

# Clean Architecture with .NET



 Professional

Dino Esposito

FREE SAMPLE CHAPTER |



# Clean Architecture with .NET

Dino Esposito

## Clean Architecture with .NET

Published with the authorization of Microsoft Corporation by:

Pearson Education, Inc., Hoboken, New Jersey

Copyright © 2024 by Dino Esposito.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit [www.pearson.com/permissions](http://www.pearson.com/permissions).

No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-13-820328-3

ISBN-10: 0-13-820328-8

Library of Congress Control Number: 2024930932

\$PrintCode

### Trademarks

Microsoft and the trademarks listed at <http://www.microsoft.com> on the “Trademarks” webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

### Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author, the publisher, and Microsoft Corporation shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the programs accompanying it.

### Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the U.S., please contact [intlcs@pearson.com](mailto:intlcs@pearson.com).

### Editor-in-Chief

Brett Bartow

### Executive Editor

Loretta Yates

### Associate Editor

Shourav Bose

### Development Editor

Kate Shoup

### Managing Editor

Sandra Schroeder

### Senior Project Editor

Tracey Croom

### Copy Editor

Dan Foster

### Indexer

Ken Johnson

### Proofreader

Jennifer Hinchliffe

### Technical Editor

Milan Jovanovic

### Editorial Assistant

Cindy Teeters

### Cover Designer

Twist Creative, Seattle

### Compositor

codeMantra

### Graphics

codeMantra

# Contents at a Glance

	<i>Introduction</i>	xv
<b>PART I</b>	<b>THE HOLY GRAIL OF MODULARITY</b>	
CHAPTER 1	The quest for modular software architecture	3
CHAPTER 2	The ultimate gist of DDD	23
CHAPTER 3	Laying the ground for modularity	47
<b>PART II</b>	<b>ARCHITECTURE CLEANUP</b>	
CHAPTER 4	The presentation layer	65
CHAPTER 5	The application layer	91
CHAPTER 6	The domain layer	133
CHAPTER 7	Domain services	169
CHAPTER 8	The infrastructure layer	189
<b>PART III</b>	<b>COMMON DILEMMAS</b>	
CHAPTER 9	Microservices versus modular monoliths	219
CHAPTER 10	Client-side versus server-side	255
CHAPTER 11	Technical debt and credit	285
	<i>Index</i>	307

*This page intentionally left blank*

# Contents

<i>Acknowledgments</i> .....	<i>xiii</i>
<i>Introduction</i> .....	<i>xv</i>

## **PART I THE HOLY GRAIL OF MODULARITY**

---

<b>Chapter 1</b>	<b>The quest for modular software architecture</b>	<b>3</b>
	In the beginning, it was three-tier .....	4
	Core facts of a three-tier system .....	4
	Layers, tiers, and modularization .....	7
	The DDD canonical architecture .....	9
	The proposed supporting architecture .....	9
	Adding more to the recipe .....	12
	Different flavors of layers .....	17
	Hexagonal architecture .....	17
	Clean architecture .....	18
	Feature-driven architecture .....	20
	Summary .....	22
<b>Chapter 2</b>	<b>The ultimate gist of DDD</b>	<b>23</b>
	Design driven by the domain .....	23
	Strategic analysis .....	24
	Tactical design .....	26
	DDD misconceptions .....	27
	Tools for strategic design .....	29
	Ubiquitous language .....	29
	A domain-specific language vocabulary .....	29
	Building the glossary .....	31
	Keeping business and code in sync .....	33
	The bounded context .....	36
	Making sense of ambiguity .....	37

Devising bounded contexts .....	39
The context map .....	42
Upstream and downstream .....	42
An example context map .....	43
An example deployment map .....	44
Summary .....	45
<b>Chapter 3 Laying the ground for modularity</b>	<b>47</b>
Aspects and principles of modularization .....	48
Separation of concerns .....	48
Loose coupling .....	49
Reusability .....	49
Dependency management .....	50
Documentation .....	50
Testability .....	50
Applying modularization .....	51
The presentation layer: interacting with the outside world .....	51
The application layer: processing received commands .....	51
The domain layer: representing domain entities .....	52
The data/infrastructure layer: persisting data .....	52
Achieving modularity .....	52
More modularity in monoliths .....	52
Introducing microservices .....	54
The simplest solution ever .....	56
Maintainability .....	57
Designing for testability .....	58
Summary .....	60
<b>PART II ARCHITECTURE CLEANUP</b>	
<b>Chapter 4 The presentation layer</b>	<b>65</b>
Project Renoir: the final destination .....	66
Introducing the application .....	66
The abstract context map .....	68

Designing the physical context map .....	71
Business requirements engineering.....	74
Breakdown of software projects .....	75
Event-based storyboards.....	76
Fundamental tasks of Project Renoir.....	77
Boundaries and deployment of the presentation layer.....	79
Knocking at the web server's door.....	79
ASP.NET application endpoints.....	80
Presentation layer development.....	82
Connecting to business workflows.....	82
Front-end and related technologies.....	86
API-only presentation.....	88
Summary .....	89

## **Chapter 5 The application layer 91**

An architectural view of Project Renoir.....	91
The access control subsystem .....	92
The document-management subsystem .....	94
Project Renoir in Visual Studio.....	95
Task orchestration .....	96
What is a task, anyway? .....	96
An example distributed task.....	97
An example task in Project Renoir .....	99
Data transfer .....	99
From the presentation layer to the application layer .....	100
From the application layer to the persistence layer .....	104
Implementation facts.....	106
Outline of an application layer.....	106
Propagating application settings .....	110
Logging.....	113
Handling and throwing exceptions .....	119



	Caching and caching patterns . . . . .	123
	Injecting SignalR connection hubs . . . . .	126
	Boundaries and deployment of the application layer . . . . .	129
	The dependency list . . . . .	129
	Deployment options . . . . .	129
	Summary . . . . .	131
<b>Chapter 6</b>	<b>The domain layer</b>	<b>133</b>
	Decomposition of the domain layer . . . . .	133
	The business domain model . . . . .	133
	Helper domain services . . . . .	137
	Devising a domain model . . . . .	138
	Shifting focus from data to behavior . . . . .	138
	Life forms in a domain model . . . . .	141
	The domain model in Project Renoir . . . . .	145
	The hitchhiker's guide to the domain . . . . .	147
	Treating software anemia . . . . .	148
	Common traits of an entity class . . . . .	149
	Rules of etiquette . . . . .	152
	Style conventions . . . . .	161
	Writing truly readable code . . . . .	165
	Summary . . . . .	168
<b>Chapter 7</b>	<b>Domain services</b>	<b>169</b>
	What is a domain service, anyway? . . . . .	170
	The stateless nature of domain services . . . . .	170
	Marking domain service classes . . . . .	170
	Domain services and ubiquitous language . . . . .	171
	Data access in domain services . . . . .	172
	Data injection in domain services . . . . .	172
	Common scenarios for domain services . . . . .	173
	Determining the loyalty status of a customer . . . . .	173
	Blinking at domain events . . . . .	174

Sending business emails .....	174
Service to hash passwords .....	175
Implementation facts .....	176
Building a sample domain service .....	176
Useful and related patterns .....	179
The REPR pattern adapted .....	180
Open points .....	184
Are domain services really necessary? .....	184
Additional scenarios for domain services .....	187
Summary .....	187

## **Chapter 8 The infrastructure layer 189**

Responsibilities of the infrastructure layer .....	190
Data persistence and storage .....	190
Communication with external services .....	190
Communication with internal services .....	191
Implementing the persistence layer .....	192
Repository classes .....	193
Using Entity Framework Core .....	196
Using Dapper .....	205
Hosting business logic in the database .....	207
Data storage architecture .....	208
Introducing command/query separation .....	208
An executive summary of event sourcing .....	213
Summary .....	215

## **PART III COMMON DILEMMAS**

---

## **Chapter 9 Microservices versus modular monoliths 219**

Moving away from legacy monoliths .....	220
Not all monoliths are equal .....	220
Potential downsides of monoliths .....	221
Facts about microservices .....	224

Early adopters . . . . .	224
Tenets of a microservices architecture and SOA . . . . .	224
How big or small is “micro”? . . . . .	225
The benefits of microservices. . . . .	227
The gray areas. . . . .	229
Can microservices fit all applications? . . . . .	235
The big misconception of big companies . . . . .	235
SOA and microservices. . . . .	237
Are microservices a good fit for your scenario? . . . . .	237
Planning and deployment. . . . .	241
Modular monoliths . . . . .	245
The delicate case of greenfield projects . . . . .	246
Outlining a modular monolith strategy for new projects . . . . .	247
From modules to microservices . . . . .	249
Summary . . . . .	253

**Chapter 10 Client-side versus server-side 255**

A brief history of web applications. . . . .	256
The prehistoric era. . . . .	256
The server-scripting era. . . . .	257
The client-scripting era. . . . .	260
Client-side rendering . . . . .	262
The HTML layer. . . . .	263
The API layer . . . . .	266
Toward a modern prehistoric era . . . . .	269
Server-side rendering. . . . .	273
Front-end–back-end separation. . . . .	274
ASP.NET front-end options. . . . .	275
ASP.NET Core versus Node.js . . . . .	278
The blocking/non-blocking saga . . . . .	280
Summary . . . . .	283

<b>Chapter 11</b>	<b>Technical debt and credit</b>	<b>285</b>
	The hidden cost of technical debt .....	285
	Dealing with technical debt .....	286
	Ways to address debt .....	288
	Debt amplifiers .....	290
	The hidden profit of technical credit .....	293
	The theory of broken windows .....	293
	The power of refactoring .....	295
	Do things right, right away .....	297
	Summary .....	299
	<i>Index</i> .....	301

*This page intentionally left blank*

# Acknowledgments

As hair thins and grays, memories return of when I was the youngest in every meeting or conference room. In 30 years of my career, I witnessed the explosion of Windows as an operating system, the rise of the web accompanied by websites and applications, and then the advent of mobile and cloud technologies.

Several times, I found myself having visions related to software technology developments, not too far from what happened a few years later. At other times, I surprised myself by formulating personal projects halfway between dreams and ambitious goals.

The most unspoken of all is the desire to travel the world, speaking at international conferences without the pressure to talk about what is cool and trendy but only about what I have seen and made work—without mincing words and without filters or reservations. To do this, I needed to work—finally—daily on the development of real applications that contributed to some kind of business and simplified the lives of some kind of audience.

Thanks to Crionet and KBMS Data Force, this is now a reality.

After many years, I have a full-time position (CTO of Crionet), a team of people grown in a few years from juniors to bold and capable professionals, and the will to share with everyone a recipe for making software that is neither secret nor magical.

I have nothing to sell; only to tell. And this book is for those who want to listen.

This book is for Silvia and Francesco.

This book is for Michela.

This book is for Giorgio and Gaetano.

This book was made possible by Loretta and Shourav and came out as you're getting it thanks to Milan, Tracey, Dan, and Kate.

This book is my best until the next one!

*This page intentionally left blank*

# Introduction

I graduated in Computer Science in the summer of 1990. At the time, there were not many places in Europe to study computers. The degree course was not even set up with its own Computer Science faculty but was an extension of the more classical faculty of Mathematics, Physics, and Natural Sciences. Those with strong computer expertise in the 1990s were really cool people—in high demand but with unclear career paths. I started as a Windows developer. Computer magazines were popular and eagerly awaited every month. I dreamt of writing for one of them. I won the chance to do it once and liked it so much that I'm still doing it today, 30 years later.

My passion for sharing knowledge was so intense that five years after my first serious developer job it became my primary occupation. For over two decades all I did was write books and articles, speak at conferences, teach courses, and do occasional consulting. Until 2020, I had a very limited exposure to production code and the routine of day-by-day development. Yet, I managed to write successful books for those who were involved in real-world projects.

Still, in a remote area of my mind was a thorny doubt: Am I just a lecture type of professional or am I also an action person? Will I be able to ever build a real-world system? The pandemic and other life changes brought me to ultimately find an answer.

I faced the daunting task of building a huge and intricate system in a fraction of the time originally scheduled that the pandemic sharply cut off. No way to design, be agile, do testing and planning—the deadline was the only certain thing. I resorted to doing—and letting a few other people do—just what I taught and had discovered while teaching for years. It worked. Not just that. Along the way, I realized that the approach we took to build software, and related patterns, also had a name: clean architecture. This book is the best I know and have learned in three years of everyday software development after over two decades of learning, teaching, and consulting.

In our company, we have several developers who joined as juniors and have grown up using and experimenting with the content of this book. It worked for us; I hope it will work for you, too!



## Who should read this book

---

Software professionals are the audience for this book, including architects, lead developers, and—I would say, especially—developers of any type of .NET applications. Everyone who wants to be a software architect should find this book helpful and worth the cost. And valid architects are, for the most part, born developers. I strongly believe that the key to great software passes through great developers, and great developers grow out of good teachers, good examples, and—hopefully—good books and courses.

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

This book expects that you have at least a minimal understanding of .NET development and object-oriented programming concepts. 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 or fancy strings in square brackets that hyperlink to some old paper listed in a bibliography at the end of the book. It's a book about building systems in the 2020s and facing the dilemmas of the 2020s, from the front end to the back end, passing through cloud platforms and scalability issues.

## This book might not be for you if...

---

If you're seeking a reference book or you want to find out how to use a given pattern or technology then this book might not for you. Instead, the goal is sharing and transferring knowledge so that you know what to do at any point. Or, at least, you now know what other guys—Dino and team—did in an analogous situation.

## Organization of this book

---

In all, modern software architecture has just one precondition: modularity. Whether you go with a distributed, service-oriented structure, a microservices fragmented pattern, or a compact monolithic application, modularity is crucial to build and manage the codebase and to further enhance the application following the needs of the business. Without modularity, you can just be able to deliver a working system once, but it will be hard to expand and update it.

Part I of this book, titled “The Holy Grail of modularity,” lays the foundation of software modularity, tracing back the history of software architecture and summarizing the gist of domain-driven design (DDD)—one of the most helpful methodologies for breaking down business domains, though far from being an absolute necessity in a project.

Part II, “Architecture cleanup,” is about the five layers that constitute, in the vision of this book, a “clean” architecture. The focus is not much on the concentric rendering of the architecture, as popularized by tons of books and articles, but on the actual value delivered by constituent layers: presentation, application, domain, domain services, and infrastructure.

Finally, Part III, “Common dilemmas,” focuses on three frequently faced stumbling blocks: monoliths or microservices, client-side or server-side for the front end, and the role and weight of technical debt.

## Downloads: reference application

---

Part II of the book describes a reference application, Project Renoir, whose entire code-base is available on GitHub at:

*<https://github.com/Youbiquitous/project-renoir>*

A zipped version of the source code is also available for download at *[MicrosoftPressStore.com/NET/download](https://MicrosoftPressStore.com/NET/download)*.



**Note** The reference application requires .NET 8 and is an ASP.NET application with a Blazor front end. It uses Entity Framework for data access and assumes a SQL Server (any version) database.

## Errata, updates, and book support

---

We’ve made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

*[MicrosoftPressStore.com/NET/errata](https://MicrosoftPressStore.com/NET/errata)*

If you discover an error that is not already listed, please submit it to us at the same page.

For additional book support and information, please visit  
*MicrosoftPressStore.com/Support*.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *http://support.microsoft.com*.

## **Stay in touch**

---

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

# The ultimate gist of DDD

*Get your facts first, and then you can distort them as much as you please.*

—Mark Twain

Domain-driven design (DDD) is a 20-year-old methodology. Over the years, there have been several books, learning paths, and conferences dedicated to it, and every day, various social networks archive hundreds of posts and comments about it. Still, although the essence of DDD remains surprisingly simple to grasp, it is much less simple to adopt.

