, 213
value objects, 198
data structure, 231–232
equality, 232–233
expressivity, 234–235
operator overloading, 234
scaffolding, 231–235
vertical scrolling, minimizing, 109
view model classes, 140, 144, 248
view model objects, 146
view models, 131, 140, 144, 183, 272
views, 146
prototypes, creating from, 144
virtual methods, Liskov's principle and, 68
viscosity of design, 34
visibility, 87
Visual Studio
Code Contracts rewriter, 83
Code Coverage tool, 99
debugging tools, 86
logical layers, separating into projects, 221
Pex add-in, 101
SpecFlow, 95–96
static-code analyzer, 34
Vlissides, John, 58

W

Waseem, Muhammad, 349
Waterfall development methodologies, 15
Web API, 150, 154
web applications, 147
single-page applications, 160–163
task triggering, 275
user interface, 151–152
Web Forms, 151–155
vs. ASP.NET MVC, 152–155
web front ends, 164
web rich clients, 164–165
websites
ASP.NET websites, 147–152
device support, 155–160
what-if scenarios and event sourcing, 316, 322
white-box testing, 101
Windows Communication Foundation (WCF) service, 150
Windows Presentation Foundation (WPF), rich client front ends, 164
wireframes, 141–144
Wirfs-Brock, Rebecca, 177
Wirify, 142
Withall, Stephen, 10
workarounds, 33–34
worker services, 149
workflows
after user action, 100, 132–133
command and event definition, 284–285, 302. *See also* saga components
orchestration of, 150, 252
of single-page applications, 161
sketching, 138, 142
tasks in, 177
wrapper classes, 62
write operations. *See also* commands
async, 259
external services for, 366
model for, 241
writing code. *See* coding; development
WURFL, 157, 222

X

XPO, 365
xUnit, 89

Y

YAGNI (You Ain't Gonna Need It), 73–74, 107
Yoder, Joseph, 28
Young, Greg, 260
Yourdon, Edward, 55

About the authors

Dino Esposito is CTO and cofounder of Crionet, a startup providing software and IT services to professional tennis and sports companies. Dino still does a lot of training and consulting and is the author of several books on web and mobile development. His most recent books are *Architecting Mobile Solutions for the Enterprise* and *Programming ASP.NET MVC*, both from Microsoft Press. Dino speaks regularly at industry conferences, including Microsoft TechEd and DevConnections, and premiere European events, such as Software Architect, DevWeek, SDD, and BASTA. A technical evangelist covering Android and Kotlin development for JetBrains, Dino is also on the development team of WURFL—the ScientiaMobile database of mobile device capabilities that is used by large organizations such as Facebook.

You can follow Dino on Twitter at @despos and through his blog: (*http://software2cents.wordpress.com*).

Andrea Saltarello is CEO and founder of Managed Designs (*http://www.manageddesigns.it*), a company providing consultancy services related to software design and development.

A solution architect, Andrea is eager to write code in real projects to get feedback related to his architectural decisions. Andrea is also the lead developer of MvcMate, an open source project aimed at providing useful extensions to the ASP.NET MVC toolkit.

As a trainer and speaker, Andrea had several speaking engagements for courses and conferences across Europe, such as BASTA! Italia, DevWeek and Software Architect. He has also taught "Operating Systems" during the "Master in Editoria Multimediale" class organized by the University “Politecnico of Milan.”

In 2001 Andrea co-founded UGIdotNET (*http://www.ugidotnet.org*), the first Italian .NET User Group, of which he is President and leader.

Andrea is passionate about sports and music, and grew up devoted to volleyball and to Depeche Mode, which he fell in love with upon first listening to Everything Counts.

You can follow Andrea on Twitter at @andysal74 and through his blog (*http://blogs.ugidotnet.org/mrbrightside*)