Today more than ever, software adds value only if it helps streamline and automate business processes. For this to happen, the software must be able to faithfully model segments of the real world. These segments are commonly referred to as *business domains*.

For a few decades, client/server, database-centric applications have provided an effective way to mirror segments of the real world—at least as those segments were perceived at the time. Now, though, working representations of segments of the real world must become much more precise to be useful. As a result, a database with just some code around is often no longer sufficient. Faithfully mirroring real-world behaviors and processes requires an extensive analysis.

What does this have to do with DDD? Ultimately, DDD has little to do with actual coding. It relates to methods and practices for exploring the internals of the business domain. The impact of DDD on coding and on the representation of the real world depends on the results of the analysis.

DDD is not strictly required per se, but it is an effective method for exploring and understanding the internal structure of the business domain. What really matters is getting an accurate analysis of the domain and careful coding to reflect it. DDD systematizes consolidated practices to produce an architectural representation of the business domain, ready for implementation.

## Design driven by the domain

---

Conceptually, DDD is about design rather than coding. It rests on two pillars: one strategic and one tactical. The original authors of DDD outlined the strategy pillar and suggested tactics to achieve it. Today, however, I believe strategic analysis is the beating heart of DDD.

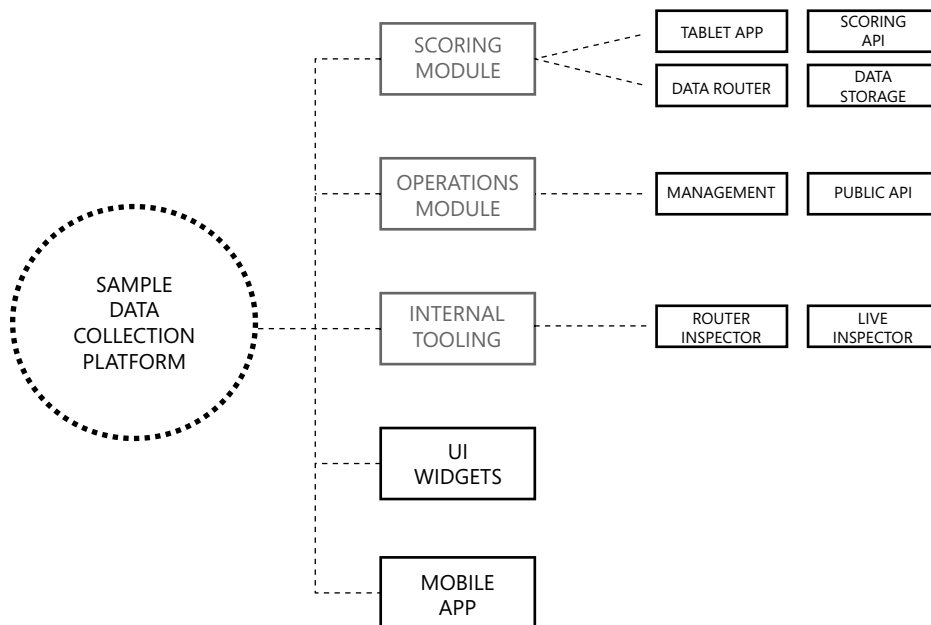
# Strategic analysis

Any world-class software application is built around a business domain. Sometimes, that business domain is large, complex, and intricate. It is not a natural law, however, that an application must represent an intricate business domain to be broken down into pieces with numerous and interconnected function points. The strategic analysis can easily return the same monolithic business domain you started from.

## Top-level architecture

The ultimate goal of the DDD strategic analysis is to express the top-level architecture of the business domain. If the business domain is large enough, then it makes sense to break it down into pieces, and DDD provides effective tools for the job. Tools like ubiquitous language (UL) and bounded contexts may help identify subdomains to work on separately. Although these subdomains may potentially overlap in some way, they remain constituent parts of the same larger ecosystem.

Figure 2-1 illustrates the conceptual breakdown of a large business domain into smaller pieces, each of which ultimately results in a deployed application. The schema—overly simplified for the purposes of this book—is adapted from a real project in sport-tech. The original business domain—a data-collection platform—is what stakeholders attempted to describe and wanted to produce. The team conducted a thorough analysis and split the original domain into five blocks. Three of these blocks were then further broken into smaller pieces. The result is 10 applications, each independent from the other in terms of technology stack and hosting model, but still able to communicate via API and in some cases sharing the same database.



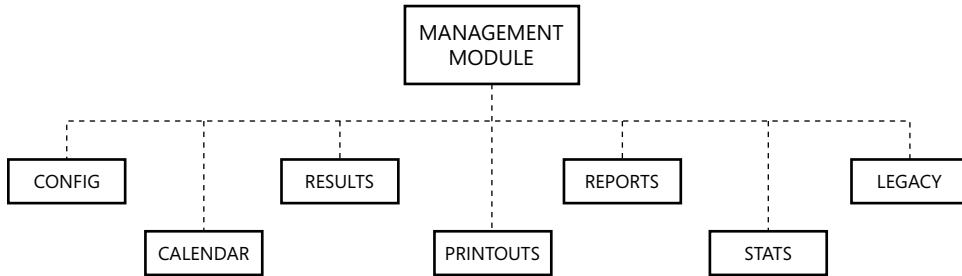
**FIGURE 2-1** Breakdown of a business domain.

## Business domain breakdown

Nobody really needs DDD (or any other specific methodology) to move from the dashed circle on the left of Figure 2-1 to the final list of 10 bold squares on the right. As hinted at earlier, DDD doesn't push new revolutionary practices; rather, it systematizes consolidated practices. With knowledge of the business and years of practice in software architecture, a senior architect might easily design a similar diagram without using DDD, instead relying on the momentum of experience and technical common sense. Still, although deep knowledge of a business domain might enable you to envision a practical way to break up the domain without the explicit use of an analytical method, DDD does provide a step-by-step procedure and guidance.

## Subdomains versus features

Recall the block labeled "Management" in Figure 2-1. This refers to a piece of functionality whose cardinality is not obvious. That is, whereas all the other blocks in Figure 2-1 reasonably map to a single leaf-level application, this one doesn't. Within the Management block, you could identify the functions shown in Figure 2-2.



**FIGURE 2-2** Further functional split of the Management module.

The question is, are these functions just features in a monolithic application or independent services? Should this block be broken down further?

Determining the ideal size of building blocks is beyond DDD. That task requires the expertise and sensitivity of the architect. In the actual project on which this example is based, we treated the Management module as a whole and treated the smaller blocks shown in Figure 2-2 as features rather than subdomains. Ultimately, the DDD breakdown of subdomains hinges on the invisible border of local functions. All the blocks in Figure 2-2 are objectively local to the Management module and not impactful or reusable within the global, top-level architecture. Hence, in the actual project we treated them as features.

## The confusing role of microservices

These days, at this point of the domain breakdown, one inevitably considers microservices. I'll return to microservices in Chapter 3, "Laying the ground for modularity," and in Chapter 9 "Microservices versus modular monoliths." Here, however, I would like to make a clear statement about microservices and DDD: DDD refers only to top-level architecture and breaks the business domain in modules known as

bounded contexts. A *bounded context* is an abstract element of the architectural design. It has its own implementation, and it can be based on microservices, but microservices are on a different level of abstraction than bounded context and DDD.



**Note** The term *microservices* refers to physical boundaries of deployable units, whereas the term *bounded contexts* refers to logical boundaries of business units. Technically, though, a microservice might implement all business functions of a bounded context. When this happens, calling it “micro” is a bit counterintuitive!

With reference to Figure 2-2, the question whether blocks are features of a domain or subdomains relates to top-level architecture. Once it is ascertained that the Management block is a leaf subdomain—namely, a bounded context—its recognized features in the implementation can be treated as in-process class libraries, functional areas, lambda functions, or even autonomous microservices. The abstraction level, though, is different.

## The actual scale of DDD solutions

Many articles and blog posts that discuss DDD and bounded contexts presume that the entire enterprise back end is the domain that needs to be decomposed. So, they identify, say, Sales, Marketing, IT, Finance, and other departments as bounded contexts on which to focus. Such a large-scale scenario is fairly uncommon, however; companies rarely plan a big rewrite of the entire back end. But should this happen, the number of architects involved at the top level of the design, as large as that may be, would be relatively small.

DDD is a design approach primarily used for designing and organizing the architecture of software systems. It’s not tied to a specific scale in terms of the size of the system. Instead, it focuses on the organization of domains and subdomains within the software. Since the beginning, it has been pushed as a method dealing with enterprise-scale applications, but it is also applicable and effective at a medium- and small-scale level.

## Tactical design

In general terms, strategy sets out what you want to achieve; tactics define how you intend to achieve it. Strategically, DDD provides tools to partition the business domain in smaller bounded contexts. Tactically, DDD suggests a default architecture to give life to each bounded context.

## The default supporting architecture

Chapter 1 presented the highlights of the default DDD supporting architecture—the layered architecture, whose inspiring principles are now at the foundation of clean architecture. The layered architecture evolved from the multi-tier architecture in vogue when DDD was first devised.

The DDD reference architecture, monolithic and OOP-friendly, is just one suggestion. It was ideal in 2004 but sufficiently abstract and universal to retain great value even now. Today, though, other options and variations exist—for example, command/query responsibility segregation (CQRS), event sourcing, and non-layered patterns such as event-driven patterns and microservices. The key point is that for a long time, with respect to DDD, applying the layered architecture and some of its side class modeling patterns has been the way to go, putting domain decomposition in the background.

## What’s a software model, anyway?

Beyond the preliminary strategic analysis, DDD is about building a software model that works in compliance with identified business needs. In his book *Domain-Driven Design: Tackling Complexity at the Heart of Software* (2003), author Eric Evans, uses the object-oriented programming (OOP) paradigm to illustrate building the software model for the business domain, and calls the resulting software model the *domain model*.

At the same time, another prominent person in the software industry, Martin Fowler—who wrote the foreword for Evans’ book—was using the same term (domain model) to indicate a design pattern for organizing the business logic. In Fowler’s definition, the domain model design pattern is a graph of interconnected objects that fully represent the domain of the problem. Everything in the model is an object and is expected to hold data and expose a behavior.

In a nutshell, in the context of DDD, the domain model is a software model. As such, it can be realized in many ways, such as OOP, functional, or CRUD. In contrast, the domain model design pattern as defined by Martin Fowler is just one possible way to implement such a software model.



**Important** In DDD, the outcome of the analysis of the business model is a software model. A *software model* is just the digital twin of the real business in software form. It doesn’t necessarily have to be an object-oriented model written following given standards.

## DDD misconceptions

The name conflict with Fowler’s design pattern—quite paradoxical in a methodology in which unambiguous language is key—sparked several misconceptions around DDD.

### The relevance of coding rules

The DDD definition details certain characteristics of the classes that participate in an object-oriented domain model: aggregates, value types, factories, behaviors, private setters, and so on. Having an object-oriented model, though, is neither mandatory nor crucial. To be crystal-clear, it’s not the extensive use of factory methods in lieu of unnamed constructors, or using carefully crafted value objects instead of loose primitive values, that makes a software project run on time and budget.

Put another way, blind observation of the coding rules set out in the DDD tactics guarantees nothing, and without a preliminary strategic design and vision, may generate more technical issues and



debt than it prevents. For example, using a functional approach in the design of the domain model is neither prohibited nor patently out of place. You're still doing DDD effectively even if you code a collection of functions or build an anemic object model with stored procedures doing the persistence work.

## The value of coding rules

When it comes to DDD coding rules, though, there's a flip side of the coin. Those rules—value types over primitive types, semantic methods over plain setters, factory methods over constructors, aggregates to better handle persistence—exist for a clear and valid reason. They enable you to build a software representation of the business model that is much more likely to be coherent with the language spoken in the business. If you don't first identify the language of the business (the ubiquitous language) and the context in which that language is spoken, the blind application of coding rules just creates unnecessary complexity with no added value.

## Database agnosticism

When you examine DDD, it's easy to conclude that the domain model should be agnostic of the persistence layer—the actual database. This is great in theory. In practice, though, no domain model is truly agnostic from the persistence.

Note, though, that the preceding sentence is not meant to encourage you to mix persistence and business logic. A clear boundary between business and persistence is necessary. (More on this in the next chapter.) The point of DDD is that when building an object-oriented software model to represent the business domain, persistence should *not* be your primary concern, period.

That said, however, be aware that at some point the same object model you may have crafted ignoring persistence concerns will be persisted. When this happens, the database and the API you may use to go to the database—for example, Entity Framework (EF) Core, Dapper, and so on—are a constraint and can't always be blissfully ignored. More precisely, blissfully ignoring the nature of the persistence layer—although a legitimate option—comes at a cost.

If you really want to keep the domain model fully agnostic of database concerns, then you should aim at having two distinct models—a domain model and a persistence model—and use adapters to switch between the two for each operation. This is extra work, whose real value must be evaluated case by case. My two cents are that a pinch of sane pragmatism is not bad at times.

## Language is not simply about naming conventions

DDD puts a lot of emphasis on how entities are named. As you'll soon see, the term *ubiquitous language* (*UL*) simply refers to a shared vocabulary of business-related terms that is ideally reflected in the conventions used to name classes and members. Hence, the emphasis on names descends from the need for code to reflect the vocabulary used in the real world. It's not a mere matter of choosing arbitrary descriptive names; quite the reverse. It's about applying the common language rules discovered in the strategic analysis and thoughtfully choosing descriptive names.

## Tools for strategic design

I've touched on the tools that DDD defines to explore and describe the business domain. Now let's look at them more closely.

You use three tools to conduct an analysis of a business model to build a conceptual view of its entities, services, and behavior:

- Ubiquitous language
- Bounded context
- Context mapping

By detecting the business language spoken in a given area, you identify subdomains and label them as bounded context of the final architecture. Bounded contexts are then connected using different types of logical relationships to form the final context map.



**Note** In the end, DDD is just what its name says it is: design driven by a preliminary, thorough analysis of the business domain.

## Ubiquitous language

---

As emphatic as it may sound, the creation of the software model for a business domain may be (fancifully) envisioned as the creation of a new world. In this perspective, quoting a couple of (sparse) sentences about the genesis of the universe from the Gospel of John may be inspiring:

- In the beginning was the Word
- The Word became flesh, and dwelt among us

Setting aside the intended meaning of “the Word,” and instead taking it literally and out of the original context, the *word* is given a central role in the beginning of the process and in the end it becomes substance. Ubiquitous language (UL) does the same.

## A domain-specific language vocabulary

As a doctor or an accountant, you learn at the outset a set of core terms whose meaning remains the same throughout the course of your career and that are—by design—understood by your peers, counterparts, and customers. Moreover, these terms are likely related to what you do every day. It's different if, instead, you are, say, a lawyer—or worse yet, a software architect or software engineer.

In both cases, you may be called to work in areas that you know little or nothing about. For example, as a lawyer, you might need to learn about high finance for the closing argument on a bankruptcy case. Likewise, as a software engineer in sport-tech, you would need to know about ranking and scoring rules to enable the application's operations to run week after week. In DDD, this is where having a UL fits in.

## Motivation for a shared glossary of terms

At the end of the day, the UL is a glossary of domain-specific terms (nouns, verbs, adjectives, and adverbs, and even idiomatic expressions and acronyms) that carry a specific and invariant meaning in the business context being analyzed. The primary goal of the glossary is to prevent misunderstandings between parties involved in the project. For this reason, it should be a shared resource used in all forms of spoken and written communication, whether user stories, RFCs, emails, technical documentation, meetings, or what have you.

In brief, the UL is the universal language of the business as it is done in the organization. In the book *Domain-Driven Design*, author Eric Evans recommends using the UL as the backbone of the model. Discovering the UL helps the team understand the business domain in order to design a software model for it.

## Choosing the natural language of the glossary

As you discover the UL of a business domain and build your glossary of terms, you will likely encounter a few unresolved issues. The most important is the natural language to use for the words in the glossary. There are a few options:

- Plain, universal English
- The customer’s spoken language
- The development team’s spoken language

While any answer might be either good or bad (or both at the same time), it can safely be said that there should be no doubt about the language to use when the team and the customer speak the same language. Beyond that, every other situation is tricky to address with general suggestions. However, in software as in life, exceptions do almost always apply. Once, talking DDD at a workshop in Poland, I heard an interesting comment: “We can’t realistically use Polish in code—let alone have Polish names or verbs appear in public URLs in web applications—as ours is an extremely cryptic language. It would be hard for everyone. We tend to use English regardless.”



**Note** In the novel *Enigma* (1995), author Robert Harris tells the story of a fictional character who deciphers stolen Enigma cryptograms during World War II. Once the character decrypts some code, though, he discovers the text looks as if it contains yet another level of cryptography—this one unknown. The mystery is solved when another cryptogram reveals the text to be a consecutive list of abbreviated Polish names!

If the language of the glossary differs from the language used by some involved parties, and translations are necessary for development purposes, then a word-to-word table is necessary to avoid ambiguity, as much as possible. Note, though, that ambiguity is measured as a function that *approaches* zero rather than reaches zero.

## Building the glossary

You determine what terms to include in the glossary through interviews and by processing the written requirements. The glossary is then refined until it takes a structured form in which natural language terms are associated with a clear meaning that meets the expectations of both domain (stakeholder) and technical (software) teams. The next sections offer a couple of examples.

### Choosing the right term

In a travel scenario, what technical people would call “deleting a booking” based on their database-oriented vision of the business, is better referred to as “canceling a booking,” because the latter verb is what people on the business side would use. Similarly, in an e-commerce scenario, “submitting an order form” is too HTML-oriented; people on the business side would likely refer to this action simply as “checking out.”

Here’s a real-world anecdote, from direct experience. While building a platform for daily operations for a tennis organization, we included a button labeled “Re-pair” on an HTML page, based on language used by one of the stakeholders. The purpose of the button was to trigger a procedure that allowed one player to change partners in a doubles tournament draw (in other words, as the stakeholder said, to “re-pair”). But we quickly learned that users were scared to click the button, and instead called the Help desk any time they wanted to “re-pair” a player. This was because another internal platform used by the organization (to which we didn’t have access) used the same term for a similar, but much more disruptive, operation. So, of course, we renamed the button and the underlying business logic method.

### Discovering the language

Having some degree of previous knowledge of the domain helps in quickly identifying all the terms that may have semantic relevance. If you’re entirely new to the domain, however, the initial research of hot terms may be like processing the text below.

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.

Verbs are potential actions, whereas nouns are potential entities. Isolating them in bold, the text becomes:

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.

The relationship between verbs and nouns is defined by the syntax rules of the language being used: subject, verb, and direct object. With reference to the preceding text,

- *Registered customer* is the subject
- *Redeem* is the verb
- *Voucher* is the direct object

As a result, we have two domain entities: (Registered-Customer and Voucher) and a behavior (Redeem) that belongs to the Registered-Customer entity and applies to the Voucher entity.

Another result from such an analysis is that the term used in the business context to indicate the title to receive a discount is *voucher* and only that. Synonyms like *coupon* or *gift card* should not be used. Anywhere.

## Dealing with acronyms

In some business scenarios, most notably the military industry, acronyms are very popular and widely used. Acronyms, however, may be hard to remember and understand.

In general, acronyms should not be included in the UL. Instead, you should introduce new words that retain the original meaning that acronyms transmit—unless an acronym is so common that not using it is a patent violation of the UL pattern itself. In this case, whether you include it in the UL is up to you. Just be aware that you need to think about how to deal with acronyms, and that each acronym may be treated differently.

Taken literally, using acronyms is a violation of the UL pattern. At the same time, because the UL is primarily about making it easier for everyone to understand and use the business language and the code, acronyms can't just be ignored. The team should evaluate, one by one, how to track those pieces of information in a way that doesn't hinder cross-team communication. An example of a popular and cross-industry acronym that can hardly be renamed is RSVP. But in tennis, the acronyms OP and WO, though popular, are too short and potentially confusing to maintain in software. So, we expanded them to Order-of-Play and Walkover.

## Dealing with technical terms

Yet another issue with the UL is how technical the language should be. Although we are focused on understanding the business domain, we do that with the purpose of building a software application. Inevitably, some spoken and written communication is contaminated by code-related terms, such as *caching*, *logging*, and *security*. Should this be avoided? Should we instead use verbose paraphrasing instead of direct and well-known technical terms? The general answer here is no. Instead, limit the use of technical terms as much as possible, but use them if necessary.

## Sharing the glossary

The value of a language is in being used rather than persisted. But just as it is helpful to have an English dictionary on hand to explain or translate words, it might also be useful to have a physical document to check for domain-specific terms.

To that end, the glossary is typically saved to a shared document that can be accessed, with different permissions, by all stakeholders. This document can be an Excel file in a OneDrive folder or, better yet, a file collaboratively edited via Microsoft Excel Online. It can even be a wiki. For example, with an in-house wiki, you can create and evolve the glossary, and even set up an internal forum to openly discuss features and updates to the language. A wiki also allows you to easily set permissions to control how editing takes place and who edits what. Finally, a GitBook site is another excellent option.



**Important** Any change to the language is a business-level decision. As such, it should always be made in full accordance with stakeholders and all involved parties. Terms of the language become software and dwell in code repositories. You should expect a one-to-one relationship between words and code, to the point that misunderstanding a term is akin to creating a bug, and wrongly naming a method misrepresents a business workflow.

## Keeping business and code in sync

The ultimate goal of the UL is not to create comprehensive documentation about the project, nor is it to set guidelines for naming code artifacts like classes and methods. The *real* goal of the UL is to serve as the backbone of the actual code. To achieve this, though, it is crucial to define and enforce a strong naming convention. Names of classes and methods should always reflect the terms in the glossary.



**Note** As strict as it may sound, you should treat a method that starts a process with a name that is different from what users call the same process as technical debt—no more, no less.

## Reflecting the UL in code

The impact of the UL on the actual code is not limited to the domain layer. The UL helps with the design of the application logic too. This is not coincidental, as the application layer is where the various business tasks for use cases are orchestrated.

As an example, imagine the checkout process of an online store. Before proceeding with a typical checkout process, you might want to validate the order. Suppose that you've set a requirement that validating the order involves ensuring that ordered goods are in stock and the payment history of the customer is not problematic. How would you organize this code?

There are a couple of good options to consider:

- Have a single Validate step for the checkout process in the application layer workflow that incorporates (and hides) all required checks.
- Have a sequence of individual validation steps right in the application layer workflow.

From a purely functional perspective, both options would work well. But only one is ideal in a given business context. The answer to the question of which is the most appropriate lies in the UL. If the UL calls for a validate action to be performed on an order during the checkout process, then you should go with the first option. If the vocabulary includes actions like check-payment-history or check-current-stock, then you should have individual steps in the workflow for just those actions.



**Note** If there's nothing in the current version of the UL to clarify a coding point, it probably means that more work on the language is required—specifically, a new round of discussion to break down concepts one more level.

## Ubiquitous language changes

There are two main reasons a UL might change:

- The team's understanding of the business context evolves.
- The business context is defined while the software application is designed and built.

The former scenario resulted in the idea of DDD more than 20 years ago. The business model was intricate, dense, and huge, and required frequent passes to define, with features and concepts introduced, removed, absorbed, or redesigned on each pass.



**Note** This type of iterative process usually occurs more quickly in the beginning of a project but slows down and perhaps almost stops at some point later. (This cycle might repeat with successive major releases of the software.)

The latter scenario is common in startup development—for example, for software specifically designed for a business project in its infancy. In this case, moving fast and breaking things is acceptable with both the software and the UL.

So, the UL might change—but not indefinitely. The development team is responsible for detecting when changes are needed and for applying them to the degree that business continuity allows. Be aware, though, that a gap between business language and code is, strictly speaking, a form of technical debt.

## Everyone makes mistakes

I have worked in the sport-tech industry for several years and have been involved in building a few platforms that now run daily operations for popular sports organizations. If tournaments run week after week, it's because the underlying software works. Sometimes, however, that software may still have design issues.

Yes, I do make mistakes at times, which result in design issues. More often, though, any design issues on my software exist because I'm pragmatic. To explain, let me share a story (with the disclaimer that this design issue will likely be sorted out by the time you read this).

Recently, my team adapted an existing software system for a different—though nearly identical—sport. One difference was that the new system did not need to support singles matches. Another difference was that points, rather than positions, would be used to order players in draws.

A segment of the domain layer and a few data repositories in the persistence layer used two properties—SinglesRank and DoublesRank. Initially, we didn't change anything in the naming (including related database tables). We simply stored doubles rankings in the DoublesRank property and left the SinglesRank property empty. Then, to use points instead of positions to order players in draws, I pragmatically suggested repurposing the otherwise-unused SinglesRank property—a perfectly effective solution that would require very minimal effort.

Just two weeks later, however, people began asking repeatedly what the heck the actual value of SinglesRank was. In other words, we experienced a gap between the UL and its representation in the code and data structures.

## Helpful programming tools

There are several features in programming languages to help shape code around a domain language. The most popular is support for classes, structs, records, and enum types. Another extremely helpful feature—at least in C#—is extension methods, which help ensure that the readability of the code is close to that of a spoken language.

An extension method is a global method that developers can use to add behavior to an existing type without deriving a new type. With extension methods, you can extend, say, the String class or even an enum type. Here are a couple of examples:

```
public static class SomeExtensions
{
    // Turns the string into the corresponding number (if any)
    // Otherwise, it returns the default value
    public static int ToInt(this string theNumber, int defaultValue = 0)
    {
        if (theNumber == null)
            return defaultValue;
        var success = int.TryParse(theNumber, var out calc);
        return success
            ? calc
            : defaultValue;
    }
    // Adds logic on top of an enum type
    public static bool IsEarlyFinish(this CompletionMode mode)
    {
        return mode == CompletionMode.Disqualified ||
            mode == CompletionMode.OnCourtRetirement ||
            mode == CompletionMode.Withdrawal;
    }
}
```



The first extension method extends the core String type to add a shortcut to turn the string to a number, if possible.

```
// With extension methods
var number = "4".ToInt();
// Without extension methods
int.TryParse("4", out var number);
```

Suppose you want to query all matches with an early finish. The code for this might take the following form:

```
var matches = db.Matches
    .Where(m => m.MatchCompletionMode.IsEarlyFinish())
    .ToList();
```

The benefit is having a tool to hide implementation details, so the actual behavioral logic can emerge.

## Value types and factory methods

Remember the misconceptions around DDD mentioned earlier in this chapter? I'm referring in particular to the relevance of coding rules.

DDD recommends several coding rules, such as using factory methods over constructors and value types over primitive types. By themselves, these rules add little value (hence, the misconception). However, in the context of the UL, these rules gain a lot more relevance. They are crucial to keeping language and code in sync.

For example, if the business domain involves money, then you'd better have a Money custom value type that handles currency and totals internally rather than manually pairing decimal values with hard-coded currency strings. Similarly, a factory method that returns an instance of a class from a named method is preferable to an unnamed constructor that is distinguishable from others only by the signature.

## The bounded context

---

Tweaking the business language and renaming classes and methods is tricky, but thanks to integrated development environment (IDE) features and plug-ins, it is not terribly problematic. However, failing to identify subdomains that are better treated independently could seriously undermine the stability of the whole solution.

No matter how hard you try, your UL will not be a unique set of definitions that is 100-percent unambiguous within your organization. In fact, the same term (for example, *customer*) might have different meanings across different business units. Like suitcases on an airport baggage belt that look alike, causing confusion among travelers, functions and names that look alike can cause problems in your solution.

Understanding differences between functions and names is crucial, and effectively addressing those differences in code is vital. Enter bounded contexts.

## Making sense of ambiguity

When analyzing a business domain, ambiguity occurs. Sometimes we run into functions that look alike but are not the same. When this occurs, developers often reveal an innate desire to create a unique hierarchy of highly abstracted entities to handle most scenarios and variations in a single place. Indeed, all developers have the secret dream of building a universal code hierarchy that traces back to a root Big-Bang object.

The reality is that abstraction is great—but more so in mathematics than in mere software. The great lesson we learn from DDD is that sometimes code fragmentation (and to some extent even code duplication) is acceptable just for the sake of maintenance.



**Note** Code duplication can be just the starting point that leads to a model that is ideal for the business. Experience teaches us that when two descriptions seem to point to the same entity (except for a few attributes), forcing them to be one is almost always a mistake; treating them as distinct entities is usually acceptable even if it is not ideal.

## The cost of abstraction

Abstraction always comes at a cost. Sometimes this cost is worth it; sometimes it is not.

Originally, abstraction came as a manna from heaven to help developers devise large domain models. Developers examined a larger problem and determined that it could be articulated as many smaller problems with quite a few things in common. Then, to combat code duplication, developers righteously added abstraction layers.

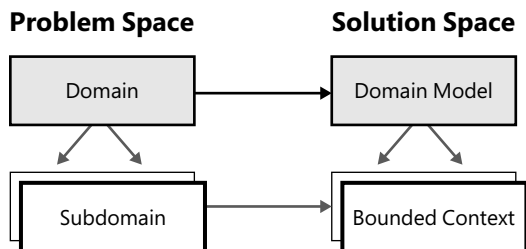
As you proceed with your analysis and learn about new features, you might add new pieces to the abstraction to accommodate variations. At some point, though, this may become unmanageable. The bottom line is that there is a blurred line between premature abstraction (which just makes the overall design uselessly complex) and intelligent planning of features ahead of time. In general, a reasonable sign that abstraction may be excessive is if you catch yourself handling switches in the implementation and using the same method to deal with multiple use cases.

So much for abstraction in coding. What about top-level architecture? With this, it's nearly the same issue. In fact, you might encounter a business domain filled with similar functions and entities. The challenge is understanding when it's a matter of abstracting the design and when it's breaking down the domain in smaller parts. If you break it down in parts, you obtain independent but connected (or connectable) functions, each of which remains autonomous and isolated.

## Using ambiguity as the borderline

A reasonable sign that you may need to break a business domain into pieces is if you encounter ambiguity regarding a term of the UL. In other words, different stakeholders use the same term to mean different things. To address such a semantic ambiguity, the initial step is to determine whether you really are at the intersection of two distinct contexts. One crucial piece of information is whether one term can be changed to a different one without compromising the coherence of the UL and its adherence to the business language.

An even subtler situation is when the same entity appears to be called with different names by different stakeholders. Usually, it's not just about having different names of entities (synonyms); it often has to do with different behaviors and different sets of attributes. So, what should you do? Use coding abstractions, or accept the risk of some duplication? (See Figure 2-3.)



**FIGURE 2-3** Domain and subdomains versus domain models and bounded contexts.

Discovering ambiguity in terms is a clear sign that two parts of the original domain could possibly be better treated as different subdomains, each of which assigns the term an unambiguous meaning. DDD calls a modeled subdomain a *bounded context*.



**Note** Realistically, when modeling a large domain, it gets progressively harder to build a single unified model. Also, people tend to use subtly different vocabularies in different parts of a large organization. The purpose of DDD is to deal with large models by dividing them into different bounded contexts and being explicit about their interrelationships.

## The savings of code duplication

From long experience in the code trenches, my hearty suggestion is that whenever you feel unsure whether abstraction is necessary, then by default, it isn't. In that case, you should use code duplication instead.

That said, I know that tons of articles and books out there (including probably a few of mine) warn developers of the “don't repeat yourself” (DRY) principle, which encourages the use of abstraction to reduce code repetitions. Likewise, I'm also well aware that the opposite principle—write every time (WET)—is bluntly dismissed as an anti-pattern.

Yet, I dare say that unless you see an obvious benefit to keeping a piece of the top-level architecture united, if a term has ambiguity within the business language that can't just be solved by renaming it using a synonym, you'd better go with an additional bounded context.

In coding, the cost of a bad abstraction is commonly much higher than the cost of duplicated code. In architecture, the cost of a tangled monolith can be devastating, in much the same way the cost of excessive fragmentation can be. Yes, as usual, it depends!

## Devising bounded contexts

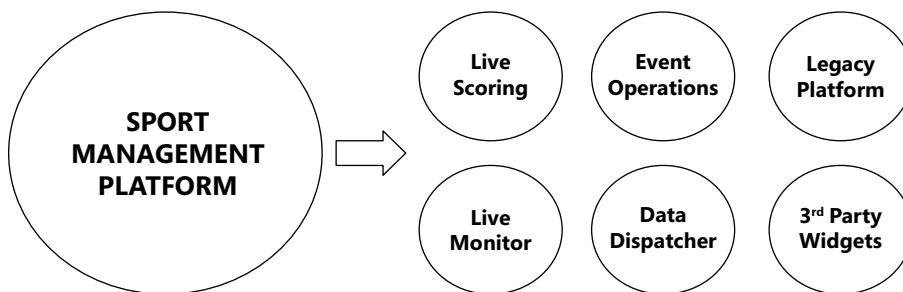
A bounded context is a segment of the original model that turns out to be better modeled and implemented as a separate module. A bounded context is characterized by three aspects:

- Its own custom UL
- Its own autonomous implementation and technology stack
- A public interface to other contexts, if it needs be connected

As a generally observed fact, the resulting set of bounded contexts born from the breakdown of a business domain tends to reflect (or at least resemble) the structure of the owner organization.

### Breakdown of a domain

Here's an example taken from a realistic sport-tech scenario. If you're called to build an entire IT system to manage the operations of a given sport, you can come up with at least the partitions in subdomains shown in Figure 2-4.



**FIGURE 2-4** Breakdown of an example domain model in a sport-tech scenario.

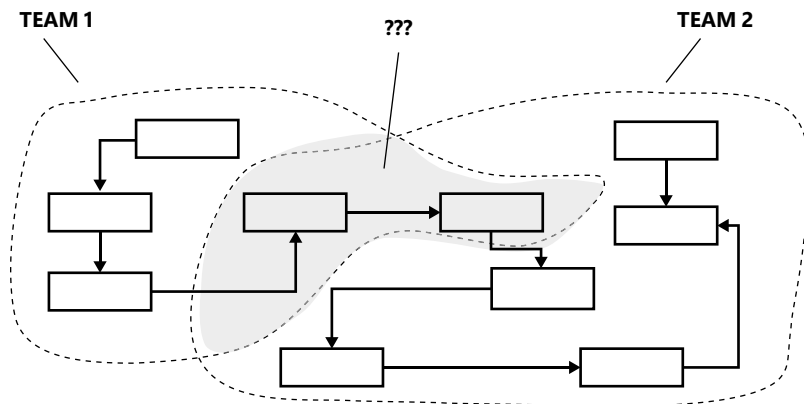
It's unrealistic to build the system as a single monolith. And it's not a matter of faith in the software creed of microservices; it's just that, with a decent analysis of the domain, processes, and requirements, you'll see quite a few distinct clusters of related operations (although maybe not just the six shown in Figure 2-4). These distinct blocks should be treated as autonomous projects for further analysis, implementation, and deployment.

In summary, each bounded context is implemented independently. And aside from some technical resources it may share with other contexts (for example, a distributed cache, database tables, or bus), it is completely autonomous from both a deployment and coding perspective.

## Shared kernels

Suppose you have two development teams working on what has been identified as a bounded context and you have an agreed-upon graph of functionalities in place. At some point, team 1 and team 2 may realize they are unwittingly working on the same small subset of software entities.

Having multiple teams share work on modules poses several synchronization issues. These range from just keeping changes to the codebase in sync to solving (slightly?) conflicting needs. Both teams must achieve coherency with their respective specs—not to mention any future evolutions that might bring the two teams into a fierce contrast. (See Figure 2-5.)



**FIGURE 2-5** Discovering a shared kernel.

There are three possible ways to deal with such a situation. The most conservative option is to let each team run its own implementation of the areas that appear common. Another option is to appoint one team the status of owner, giving it the final word on any conflicts. As an alternative, you could just let the teams come to a mutual agreement each time a conflict arises. Finally, there is the shared kernel option.

Shared kernel is a special flavor of bounded context. It results from a further breakdown of an existing bounded context. For example, the subdomain in Figure 2-5 will be partitioned in three contexts—one under the total control of team 1, one under the total control of team 2, and a third one. Who's in charge of the shared kernel? Again, the decision is up to the architect team, but it can be one of the existing teams or even a new team.

## Legacy and external systems

For the most part, bounded contexts isolate a certain related amount of behavior. Identifying these contexts is up to the architect team. However, certain pieces of the overall system should be treated as distinct bounded contexts by default—in particular, wrappers around legacy applications and external subsystems.

Whenever you have such strict dependencies on systems you don't control (or are not allowed to control), the safest thing you can do is create a wrapper around those known interfaces—whether a plain shared database connection string or an API. These wrappers serve a double purpose. First, they are an isolated part of the final system that simply call remote endpoints by proxy. Second, they can further isolate the general system from future changes on those remote endpoints.

In DDD jargon, the isolated wrappers around an external system are called an anti-corruption layer (ACL). Simply put, an ACL is a thin layer of code that implements a familiar pattern. It offers your calling modules a dedicated and stable (because you own it) programming interface that internally deals with the intricacies of the endpoints. In other words, the ACL is the only section of your code where the nitty-gritty details of the remote endpoints are known. No part of your code is ever exposed to that. As a result, in the event of breaking changes that occur outside your control, you have only one, ideally small, piece of code to check and fix.

## Coding options of bounded contexts

How would you code a bounded context? Technically, a bounded context is only a module treated in isolation from others. Often, this also means that a bounded context is deployed autonomously. However, the range of options for coding a bounded context is ample and includes in-process options.

The most common scenario—and the most common reason for wanting a bounded context—is to deploy it as a standalone web service accessible via HTTPS and JSON, optionally with a private or shared database. A bounded context, though, can easily be a class library distributed as a plain DLL or, better yet, as a NuGet package. For example, it is almost always a class library when it represents the proxy to an external system.

The public interface of a bounded context with other bounded contexts can be anything that allows for communication: a REST or gRPC gateway, a SignalR or in-process dependency, a shared database, a message bus, or whatever else.



**Note** Does the definition of a bounded context sound like that of a microservice? As you'll see in Chapter 9, there is a resemblance to the definition of *microservice* given by Martin Fowler: a module that runs in its own process and communicates through lightweight mechanisms such as an HTTPS API. In Fowler's vision, a microservice is built around specific business capabilities. The issue is in the intended meaning of the prefix *micro*. Size aside, I like to think of a bounded context as the theoretical foundation of a microservice. The same is true if we consider the alternative architecture of modular monoliths (see Chapter 9). A bounded context is also the theoretical foundation of a module in a monolith. I say "theoretical" for a reason: microservices and modular monoliths live in the space of the software solution, whereas bounded contexts exist in the space of the business domain.

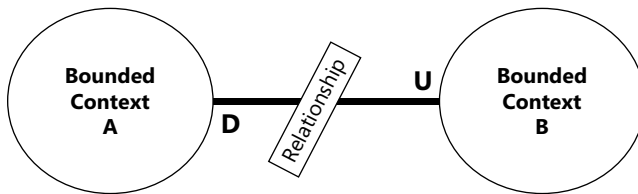
# The context map

The outcome of a DDD analysis of business domain requirements is a collection of bounded contexts that, when combined, form the whole set of functions to implement. How are bounded contexts connected? Interestingly, connection occurs at two distinct levels. One is the physical connection between running host processes. As mentioned, such connections can take the form of HTTPS, SignalR, shared databases, or message buses. But another equally important level of connection is logical and collaborative rather than physical. The following sections explore the types of business relationships supported between bounded contexts.

Bounded contexts and their relationships form a graph that DDD defines as the *context map*. In the map, each bounded context is connected to others with which it is correlated in terms of functionalities. It doesn't have to be a physical connection, though. Often, it looks much more like a logical dependency.

## Upstream and downstream

Each DDD relationship between two bounded contexts is rendered with an arc connecting two nodes of a graph. More precisely, the arc has a directed edge characterized by the letter U (upstream context) or D (downstream context). (See Figure 2-6.)



**FIGURE 2-6** Graphical notation of a context map relationship.

An upstream bounded context influences a downstream bounded context, but the opposite is not true. Such an influence may take various forms. Obviously, the code in the upstream context is available as a reference to the downstream context. It also means, though, that the schedule of work in the upstream context cannot be changed on demand by the team managing the downstream context. Furthermore, the response of the upstream team to requests for change may not be as prompt as desired by the downstream team.

Starting from the notion of upstream and downstream contexts, DDD defines a few specific types of relationships. Essentially, each relationship defines a different type of mutual dependency between involved contexts. These relationships are as follows:

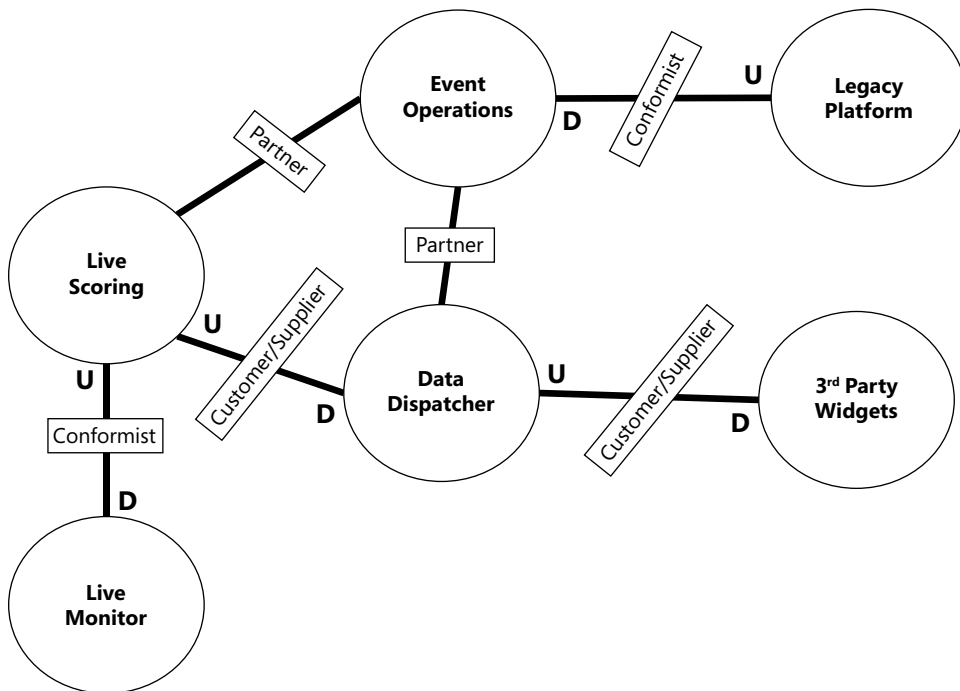
- **Conformist** A conformist relationship indicates that the code in the downstream context is totally dependent on the code in the upstream context. At the end of the day, this means that if a breaking change happens upstream, the downstream context must adapt and conform. By design, the downstream context has no room to negotiate about changes. Typically, a conformist relationship exists when the upstream context is based on some legacy code or is an external

service (for example, a public API) placed outside the control of the development teams. Another possible scenario is when the chief architect sets one context as high priority, meaning that any changes the team plans must be reflected, by design, by all other contexts and teams.

- **Customer/supplier** In this parent–child type of relationship, the downstream customer context depends on the upstream supplier context and must adapt to any changes. Unlike the conformist relationship, though, with the customer/supplier relationship, the two parties are encouraged to negotiate changes that may affect each other. For example, the downstream customer team can share concerns and expect that the upstream supplier team will address them in some way. Ultimately, though, the final word belongs to the upstream supplier context.
- **Partner** The partner relationship is a form of mutual dependency set between the two involved bounded contexts. Put another way, both contexts depend on each other for the actual delivery of the code. This means that no team is allowed to make changes to the public interface of the context without consulting with the other team and reaching a mutual agreement.

## An example context map

Considering this discussion of bounded contexts and relationships, one might reasonably ask how these work in a realistic example. Figure 2-4 showed an example breakdown of a sport-tech data-collection business domain. Figure 2-7 shows a possible set of relationships for that scenario.



**FIGURE 2-7** An example context map for the bounded contexts identified in Figure 2-4.



Let's review the diagram proceeding from left to right:

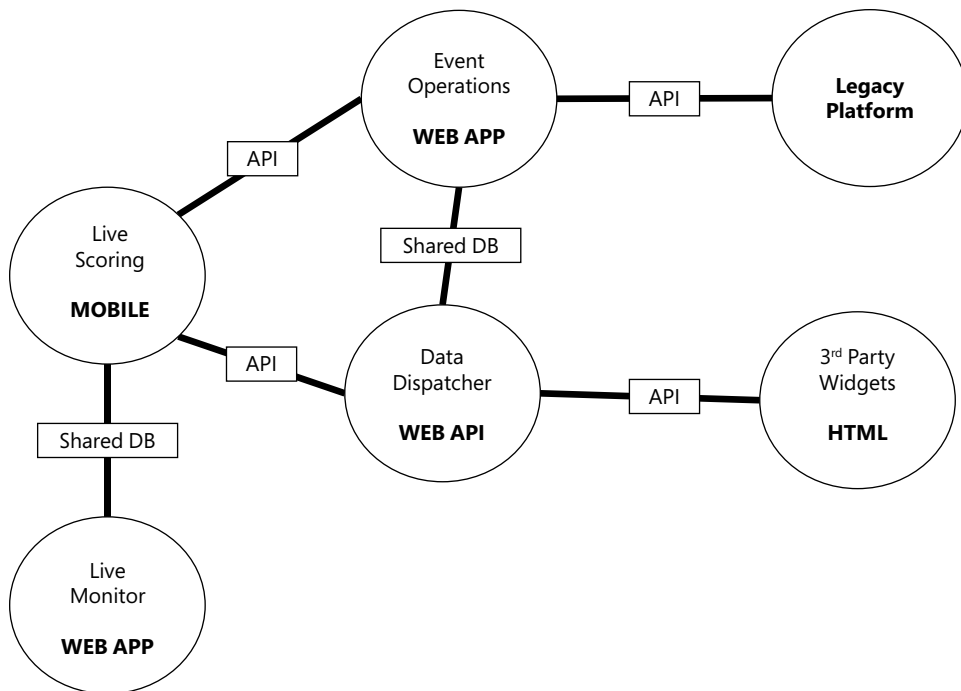
- The Live Scoring context dominates the Data Dispatcher and Live Monitor contexts. So, any changes required for the Live Scoring context must be immediately accepted and reflected by the downstream contexts. This is reasonable, because the Data Dispatcher context is simply expected to route live information to takers and the Live Monitor context just proxies live data for internal scouting and analysis. Indeed, both relationships could be set to conformist, which is even stricter.
- The Live Scoring context partners with the Event Operations context. This is because in the architect's vision, the two modules may influence each other, and changes in one may be as important as changes in the other. A similar production system might have a partner relationship between the Live Scoring and Event Operations contexts, in which case it's often true that one team must conform to changes requested by the other (always for strict business reasons).
- The Event Operations context is totally dependent on the legacy applications connected to the system. This means that live data should be packaged and pushed in exactly the legacy format, with no room for negotiation.
- The Data Dispatcher context and the Event Operations context are partners, as both contexts collect and shape data to be distributed to the outside world, such as to media and IT partners.
- The Third-Party Widgets context contains widgets designed to be embedded in websites. As such, they are subject to conditions set by the Data Dispatcher context. From the perspective of the widget module, the dispatcher is a closed external system.



**Important** The person responsible for setting up the network of relationships is the chief architect. The direction of connections also has an impact on teams, their schedule, and their way of working.

## An example deployment map

The context map is a theoretical map of functions. It says nothing about the actual topology of the deployment environment. In fact, as mentioned, a bounded context may even be a class library coded in an application that turns out to be another bounded context. Often, a bounded context maps to a deployed (web) service, but this is not a general rule. That said, let's imagine a possible deployment map for the context map in Figure 2-7. Figure 2-8 shows a quite realistic high-level deployment scenario for a sport-tech data-collection platform.



**FIGURE 2-8** An example deployment map.

## Summary

This chapter focused on DDD strategic design in a way that is mostly agnostic of software technology and frameworks. The strategic part of DDD is crucial; it involves discovering the top-level architecture of the system using a few analysis patterns and common practices.

The chapter covered the role of the UL, the discovery of distinct bounded contexts, and the relationships the chief architect may use to link contexts together. The map of contexts—the final deliverable of the DDD strategic analysis—is not yet a deployable architecture, but it is key to understanding how to map identified blocks to running services.

All these notions are conceptually valid and describe the real mechanics of DDD. However, it might seem as though they have limited concrete value if measured against relatively simple and small business domains. The actual value of DDD analysis shines when the density of the final map is well beyond the tens of units. Indeed, the largest map I've ever seen (for a pharmaceutical company) contained more than 400 bounded contexts. The screenshot of the map was too dense to count!

The next chapter draws some conclusions about the structure of a .NET and ASP.NET project that maintains clear boundaries between layers. In Part II of the book, we'll delve into each layer.

*This page intentionally left blank*

# Index

## SYMBOLS

{ } (braces), coding style conventions, 161

## A

accessibility

access control

exception details, 120

file access, application layer, 107–108

Project Renoir, 92–94

data access, domain services, 172

rich frameworks, 269

user access control, Project Renoir, 78

ACL (Anti-Corruption Layer), 41

ACM Turing Award, 47

acronyms, UL glossaries, 32

ADO.NET, repository pattern, 194

aggregates

business domain model, 135, 136, 143–144

characteristics of, 143

classes, 144

consistency boundaries, 143

isolation, 143

persistence-driven definitions, Project Renoir, 146–147

relationships, 143

roots, 104

transactional boundaries, 143

Agile

development, microservices, 228

methodologies, technical credit, 294–295

*Agile Manifesto, The*, 18, 75

agility, FDA, 21–22

AJAX (Asynchronous JavaScript and XML), 260

Amazon, 224

ambiguity, DDD architectures, 37–39

anemia, software, 148–149

anemic programming, business domain model, 135

Angular, 261–262, 265–266

annotations, entities, 152

API (Application Programming Interface)

API-only presentations, 88–89

endpoints, 89

Minimal API, 81–82, 273

Web API, 191

web exposure, 88

API layer, 266–269

application endpoints, ASP.NET, 80–82

application layer, 10, 91

application services, blueprint of, 107

application settings, 110

classes, 111

data mergers from various sources, 110–111

hot reloads, 112–113

injecting into application services, 111–112

boundaries of, 129–130

caching, 123

cache-aside patterns, 124

distributed caches, 123–124

location of, 124–125

in-memory caches, 123

organizing data in caches, 126

write-through patterns, 124

cross-cutting, 110

data transfers

from application layer to persistence layer, 104–106

from presentation layer to application layer, 100–103

dependencies, 129

deploying, 129

microservices, 130

separate class libraries, 130

tightly coupled with web applications, 129

exception handling, 119–122

file access, 107–108

logging, 113

application facts, 117–118

ASP.NET Core, 113

ASP.NET loggers (default), 113–114

configuring loggers, 116–117

embedding loggers in base controllers, 118

production-level loggers, 114–116

registering loggers, 113–114

modularization, 51–52

multi-tiered architectures, 6

outline of, 106–110

presentation layer

dependencies, 83–84

mediator connections, 85–86

message bus connections, 86

Project Renoir

access control, 92–94

architectural view, 91–92

## application layer

- data transfers, application layer to persistence layer, 104–106
- data transfers, presentation layer to application layer, 100–103
- document management, 94–95
- fixed user/role association, 93
- flexible user/asset/role association, 93
- permissions, 94–95
- sharing documents, 95
- task orchestration, 96–99
- user authentication, 92
- SignalR connection hubs, 126
  - monitoring hubs, 126–127
  - notifications, sending to client browsers, 128
  - propagating, 127–128
- task orchestration, 96
  - defining tasks, 96–97
  - distributed tasks, 97–99
  - example task, 99
- throwing exceptions, 119–122
- use-case workflows, 108–109
- application services
  - blueprint of, 107
  - domain services versus, 184–185
  - injecting application settings into, 111–112
  - libraries, 72–73
- applications
  - API layer, 266
    - GraphQL API, 266–269
    - REST API, 266–269
  - application layer deployments, 129
  - base classes, entities, 151–152
  - brief history of, 256
  - collections of applications versus microservices, 241
  - defined, 6–7
  - facts, logging, 117–118
  - HTML layer
    - front-end pages, 263
    - rendering HTML, 264–265
    - SSG, 272
    - Svelte, 270–271
    - text templating, 263–264
  - legacy applications
    - dealing with, 220–221
    - origin of, 220
  - microservices, flexibility in all applications, 235
  - rich frameworks
    - accessibility, 269
    - BFF, 270
    - drawbacks of, 269–270
    - performance overhead, 269
    - SEO, 269
    - SSR, 270
  - settings, 110
    - classes, 111
    - data mergers, from various sources, 110–111
    - hot reloads, 112–113
    - injecting into application services, 111–112
  - SPA, 260–261
  - SSG, 271–272
- “archipelago of services,” microservices as, 227
- architectures
  - CA, 18–20
  - client/server architectures, 4
  - CQRS, 12–13
  - data storage, 208
  - DDD architectures, 9, 12, 23
    - ACL, 41
    - ambiguity, 37–39
    - application layer. *See* separate entry
    - author’s experience with, 139–140
    - bounded contexts, 39–41
    - coding rules, relevance of, 27–28
    - coding rules, value of, 28
    - context maps, 42–44
    - database agnosticism, 28
    - deployment maps, 44–45
    - domain layer. *See* separate entry
    - domain models, 27
    - infrastructure layer. *See* separate entry
    - language rules, 28
    - layer interconnections, 12
    - misconceptions of, 27–28
    - persistence ignorance, 141
    - presentation layer, 10
    - programming tools, 35–36
    - proposed supporting architecture, 9–10
    - scale of, 26
    - software models, 27
    - strategic analysis, 24–26
    - strategic design tools (overview), 29
    - supporting architectures (default), 26–27
    - tactical design, 26–27
    - UL, 29–39
  - EDA, 16
  - event sourcing, 14–16
  - FDA, 20
    - agility, 21–22
    - tradeoffs, 21–22
    - VSA, 21
  - HA, 17–18
  - layers, defined, 5
  - microservices implications, 242
  - multi-tiered architectures, 4–5
    - application layer, 6
    - business layer, 8
    - data layer, 8
    - defining applications, 6–7
    - defining layers, 5
    - defining tiers, 5
    - domain layer, 6
    - infrastructure layer, 6
    - presentation layer, 6, 8
    - purpose of, 9
    - SoC, 7, 9
    - software monoliths, 5
    - value of *N*, 6
  - Project Renoir architectural view, 91–92
  - SOA, 224, 225–226
    - microservices and, 237

- tenets of, 224–225
  - software architectures
    - modular monoliths, 248
    - Zen of, 297
  - three-tier architectures, 4–5, 10
    - defining layers, 5
    - defining tiers, 5
    - software monoliths, 5
    - U.S. Prohibition Act, 7
    - value of *N*, 6
  - tiers, defined, 5
  - VSA, 21
  - ASP.NET
    - Blazor, 278
    - HTMX, 277
    - loggers (default), 113–114
    - multithreading, 282–283
    - Project Renoir, application endpoints, 80–82
    - Razor, 275–277
    - SSG, 272
    - Svelte, 276
    - Vanilla JavaScript, 275–276
    - Vue.js framework, 276–277
    - Web Forms, 258–259
    - web stacks, front-end and back-end separation, 274
  - ASP.NET Core
    - DI containers, 180–182
    - distributed caches, 123–124
    - logging, 113
    - in-memory caches, 123
    - microservices, 243–244
    - middleware, 106
    - Minimal API, 273
    - Node.js versus, 278–281
    - Project Renoir
      - application gateways, 80
      - middleware, 79–80
  - assets (Project Renoir), flexible user/asset/role
  - association, 93
  - attribute-based equality, domain value types, 142
  - Atwood, Jeff, 240–241
  - authentication
    - microservices, 231
    - Project Renoir, 92
  - authorization
    - domain services, 187
    - microservices, 231
- ## B
- back-end and front-end separation, 274
    - data, 274–275
    - markups, 274–275
    - single web stacks, 274
  - Barham, Paul, 237
  - base controllers, embedding loggers in, 118
  - batch operations, EF Core, 205
  - Beck, Kent, 133
  - behavioral gaps (business domain model), filling, 137
  - BFF (Back-end for Front-end), 270
  - Blazor
    - ASP.NET, 278
    - server apps, 87
  - blinking at domain events, 174
  - Boolean methods, clean code etiquette, 157–158
  - boundaries, modular monoliths
    - extracting, 250–251
    - logical boundaries, 250
  - bounded contexts, 25–26, 36–37, 38, 69
    - coding options, 41
    - context maps, 42–44
    - deployment maps, 44–45
    - domain model breakdowns, 39
    - external systems, 40–41
    - legacy systems, 40–41
    - shared kernels, 40
  - braces ({}), coding style conventions, 161
  - bridging domains/infrastructures, business domain model, 137–138
  - broken window theory, 293–295
  - browser wars, 257
  - bubbling exceptions, 121–122
  - BUFD (Big Up-Front Design), 246
  - business domain model, 23, 133–134, 138
    - aggregates, 135, 136, 143–144, 146–147
    - anemic programming, 135
    - breakdowns, 24–25
    - bridging domains/infrastructures, 137–138
    - complexity, 147–148
    - cross-cutting, 136
    - data-centric system builds, 138–139
    - decision trees, 147–148
    - domain entities, 141–142
      - application-specific base classes, 151–152
      - common traits of, 149–152
      - data annotations, 152
      - identity, 150
      - loading states, 144–145
      - states, 138
      - top-level base cases, 149–150
    - domain services, states, 138
    - domain value types, 142–143
    - entities, 134, 141–142, 149–152
    - filling behavioral gaps, 137
    - functional programming, 135
    - helper domain services, 137–138
    - internals of, 134–135
    - life forms in, 141–145
    - microservices, 25–26
    - models, defined, 140
    - OOP, 135
    - paradigms, 135
    - persistence ignorance, 141
    - persistence of, 135–136
    - Project Renoir, 145
      - aggregates, 146–147
      - dependency lists, 145
      - life forms in libraries, 146
      - top-level base cases, 149–150
    - roots, 143–144
    - software anemia, 148–149

## business domain model

- subdomains, 25
  - top-level architectures, 24
  - value objects, 134
  - business emails, sending with domain services, 174–175
  - business layer, multi-tiered architectures, 8
  - business logic
    - defined, 91
    - errors, data transfers, 105–106
    - hosting in databases, 207
      - stored procedures, EF Core, 207–208
      - stored procedures, pros/cons, 207
    - message-based business logic, 14–15
  - business rules, handling with strategy pattern, 182–183
  - business terms, Project Renoir, 68
  - business workflow connections, presentation layer, 82
    - application layer dependency, 83–84
    - controller methods, 82–83
    - exceptions to controller rules, 84–85
    - mediator connections, 85–86
  - business-requirements engineering
    - domain models, 76–77
    - event modeling, 76–77
    - event storming, 76
    - event-based storyboards, 76–77
    - software projects, 74–75
      - Agile Manifesto, The*, 75
      - waterfall model, 75
- ## C
- C#, DTO implementations, 100
  - CA (Clean Architectures), 18–20
  - caching, 123
    - cache-aside patterns, 124
    - distributed caches, 123–124
    - location of, 124–125
    - in-memory caches, 123
    - organizing data in caches, 126
    - write-through patterns, 124
  - centralized logging services, microservices, 230–231
  - circuit breakers, 234
  - class libraries, application layer deployments, 130
  - classes
    - aggregates, 144
    - application settings, 111
    - DocumentManagerService class, 177–179
    - domain services
      - DocumentManagerService class, 177–179
      - marking classes, 170–171
    - partial classes, 162
    - repository classes, 193–196
    - WCF reference classes, 191
  - clean code, 20
    - DIP, 167
    - etiquette, 152
      - Boolean methods, 157–158
      - constant values, 159–160
      - Data Clump anti-pattern, 160–161
      - ER principle, 153
      - extension methods, 156–157
    - Extract Method refactoring pattern, 155–156
    - if pollution, 153
    - LINQ, 154–155
    - loop emissions, 154–155
    - naturalizing enums, 158–159
    - pattern matching, 153–154
    - syntactic sugar, 156–157
  - code assistants, 166
  - DDD architectures
    - code duplication, 38–39
    - programming tools, 35–36
    - relevance of coding rules, 27–28
    - value of coding rules, 28
  - DIP, 167
  - ISP, 167
  - LSP, 167
  - Extract Method refactoring pattern, 155–156
  - if pollution, 153
  - LINQ, 154–155
  - loop emissions, 154–155
  - naturalizing enums, 158–159
  - pattern matching, 153–154
  - syntactic sugar, 156–157
  - ISP, 167
  - LSP, 167
  - OCP, 167
  - SOLID acronym, 166–167
  - SRP, 166–167
  - style conventions, 161–165
    - braces ({}), 161
    - comments, 164–165
    - consistent naming conventions, 161
    - indentation, 161
    - line length, 163–164
    - meaningful naming, 161
    - method length, 163–164
    - partial classes, 162
    - spacing, 162
    - Visual Studio regions, 163
  - client browsers, sending SignalR notifications to, 128
  - client-scripting, 260
    - AJAX, 260
    - Angular, 261–262, 265–266
    - CSR, 262
    - JavaScript, modern application frameworks, 261–262
    - React, 261, 262, 265–266
  - client/server architectures, 4
  - Cockburn, Alistair, 18
  - Cockcroft, Adrian, 236–237
  - Codd, Dr. Edgar F.139
  - code assistants, 166
  - code bases, modular monoliths, 248
  - code-behind classes, Razor pages, 81
  - coding
    - bounded contexts, 41
    - clean code etiquette, 20, 152
      - Boolean methods, 157–158
      - constant values, 159–160
      - Data Clump anti-pattern, 160–161
      - ER principle, 153
      - extension methods, 156–157
      - Extract Method refactoring pattern, 155–156
      - if pollution, 153
      - LINQ, 154–155
      - loop emissions, 154–155
      - naturalizing enums, 158–159
      - pattern matching, 153–154
      - syntactic sugar, 156–157
    - code assistants, 166
    - DDD architectures
      - code duplication, 38–39
      - programming tools, 35–36
      - relevance of coding rules, 27–28
      - value of coding rules, 28
    - DIP, 167
    - ISP, 167
    - LSP, 167

- monoliths, code development/maintenance, 221–222
  - OCP, 167
  - readable code
    - author's experience with, 165–166
    - writing, 165–166
  - reusability, 196
  - SOLID acronym, 166–167
  - SRP, 166–167
  - style conventions, 161–165
    - braces ({}), 161
    - comments, 164–165
    - consistent naming conventions, 161
    - indentation, 161
    - line length, 163–164
    - meaningful naming, 161
    - method length, 163–164
    - partial classes, 162
    - spacing, 162
    - Visual Studio regions, 163
  - UL, impact on coding, 33–34
  - Zen of, 297–298
  - commands
    - CQRS, 12–13
      - architectural perspective, 209–210
      - business perspective, 210–211
      - distinct databases, 211-
      - shared databases, 211
    - query separation, 208–213
  - comments, style conventions, 164–165
  - compiled queries, EF Core, 204–205
  - complexity, business domain model, 147–148
  - Concerns (SoC), Separation of, 7, 9, 20, 47, 48–49
  - configuring
    - loggers, 116–117
    - monitoring hubs, 126–127
  - conformists, context maps, 42–43
  - consistency
    - boundaries, aggregates, 143
    - domain entities, 142
    - domain value types, 142
    - microservices, 232
    - naming conventions, 161
  - constant values, clean code etiquette, 159–160
  - context maps, 42
    - application services libraries, 72–73
    - bounded contexts, 69
    - conformists, 42–43
    - customers/suppliers, 43
    - domain models, 69–71, 73
    - downstream context, 42–43
    - EF, 70
    - example of, 43–44
    - front-end application project, 71
    - helper libraries, 74
    - infrastructure libraries, 73
    - O/RM tool, 70
    - partners, 43
    - persistence libraries, 74
    - Project Renoir
      - abstract context maps, 68–71
      - physical context maps, 71–74
      - UL, 68–69
      - upstream context, 42–43
  - controller methods, 82–83
  - controllers (base), embedding loggers in, 118
  - costs, 239
    - microservices, 239–240
    - technical debt, 285–286
  - coupling, loose, 49
  - CQRS (Command/Query Responsibility Segregation), 12–13
    - architectural perspective, 209–210
    - business perspective, 210–211
    - distinct databases, 211-
    - shared databases, 211
  - Craver, Nick, 241
  - credit, technical, 285
    - Agile methodologies, 294–295
    - broken window theory, 293–295
    - design principles, 294
    - profit of, 293
    - refactoring, 293
    - testability, 295
  - cross-cutting
    - application layer, 110
    - business domain model, 136
    - microservices, 230
  - CSR (Client-Side Rendering), 262
  - CSS (Cascading Style Sheets), 256
  - customers
    - context maps, 43
    - loyalty status, determining with domain services, 173
- ## D
- Dapper
    - defined, 205
    - internal operation, 206
    - operating, 206
  - data access, domain services, 172
  - data annotations, entities, 152
  - Data Clump anti-pattern, 160–161
  - data injection, domain services, 172
  - data layer
    - modularization, 52
    - multi-tiered architectures, 8
  - data management, microservices, 233–234
  - data mergers, from various sources, 110–111
  - data normalization, domain services, 187
  - data organization in caches, 126
  - data persistence, infrastructure layer, 190
  - data separation, front-end and back-end, 274–275
  - data storage
    - architectures, 208
    - infrastructure layer, 190
  - data transfers
    - from application layer to persistence layer, 104–106
    - business logic errors, 105–106
    - disconnecting from HTTP context, 100–101
    - domain entities, 103



## data transfers

- DTO
  - C# implementations, 100
  - defined, 99
  - input view model, 101–102
  - from presentation layer to application layer, 100–103
  - repositories, 104
  - response view model, 102
- databases
  - agnosticism, DDD architectures, 28
  - business logic, hosting in databases, 207
    - stored procedures, EF Core, 207–208
    - stored procedures, pros/cons, 207
  - CQRS
    - distinct databases, 211–
      - shared databases, 211
  - EF Core connections, 197–198
  - microservice database patterns, 233
  - shared databases
    - CQRS, 211
    - infrastructure layer, 191
- data-centric system builds, business domain model, 138–139
- DbContext object pooling, 204
- DDD architectures, 9, 12, 23
  - ACL, 41
  - ambiguity, 37–39
  - application layer. *See separate entry*
  - author's experience with, 139–140
  - bounded contexts, 36–37, 38, 69
    - coding options, 41
    - context maps, 42–44
    - deployment maps, 44–45
    - domain model breakdowns, 39
    - external systems, 40–41
    - legacy systems, 40–41
    - shared kernels, 40
  - business-requirements engineering, 74–75
    - Agile Manifesto, The*, 75
    - domain models, 76–77
    - event modeling, 76–77
    - event storming, 76
    - event-based storyboards, 76–77
    - waterfall model, 75
  - code duplication, 38–39
  - coding rules
    - relevance of, 27–28
    - value of, 28
  - context maps
    - abstract context maps, 68–71
    - application services libraries, 72–73
    - bounded contexts, 69
    - domain model libraries, 73
    - domain models, 69–71
    - EF, 70
    - front-end application project, 71
    - helper libraries, 74
    - infrastructure libraries, 73
    - O/RM tool, 70
    - persistence libraries, 74
    - physical context maps, 71–74
    - UL, 68–69
  - database agnosticism, 28
  - domain layer. *See separate entry*
  - domain models, 27
  - infrastructure layer. *See separate entry*
  - language rules, 28
  - layer interconnections, 12
  - misconceptions of, 27–28
  - persistence ignorance, 141
  - presentation layer. *See separate entry*
  - programming tools, 35–36
  - proposed supporting architecture, 9–10
  - scale of, 26
  - software models, 27
  - strategic analysis, 24
    - bounded contexts, 25–26
    - business domains, breakdowns, 24–25
    - business domains, microservices, 25–26
    - business domains, subdomains, 25
    - business domains, top-level architectures, 24
  - strategic design tools (overview), 29
  - supporting architectures (default), 26–27
  - tactical design, 26–27
  - UL, 29, 68–69
    - acronyms in glossaries, 32
    - building glossaries, 31–32
    - changes to languages, 33, 34–35
    - choosing natural language of glossaries, 30
    - factory methods, 36
    - goal of, 33
    - impact on coding, 33–34
    - shared glossaries of terms, 30
    - sharing glossaries, 32–33
    - technical terms in glossaries, 32
    - value types, 36
- debt, technical
  - amplifiers, 292
    - lack of documentation, 290–291
    - lack of skills, 292
    - rapid prototyping, 291–292
    - scope creep, 291
  - costs, 285–286
  - defined, 285
  - genesis of, 296–297
  - impact of, 289
  - pragmatic perspective, 286–287
  - quality bar, raising, 298
  - reasons for creating debt, 287
  - reducing
    - inventory of debt items, 289
    - separating projects, 288–289
    - ship-and-remove strategy, 289
    - slack time, 290
    - spikes, 290
    - timeboxing, 290
  - refactoring, 295
  - signs of, 287–288
  - Zen of coding, 297–298
  - Zen of software infrastructure, 297
- debugging, modular monoliths, 248
- decision trees, business domain model, 147–148
- dependencies

- application layer, 129
- decoupling in monoliths, 53
- injections,
  - domain services, 180–182
- lists, Project Renoir, business domain model, 145
- managing, modularization, 50
- on other functions, domain services, 177
- deploying
  - application layer, 129
  - microservices, 130
  - separate class libraries, 130
  - tightly coupled with web applications, 129
  - microservices, 226–227, 243–244
  - monoliths, 223
  - presentation layer, 79
- deployment maps, 44–45
- design patterns, testability, 59–60
- design principles
  - DRY, 294
  - KISS, 248, 294
  - SOLID, 166–167, 294
  - technical credit, 294
  - YAGNI, 294
- Deutsch, Peter, 247
- development costs, microservices, 239
- development velocity, modular monoliths, 249
- DI containers, 180–182
- dictionary of business terms, Project Renoir, 68
- Dijkstra, Edsger W.7, 20, 91
- DIP (Dependency Inversion Principle), 167
- disabling object tracking with EF Core, 204
- disconnecting from HTTP context, data transfers, 100–101
- distributed caches, 123–124
- distributed tasks, example of, 97–99
- distributed transactions, microservices, 232
- diversity, technology
  - microservices, 55–56
  - monoliths, 223
- documentation
  - lack of, debt amplifiers, 290–291
  - modularization, 50
  - Project Renoir, document management, 94–95
  - sharing documents, 95
- DocumentManagerService class, domain services, 177–179
- DOM (Document Object Models), 257, 265–266
- Domain-Driven Design: Tackling Complexity at the Heart of Software* (2003), 27, 139, 170
- domain layer, 11, 133
  - business domain model, 133–134, 138
    - aggregates, 135, 136
    - anemic programming, 135
    - application-specific base classes, 151–152
    - bridging domains/infrastructures, 137–138
    - complexity, 147–148
    - cross-cutting, 136
    - data annotations, 152
    - data-centric system builds, 138–139
    - decision trees, 147–148
    - domain entities, 141–142

## domains/infrastructures (business domain model)

- domain entities, common traits of, 149–152
- domain entities, identity, 150
- domain entities, states, 138
- domain services, states, 138
- domain value types, 142–143
- entities, 134, 141–142, 149–152
- filling behavioral gaps, 137
- functional programming, 135
- helper domain services, 137–138
- internals of, 134–135
- life forms in, 141–145
- models, defined, 140
- OOP, 135
- paradigms, 135
- persistence ignorance, 141
- persistence of, 135–136
- Project Renoir, 145–147
- software anemia, 148–149
- value objects, 134
- decomposition of, 133
- modularization, 52
- multi-tiered architectures, 6
- in perspective, 134
- domains/infrastructures (business domain model)
  - bridging, 137–138
  - business domains, 23
  - business emails, sending, 174–175
  - costs, microservices, 239
  - DI containers, 180–182
  - DocumentManagerService class, 177–179
  - domain validation, 187
  - entities, 52
    - application-specific base classes, 151–152
    - business domain model, 141–142
    - common traits of, 149–152
    - consistency, 142
    - data annotations, 152
    - data transfers, 103
    - identity, 141
    - life cycles, 142
    - loading states, 144–145
    - mutability, 141–142
    - states, 138
    - top-level base cases, 149–150
  - events, blinking at, 174
  - If...Then...Throw pattern, 179–180
  - implementing, 176
  - impure/pure domain services, 185–186
  - interfaces, creating, 177
  - legacy system integration, 187
  - libraries, 73
  - marking classes, 170–171
  - models, 11, 27, 76–77
    - breakdowns, bounded contexts, 39
    - context maps, 69–71
    - libraries, 73
    - persistence models versus, 105, 201–203
  - necessity of, 184–186
  - open points, 184
  - pure/impure domain services, 185–186
  - readiness, microservices, 238

## domains/infrastructures (business domain model)

- repositories
    - domains versus, 193
    - expanding scope of, 186
  - REPR pattern, 180
  - security, 187
  - service to hash passwords, 175–176
  - services, 11, 52, 169
    - application services versus, 184–185
    - authorization, 187
    - blinking at domain events, 174
    - building, 176–179
    - common scenarios, 173–177, 187
    - customer loyalty status, determining, 173
    - data access, 172
    - data injection, 172
    - data normalization, 187
    - defined, 170
    - dependencies, injections, 180–182
    - dependencies, on other functions, 177
  - special case pattern, 183–184
  - stateless nature of, 170
  - states, 138
  - strategy pattern, 182–183
  - UL, 171
  - validation, 187
  - value types
    - attribute-based equality, 142
    - business domain model, 142–143
    - consistency, 142
    - immutability, 142
    - invariants, 142
    - life cycles, 142
    - no life cycle, 142
    - primitive types, 143
  - DotNetCore Show, 241
  - downstream context, context maps, 42–43
  - driven ports, HA, 18
  - driver ports, HA, 18
  - DRY (Don't Repeat Yourself), 294
  - DTO (Data Transfer Objects)
    - C# implementations, 100
    - defined, 99
  - dumb pipes, 227
  - duplicating code, DDD architectures, 38–39
- ## E
- eager loading, EF Core, 204
  - EDA (Event-Driven Architectures), 16
  - EF (Entity Frameworks), context maps, 70
  - EF Core, 196–197
    - batch operations, 205
    - compiled queries, 204–205
    - database connections, 197–198
    - DbContext object pooling, 204
    - eager loading, 204
    - object tracking, disabling, 204
    - pagination, 205
    - persistence models, building, 199–201
    - repository pattern, 194
    - stored procedures, 207–208
    - unavoidable practices, 204–205
  - “Eight Fallacies of Distributed Computing”<sup>247</sup>
  - emails (business), sending with domain services, 174–175
  - embedding loggers in base controllers, 118
  - endpoints
    - API, 89
    - smart endpoints, 227
  - Enigma* (1995), 30
  - entities
    - application-specific base classes, 151–152
    - business domain model, 134, 141–142
    - common traits of, 149–152
    - data annotations, 152
    - definitions, event modeling, 77
    - identity, 150
    - top-level base cases, 149–150
  - enums, naturalizing, clean code etiquette, 158–159
  - equality (attribute-based), domain value types, 142
  - ER principle, 153
  - ES (Event Sourcing), 14–16
    - architectural implications, 214–215
    - characterizing traits, 213–214
    - executive summary, 213–215
  - Esposito, Francesco, 72
  - etiquette, clean code, 152
    - Boolean methods, 157–158
    - constant values, 159–160
    - Data Clump anti-pattern, 160–161
    - ER principle, 153
    - extension methods, 156–157
    - Extract Method refactoring pattern, 155–156
    - if pollution, 153
    - LINQ, 154–155
    - loop emissions, 154–155
    - naturalizing enums, 158–159
    - pattern matching, 153–154
    - syntactic sugar, 156–157
  - Evans, Eric, 27, 139, 170
  - event modeling, 76–77
  - event storming, 76
  - event-based storyboards, 76–77
  - exception handling/throwing, 119
    - accessing exception details, 120
    - bubbling exceptions, 121–122
    - custom exception classes, 120–121
    - middleware, 119
    - reformulating exceptions, 121–122
    - swallowing exceptions, 121–122
  - exchanging data, repositories, 104
  - extension methods, clean code etiquette, 156–157
  - external services, communication via infrastructure layer, 190–191
  - external systems, bounded contexts, 40–41
  - Extract Method refactoring pattern, 155–156
- ## F
- Facebook, 296–297
  - factory methods, DDD architectures, 36
  - fault tolerance, microservices, 229

## infrastructures/domains (business domain model)

- FDA (Feature-Driven Architectures), 20
  - agility, 21–22
  - tradeoffs, 21–22
  - VSA, 21
- Fellowship of the Ring, The*, 255
- Fermi, Enrico, 169
- files
  - access, application layer, 107–108
  - shared files/databases
    - CQRS, 211
    - infrastructure layer, 191
- filling behavioral gaps, business domain model, 137
- fixed user/role association, Project Renoir, 93
- flexible user/asset/role association, Project Renoir, 93
- Fowler, Martin, 27
- frameworks, rich, 269–270
  - accessibility, 269
  - BFF, 270
  - performance overhead, 269
  - SEO, 269
  - SSR, 270
- front-end and back-end separation, 274
  - data, 274–275
  - markups, 274–275
  - single web stacks, 274
- front-end application project, context maps, 71
- front-end options, ASP.NET
  - Blazor, 278
  - HTMX, 277
  - Razor, 275–277
  - Svelte, 276
  - Vanilla JavaScript, 275–276
  - Vue.js framework, 276–277
- front-end pages, HTML layer, 263
- functional programming, business domain model, 135
- functions maps, Project Renoir, 77–78

## G

- glossaries, UL
  - acronyms, 32
  - building glossaries, 31–32
  - choosing natural language of glossaries, 30
  - shared glossaries of terms, 30
  - sharing glossaries, 32–33
  - technical terms, 32
- Gmail, 260–261
- Gödel, Kurt, 3
- Google, 237
- Gosling, James, 247
- GPT (Generative Pre-trained Transformer), 72
- GraphQL API, 266–269
- greenfield projects, microservices, 246–247
- gRPC (gRPC Remote Procedure Calls), 235

## H

- HA (Hexagonal Architectures), 17–18
- handling exceptions, 119–122

- Harris, Robert, 30
- hash passwords, service to, 175–176
- helper domain services, 137–138
- helper libraries, 74
- Hoare, Sir Tony, 47
- horizontal scalability, monoliths and modularization, 54
- hosting business logic in databases, stored procedures, 207
  - EF Core, 207–208
  - pros/cons, 207
- hot reloads, application settings, 112–113
- HTML layer
  - front-end pages, 263
  - rendering HTML, 264–265
  - SSG, 272
  - Svelte, 270–271
  - text templating, 263–264
- HTMX, 277
- HTTP context (data transfers), disconnecting from, 100
- hubs (monitoring), configuring, 126–127
- human resources costs, microservices, 239

## I

- IBM 360 system, 4
- IBM San Jose Research Laboratory, 139
- identity
  - domain entities, 141
  - entities, 150
- if pollution, 153
- If...Then...Throw pattern, domain services, 179–180
- IIS middleware, Project Renoir, 79
- immutability, domain value types, 142
- impure but persistent domain models, 203
- impure/pure domain services, 185–186
- Incompleteness, Theorem of, 3
- increasing technical debt, 292
  - lack of documentation, 290–291
  - lack of skills, 292
  - rapid prototyping, 291–292
  - scope creep, 291
- indentation, coding style conventions, 161
- infrastructure layer, 11, 189. *See also* persistence layer
  - data persistence, 190
  - data storage, 190
  - external services communication, 190–191
  - internal services communication, 191–192
  - modularization, 52
  - multi-tiered architectures, 6
  - responsibilities of, 190
  - shared files/databases, 191
  - WCF reference classes, 191
  - Web API, 191
- infrastructures/domains (business domain model)
  - bridging, 137–138
  - business domains, 23
  - business emails, sending, 174–175
  - costs, microservices, 239
  - DI containers, 180–182
  - DocumentManagerService class, 177–179

## infrastructures/domains (business domain model)

- domain validation, 187
  - entities, 52
    - application-specific base classes, 151–152
    - business domain model, 141–142
    - common traits of, 149–152
    - consistency, 142
    - data annotations, 152
    - data transfers, 103
    - identity, 141
    - life cycles, 142
    - loading states, 144–145
    - mutability, 141–142
    - states, 138
    - top-level base cases, 149–150
  - events, blinking at, 174
  - If...Then...Throw pattern, 179–180
  - implementing, 176
  - impure/pure domain services, 185–186
  - interfaces, creating, 177
  - legacy system integration, 187
  - libraries, 73
  - marking classes, 170–171
  - models, 11, 27, 76–77
    - breakdowns, bounded contexts, 39
    - context maps, 69–71
    - libraries, 73
    - persistence models versus, 105, 201–203
  - necessity of, 184–186
  - open points, 184
  - pure/impure domain services, 185–186
  - readiness, microservices, 238
  - repositories
    - domains versus, 193
    - expanding scope of, 186
  - REPR pattern, 180
  - security, 187
  - service to hash passwords, 175–176
  - services, 11, 52, 169
    - application services versus, 184–185
    - authorization, 187
    - blinking at domain events, 174
    - building, 176–179
    - common scenarios, 173–177, 187
    - customer loyalty status, determining, 173
    - data access, 172
    - data injection, 172
    - data normalization, 187
    - defined, 170
    - dependencies, injections, 180–182
    - dependencies, on other functions, 177
  - special case pattern, 183–184
  - stateless nature of, 170
  - states, 138
  - strategy pattern, 182–183
  - UL, 171
  - validation, 187
  - value types
    - attribute-based equality, 142
    - business domain model, 142–143
    - consistency, 142
    - immutability, 142
    - invariants, 142
    - life cycles, 142
    - no life cycle, 142
    - primitive types, 143
  - injecting
    - data, domain services, 172
    - dependencies, , 180–182
  - in-memory caches, 123
  - input view model, data transfers, 101–102
  - interfaces, domain services, 177
  - internal services, infrastructure layer communication, 191–192
  - invariants, domain value types, 142
  - inventories of debt items, reducing technical debt, 289
  - I/O bound tasks, processing in Node.js, 281–282
  - IoC (Inversion of Control),
    - isolation, aggregates, 143
  - ISP (Interface Segregation Principle), 167
- ## J
- JavaScript, 256–257
    - Angular, 261–262, 265–266
    - HTMX, 277
    - modern application frameworks, 261–262
    - Node.js, 273
      - ASP.NET Core versus, 278–281
      - processing I/O bound tasks, 281–282
      - processing requests, 281
    - React, 261, 262, 265–266
    - Vanilla JavaScript, 272–273, 275–276
    - Vue.js framework, 272, 276–277
  - jobs, defined, 4
- ## K
- Kerouac, Jack, 246
  - KISS principle, 248, 294
  - Kubernetes, microservice deployments, 243–244
- ## L
- lack of documentation, debt amplifiers, 290–291
  - lack of skills, debt amplifiers, 292
  - language rules, DDD architectures, 28
    - changes to languages, 33
    - glossaries, 30
  - layers
    - defined, 5
    - interconnections, DDD architectures, 12
  - legacy applications, 220
    - dealing with, 220–221
    - origin of, 220
  - legacy systems
    - bounded contexts, 40–41
    - domain service integration, 187
  - Legoization, 7

- libraries
    - application services libraries, 72–73
    - class libraries, application layer deployments, 130
    - domain model libraries, 73
    - helper libraries, 74
    - infrastructure libraries, 73
    - MediatR mediator library, 85–86
    - persistence libraries, 74
  - life cycles
    - domain entities, 142
    - domain value types, 142
  - life forms, Project Renoir, business domain model, 146
  - line length, style conventions, 163–164
  - LINQ (Language Integrated Queries), 154–155
  - Liskov, Barbara, 167
  - LLM (Large Language Models), 72
  - loading, eager, 204
  - logging
    - application facts, 117–118
    - application layer, 113
    - ASP.NET loggers (default), 113–114
    - centralized logging services, microservices, 230–231
    - configuring loggers, 116–117
    - embedding loggers in base controllers, 118
    - production-level loggers, 114–116
    - registering loggers, 113–114
  - logical boundaries, modular monoliths, 250
  - logical modules, monoliths and modularization, 53
  - loop emissions, clean code etiquette, 154–155
  - loose coupling
    - modular monoliths, 249
    - modularization, 49
  - loyalty status of customers, determining with domain services, 173
  - LSP (Liskov’s Substitution Principle), 167
- ## M
- maintainability, 57
    - readability, 57
    - reusability, 57–58
    - scalability, 58
  - maintenance
    - coding in monoliths, 221–222
    - microservices costs, 239
  - managing data, microservices, 233–234
  - marking classes, domain services, 170–171
  - markups, front-end and back-end separation, 274–275
  - Martin, Robert, 18, 167
  - meaningful naming, coding style conventions, 161
  - mediator libraries, 85–86
  - MediatR mediator library, 85–86
  - memory, in-memory caches, 123
  - merge conflicts, monoliths, 222
  - merging data from various sources, 110–111
  - message-based business logic, 14–15
  - message buses, application layer/presentation layer connections, 86
  - method length, style conventions, 163–164
  - micro-O/RM, repository pattern, 194–195
  - microservices, 219
    - Agile development, 228
    - application layer deployments, 130
    - “archipelago of services” 227
    - architectural implications, 242
    - aspects of (overview), 54–55
    - authentication, 231
    - authorization, 231
    - benefits of, 227–229
    - business domains, 25–26
    - centralized logging services, 230–231
    - challenges of, 56
    - circuit breakers, 234
    - collected applications versus, 241
    - consistency, 232
    - costs, 239–240
    - cross-cutting, 230
    - data management, 233–234
    - database patterns, 233
    - deploying, 226–227, 243–244
    - determining necessary number of, 242
    - determining need for (scenarios), 237–241
    - distributed transactions, 232
    - dumb pipes, 227
    - early adopters, 224
    - fault tolerance, 229
    - as first choice, 247
    - flexibility in all applications, 235
    - gray areas, 229–235
    - greenfield projects, 246–247
    - infrastructure readiness, 238
    - intricacy of, 247
    - logical decomposition of systems, 225–226, 242
    - misconceptions of, 235–236
    - modular monoliths, transitioning to microservices, 249–252
    - modularization, 49
    - Netflix, 236–237
    - Nuvolaris, 245
    - operational overhead, 234–235
    - planning, 241–242
    - scalability, 228, 237–238
    - serverless environments, 244–245
    - service coordination, 229–230
    - size of, 225–227
    - smart endpoints, 227
    - SOA, 224, 237
    - Stack Overflow, 240–241
    - technology diversity, 55–56, 239
    - tenets of, 224–225
  - middleware
    - ASP.NET Core, 106
    - exception handling, 119
  - Minimal API, 81–82, 273
  - models
    - binding, 280
    - defined, 140
  - modularization, 47, 52
    - application layer, 51–52
    - applying (overview), 51
    - aspects of (overview), 48

## modularization

- client/server architectures, 4
  - data layer, 52
  - dependency management, 50
  - development of, 3
  - documentation, 50
  - domain layer, 52
  - infrastructure layer, 52
  - levels of, 47–48
  - loose coupling, 49
  - microservices, 49
    - aspects of (overview), 54–55
    - challenges of, 56
    - technology diversity, 55–56
  - modular monoliths, 66, 245–246
    - applications, 52–53, 66, 220–221
    - boundaries, 250–251
    - BUFD, 246
    - code bases, 248
    - code development/maintenance, 221–222
    - debugging, 248
    - decomposition of, 251
    - decoupling dependencies, 53
    - deployments, 223
    - development velocity, 249
    - features, 252
    - KISS principle, 248
    - legacy applications, 220–221
    - logical modules, 53
    - loose coupling, 249
    - merge conflicts, 222
    - new project strategies, 247–250
    - performance, 249
    - potential downsides of, 221–223
    - scalability, 54, 222
    - session states, 249
  - shared states, 249
  - software architectures, 248
  - sticky sessions, 249
  - testing, 248
  - traits of, 248–249
  - transitioning to microservices, 249–252
  - presentation layer, 51
  - principles of (overview), 48
  - quest for, 3
  - reusability, 49
  - SoC, 7, 9, 20, 47, 48–49
  - SSE, 56–57
  - testability, 50
  - monitoring hubs, configuring, 126–127
  - monoliths,
    - modular monoliths, 66, 245–246
      - applications, 52–53, 66, 220–221
      - boundaries, 250–251
      - BUFD, 246
      - code bases, 248
      - code development/maintenance, 221–222
      - debugging, 248
      - decomposition of, 251
      - decoupling dependencies, 53
      - deployments, 223
      - development velocity, 249
      - features, 252
      - KISS principle, 248
      - legacy applications, 220–221
      - logical modules, 53
      - loose coupling, 249
      - merge conflicts, 222
      - new project strategies, 247–250
      - performance, 249
      - potential downsides of, 221–223
      - scalability, 54, 222
      - session states, 249
    - shared states, 249
    - software architectures, 248
    - sticky sessions, 249
    - testing, 248
    - traits of, 248–249
    - transitioning to microservices, 249–252
    - presentation layer, 51
    - principles of (overview), 48
    - quest for, 3
    - reusability, 49
    - SoC, 7, 9, 20, 47, 48–49
    - SSE, 56–57
    - testability, 50
  - multithreading, ASP.NET, 282–283
  - multi-tiered architectures, 4–5
    - application layer, 6
    - applications, defined, 6–7
    - business layer, 8
    - data layer, 8
    - defining
      - layers, 5
      - tiers, 5
    - domain layer, 6
    - infrastructure layer, 6
    - presentation layer, 6, 8
    - purpose of, 9
    - SoC, 7, 9
    - software monoliths, 5
    - value of *N*, 6
  - mutability, domain entities, 141–142
  - MVC (Model View Controller)
    - methods, Project Renoir, 80–81
    - patterns, 259–260
- ## N
- N*, value of, 6
  - naming conventions
    - consistent naming conventions, 161
    - meaningful naming, 161
  - naturalizing enums, clean code etiquette, 158–159
  - Netflix, 224, 236–237
  - Newton, Sir Isaac, 189
  - Nietzsche, Friedrich, 219
  - no life cycle, domain value types, 142
  - Node.js, 273
    - ASP.NET Core versus, 278–281
    - I/O bound tasks, processing, 281–282
    - requests, processing, 281
  - normalizing data, domain services, 187
  - notifications (SignalR), sending to client browsers, 128
  - NuGet, Project Renoir, 73
  - Nuvolaris, 245

## O

object tracking, disabling with EF Core, 204  
 OCP (Open/Closed Principle), 167  
*On The Road*, 246  
 “On the Role of Scientific Thought”  
 OOP (Object-Oriented Programming), 135  
 open points, domain services, 184  
 operational costs, microservices, 239  
 operational overhead, microservices, 234–235  
 organizing data in caches, 126  
 O/RM tool, context maps, 70

## P

pagination, EF Core, 205  
 partial classes, 162  
 partners, context maps, 43  
 passwords, service to hash, 175–176  
 pattern matching, clean code etiquette, 153–154  
 performance
 

- modular monoliths, 249
- overhead, rich frameworks, 269

 permissions, Project Renoir, 94, 95  
 persistence ignorance, 141  
 persistence layer. *See also* infrastructure layer  
 Dapper
 

- defined, 205
- internal operation, 206
- operating, 206

 data transfers, from application layer to persistence layer, 104–106  
 domain models
 

- impure but persistent domain models, 203
- persistence models versus, 201–203

 EF Core, 196–197
 

- batch operations, 205
- building persistence models, 199–201
- compiled queries, 204–205
- database connections, 197–198
- DbContext object pooling, 204
- disabling object tracking, 204
- eager loading, 204
- pagination, 205
- stored procedures, 207–208
- unavoidable practices, 204–205

 implementing, 192  
 impure but persistent domain models, 203  
 persistence models
 

- building EF Core-specific models, 199–201
- domain models versus, 201–203

 repository classes, 193–196  
 repository pattern, 193–196  
 UoW pattern, 196  
 persistence libraries, 74  
 persistence models, domain models versus, 105, 201–203  
 persistence of data, infrastructure layer, 190  
 personas, Project Renoir, 69  
 Picasso, Pablo, 65

pipes, dumb, 227  
 planning microservices, 241–242  
 Polly, transitioning modular monoliths to microservices, 251  
 ports, driver (HA), 18  
 presentation layer, 65
 

- API-only presentations, 88–89
- application layer
  - dependencies, 83–84
  - mediator connections, 85–86
  - message bus connections, 86
- Blazor server apps, 87
- boundaries of, 79
- business workflow connections, 82
  - application layer dependency, 83–84
  - controller methods, 82–83
  - exceptions to controller rules, 84–85
  - mediator connections, 85–86
- business-requirements engineering, 74–75
  - Agile Manifesto, The*, 75
  - domain models, 76–77
  - event modeling, 76–77
  - event storming, 76
  - event-based storyboards, 76–77
  - waterfall model, 75
- context maps
  - abstract context maps, 68–71
  - application services libraries, 72–73
  - bounded contexts, 69
  - domain model libraries, 73
  - domain models, 69–71
  - EF, 70
  - front-end application project, 71
  - helper libraries, 74
  - infrastructure libraries, 73
  - O/RM tool, 70
  - persistence libraries, 74
  - physical context maps, 71–74
  - UL, 68–69
- DDD architectures, 10
- deploying, 79
- development (overview), 82
- modularization, 51
- multi-tiered architectures, 6, 8
- Project Renoir, 66
  - ASP.NET, application endpoints, 80–82
  - ASP.NET Core, application gateways, 80
  - ASP.NET Core, middleware, 79–80
  - business terms, 68
  - business-requirements engineering, 74–77
  - context maps, abstract, 68–71
  - context maps, physical, 71–74
  - functions maps, 77–78
  - fundamental tasks, 77–79
  - IIS middleware, 79
  - introduction to (overview), 66
  - Minimal API endpoints, 81–82
  - MVC methods, 80–81
  - personas, 69
  - product-related features, 78–79



## presentation layer

- Razor page code-behind classes, 81
  - release notes, 66–67
  - release notes, creation tools, 66–67
  - release notes, writing, 67
  - user access control, 78
  - SSR, 86–87
  - Wisej, 87–88
  - primitive types, domain value types, 143
  - procedures, stored
    - EF Core, 207–208
    - pros/cons, 207
  - production-level loggers, 114–116
  - product-related features, Project Renoir, 78–79
  - Programming Large Language Models with Azure OpenAI* (2024), 72
  - programming tools, DDD architectures, 35–36
  - Prohibition Act, U.S.7
  - Project Renoir, 66
    - access control, 92–94
    - application layer
      - application services, 107
      - application settings, 110–113
      - boundaries of, 129–130
      - caching, 123–126
      - cross-cutting, 110
      - deploying, 129–130
      - file access, 107–108
      - handling exceptions, 119–122
      - logging, 113–118
      - outline of, 106–110
      - SignalR connection hubs, 126–128
      - throwing exceptions, 119–122
      - use-case workflows, 108–109
    - architectural view, 91–92
  - ASP.NET, application endpoints, 80–82
  - ASP.NET Core
    - application gateways, 80
    - middleware, 79–80
  - authentication, 92
  - business domain model, 145
    - aggregates, 146–147
    - dependency lists, 145
    - life forms in libraries, 146
    - top-level base cases, 149–150
  - business-requirements engineering, 74–75
    - Agile Manifesto, The*, 75
    - domain models, 76–77
    - event modeling, 76–77
    - event storming, 76
    - event-based storyboards, 76–77
    - waterfall model, 75
    - business terms, 68
  - context maps
    - abstract context maps, 68–71
    - application services libraries, 72–73
    - bounded contexts, 69
    - domain model libraries, 73
    - domain models, 69–71
    - EF, 70
    - front-end application project, 71
    - helper libraries, 74
    - infrastructure libraries, 73
    - O/RM tool, 70
    - persistence libraries, 74
    - physical context maps, 71–74
    - UL, 68–69
  - data transfers
    - from application layer to persistence layer, 104–106
    - from presentation layer to application layer, 100–103
  - document management, 94–95
  - domain services, building, 176–179
  - fixed user/role association, 93
  - flexible user/asset/role association, 93
  - functions maps, 77–78
  - fundamental tasks, 77–79
  - IIS middleware, 79
  - introduction to (overview), 66
  - Minimal API endpoints, 81–82
  - modular monoliths, transitioning to microservices, 249–252
  - MVC methods, 80–81
  - NuGet, 73
  - permissions, 94–95
  - personas, 69
  - product-related features, 78–79
  - Razor page code-behind classes, 81
  - release notes, 66–67
    - creation tools, 68
    - writing, 67
  - sharing documents, 95
  - task orchestration, 96
    - defining tasks, 96–97
    - distributed tasks, 97–99
    - example task, 99
  - user access control, 78
  - user authentication, 92
  - Visual Studio, 95
  - Youbiquitous.Martlet, 73
- protobufs (Protocol Buffers), 235
- prototyping, rapid, 291–292
- pure/impure domain services, 185–186

## Q

- QCon London 2023, 237
- quality bar, raising, 298
- queries
  - command/query separation, 208–213
  - compiled queries, EF Core, 204–205
  - CQRS, 12–13
    - architectural perspective, 209–210
    - business perspective, 210–211
    - distinct databases, 211-
    - shared databases, 211
  - LINQ, 154–155
  - SQL, 139

## R

- rapid prototyping, 291–292
- Razor
  - ASP.NET web stacks, 275–277
  - pages, code-behind classes, 81
- RDBMS (Relational Database Management Systems), 139
- React, 261, 262, 265–266
- readability
  - coding
    - author’s experience with, 165–166
    - writing, 165–166
  - maintainability, 57
- readiness of infrastructures, microservices, 238
- reducing technical debt
  - inventory of debt items, 289
  - separating projects, 288–289
  - ship-and-remove strategy, 289
  - slack time, 290
  - spikes, 290
  - timeboxing, 290
- refactoring
  - goals of, 296
  - as learning experience, 296
  - power of, 295
  - technical credit, 293
  - technical debt, 295
- reference classes, WCF, 191
- reformulating exceptions, 121–122
- registering loggers, 113–114
- relationships, aggregates, 143
- release notes, 66
  - creation tools, 68
  - writing, 67
- reloading (hot), application settings, 112–113
- rendering
  - HTML, 264–265
  - SSR, 86–87, 270, 273–274
- repositories
  - classes, 193–196
  - data transfers, 104
  - domain services versus, 193
  - expanding scope of, 186
  - patterns, 193–196
- REPR pattern, domain services, 180
- ReSharper and Visual Studio, 166
- response view model, data transfers, 102
- REST API (Representational State Transfer API), 266–269
- reusability
  - coding, 196
  - maintainability, 57–58
  - modularization, 49
- rich frameworks
  - accessibility, 269
  - BFF, 270
  - drawbacks of, 269–270
  - performance overhead, 269
  - SEO, 269
  - SSR, 270
- roles, Project Renoir
  - fixed user/role association, 93

- flexible user/asset/role association, 93
- roots, business domain model, 143–144
- rules (business), handling with strategy pattern, 182–183

## S

- scalability
  - DDD architectures, 9
  - event sourcing, 16
  - horizontal scalability, 54
  - maintainability, 58
  - microservices, 228, 237–238
  - monoliths, 54, 222
- scope creep, 291
- scripting
  - client-scripting, 260
    - AJAX, 260
    - Angular, 261–262, 265–266
    - CSR, 262
    - JavaScript, modern application frameworks, 261–262
    - React, 261, 262, 265–266
  - server-side scripting, 257–258
    - ASP.NET Web Forms, 258–259
    - MVC patterns, 259–260
- security, domain services, 187
- sending business emails with domain services, 174–175
- SEO (Search Engine Optimization), 269
- “Separating Data from Function in a Distributed File System” 4
- separating front-end and back-end, 274
  - data, 274–275
  - markups, 274–275
  - single web stacks, 274
- serverless environments, microservices, 244–245
- server-side scripting, 257–258
  - ASP.NET Web Forms, 258–259
  - MVC patterns, 259–260
- service to hash passwords, 175–176
- services
  - centralized logging services, microservices, 230–231
  - domain services, 11, 52, 169
    - application services versus, 184–185
    - authorization, 187
    - blinking at domain events, 174
    - building, 176–179
    - common scenarios, 173–177, 187
    - customer loyalty status, determining, 173
    - data access, 172
    - data injection, 172
    - data normalization, 187
    - defined, 170
    - dependencies, injections, 180–182
    - dependencies, on other functions, 177
    - microservice coordination, 229–230
- session states, modular monoliths, 249
- sharing
  - documents, Project Renoir, 95
  - files/databases
    - CQRS, 211

## sharing

- infrastructure layer, 191
  - kernels, bounded contexts, 40
  - states, modular monoliths, 249
  - UL glossaries, 32–33
  - ship-and-remove strategy, reducing technical debt, 289
  - SignalR connection hubs, 126
    - monitoring hubs, 126–127
    - notifications, sending to client browsers, 128
    - propagating, 127–128
  - single web stacks, front-end and back-end separation, 275
  - skills (debt amplifiers), lack of, 292
  - slack time, 290
  - smart endpoints, 227
  - SOA (Service-Oriented Architectures), 224, 225–226
    - microservices and, 237
    - tenets of, 224–225
  - SoC (Separation of Concerns), 7, 9, 20, 47, 48–49
  - software
    - anemia, 148–149
    - architectures
      - modular monoliths, 248
      - Zen of, 297
    - models, defined, 27
    - monoliths, defined, 5
    - projects, business-requirements engineering, 74–75
      - Agile Manifesto, The, 75
      - waterfall model, 75
  - SOLID acronym, 166–167, 294
  - SPA (Single-Page Applications), 260–261
  - spacing, style conventions, 162
  - special case pattern, 183–184
  - spikes, 290
  - Spolsky, Joel, 240–241
  - SQL (Structured Query Language), 139
  - SRP (Single Responsibility Principle), 166–167
  - SSE (Simplest Solution Ever), 56–57
  - SSG (Static Site Generation), 271–272
  - SSR (Server-Side Rendering), 86–87, 270, 273–274
  - Stack Overflow, 240–241
  - stateless nature of domain services, 170
  - states, loading into domain entities, 144–145
  - sticky sessions, modular monoliths, 249
  - storage
    - data
      - architectures, 208
      - infrastructure layer, 190
    - procedures
      - EF Core, 207–208
      - pros/cons, 207
  - storyboards, event-based, 76–77
  - strategic analysis, DDD architectures, 24
    - bounded contexts, 25–26
    - business domains
      - breakdowns, 24–25
      - subdomains, 25
      - top-level architectures, 24
    - microservices, 25–26
  - strategy pattern (business rules), 182–183
  - style conventions
    - braces ({}), 161
    - clean code etiquette, 161–165
    - comments, 164–165
    - consistent naming conventions, 161
    - indentation, 161
    - line length, 163–164
    - meaningful naming, 161
    - method length, 163–164
    - partial classes, 162
    - spacing, 162
    - Visual Studio regions, 163
  - subdomains, business domains, 25
  - suppliers, context maps, 43
  - Svelte, 270–271, 276
  - swallowing exceptions, 121–122
  - syntactic sugar, 156–157
- ## T
- tactical design, DDD architectures, 26–27
  - task orchestration, 96
    - defining tasks, 96–97
    - Project Renoir example, 99
  - TDD (Test-Driven Design), 50, 60
  - technical credit, 285
    - Agile methodologies, 294–295
    - broken window theory, 293–295
    - design principles, 294
    - profit of, 293
    - refactoring, 293
    - testability, 295
  - technical debt
    - amplifiers, 292
      - lack of documentation, 290–291
      - lack of skills, 292
      - rapid prototyping, 291–292
      - scope creep, 291
    - costs, 285–286
    - defined, 285
    - genesis of, 296–297
    - impact of, 289
    - pragmatic perspective, 286–287
    - quality bar, raising, 298
    - reasons for creating debt, 287
    - reducing
      - inventory of debt items, 289
      - separating projects, 288–289
      - ship-and-remove strategy, 289
      - slack time, 290
      - spikes, 290
      - timeboxing, 290
    - refactoring, 295
    - signs of, 287–288
    - Zen of coding, 297–298
    - Zen of software infrastructure, 297
  - technical terms, UL glossaries, 32
  - technology diversity

- microservices, 55–56, 239
- monoliths, 223
- templates, text, 263–264
- terms (business), Project Renoir, 68
- testability, 295
  - design patterns, 59–60
  - modularization, 50
  - principles of (overview), 58–59
  - TDD, 60
- testing modular monoliths, 248
- text templating, 263–264
- Theorem of Incompleteness, 3
- three-tier architectures, 4–5, 10
  - defining
    - layers, 5
    - tiers, 5
  - software monoliths, 5
  - U.S. Prohibition Act, 7
  - value of  $N$ , 6
- throwing exceptions, 119–122
- tiers, defined, 5
- timeboxing, 290
- Tolkien, J.R.R.255
- top-level base cases, entities, 149–150
- tracking objects, disabling with EF Core, 204
- transactional boundaries, aggregates, 143
- transferring data
  - from application layer to persistence layer, 104–106
  - business logic errors, 105–106
  - disconnecting from HTTP context, 100–101
  - domain entities, 103
  - DTO
    - C# implementations, 100
    - defined, 99
  - input view model, 101–102
  - from presentation layer to application layer, 100–103
  - repositories, 104
  - response view model, 102
- Turing, Alan, 3
- Twain, Mark, 23

## U

- UL (Ubiquitous Language)
  - changes to languages, 33, 34–35
  - context maps, 68–69
  - DDD architectures, 29
  - domain services, 171
  - factory methods, 36
  - glossaries
    - acronyms, 32
    - building, 31–32
    - choosing natural language of glossaries, 30
    - shared terms, 30
    - sharing, 32–33
    - technical terms, 32
  - goal of, 33
  - impact on coding, 33–34
  - value types, 36

- UoW pattern, 196
- upstream context, context maps, 42–43
- U.S. Prohibition Act, 7
- use cases
  - event modeling, 77
  - workflows, application layer, 108–109
- users, Project Renoir
  - access control, 78
  - authentication, 92
    - fixed user/role association, 93
    - flexible user/asset/role association, 93

## V

- validation, domains, 187
- value objects, business domain model, 134
- value of  $N$ , 6
- value types, DDD architectures, 36
- values (constant), clean code etiquette, 159–160
- Vanilla JavaScript, 272–273, 275–276
- Visual Studio
  - Project Renoir, 95
  - regions, style conventions, 163
  - ReSharper and, 166
- Von Neumann, John, 3, 4
- VSA (Vertical Slice Architectures), 21
- Vue.js framework, 272, 276–277

## W

- waterfall model, 75
- WCF (Windows Communication Foundation)
  - microservices, 225–226
  - reference classes, 191
- Web API, 191
- web applications
  - API layer, 266
    - GraphQL API, 266–269
    - REST API, 266–269
  - application layer deployments, 129
  - brief history of, 256
  - defined, 6–7
  - HTML layer
    - front-end pages, 263
    - rendering HTML, 264–265
    - SSG, 272
    - Svelte, 270–271
    - text templating, 263–264
  - rich frameworks
    - accessibility, 269
    - BFF, 270
    - drawbacks of, 269–270
    - performance overhead, 269
    - SEO, 269
    - SSR, 270
    - SPA, 260–261
    - SSG, 271–272

## web browser wars

- web browser wars, 257
- Web Forms, ASP.NET, 258–259
- web stacks, front-end and back-end separation, 274
- Wilde, Oscar, 285
- Wisej, 87–88
- workflows (use-case), application layer, 108–109
- write-through patterns, caching, 124
- writing
  - readable code, 165–166
  - release notes, 67

## X - Y

- YAGNI (You Aren't Gonna Need It), 294
- Youbiquitous.Martlet, Project Renoir, 73

## Z

- Zave, Dr. Pamela, 3
- Zen of coding, 297–298
- Zen of software infrastructure, 297
- Zerox PARC computer scientists, 4