Microsoft

Clean Architecture with .NET

Professional

Dino Esposito

FREE SAMPLE CHAPTER |

Microsoft

Clean Architecture with .NET

Dino Esposito

Clean Architecture with .NET

Published with the authorization of Microsoft Corporation by:

Pearson Education, Inc., Hoboken, New Jersey

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

ISBN-10: 0-13-820328-8

Library of Congress Control Number: 2024930932

$PrintCode

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

This page intentionally left blank

Contents

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 codebase 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*.

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

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*

CHAPTER 2

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.

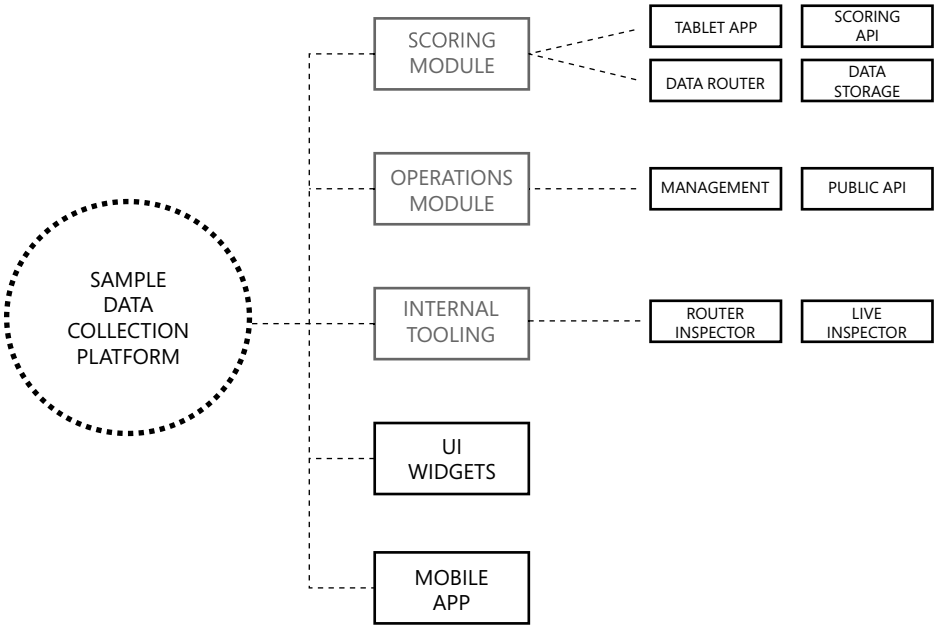

FIGURE 2-1 Breakdown of a business domain.

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.

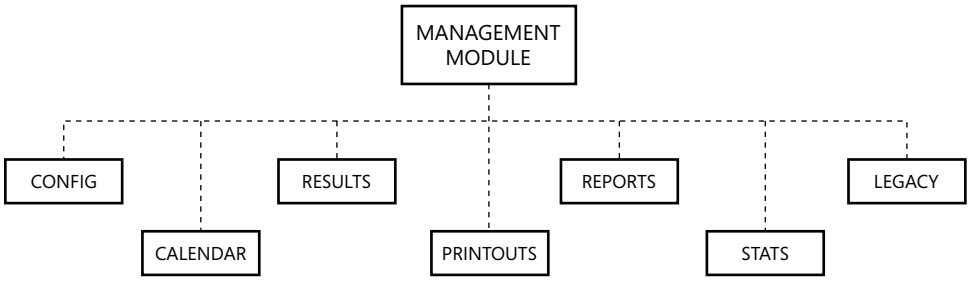

FIGURE 2-2 Further functional split of the Management module.

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.)

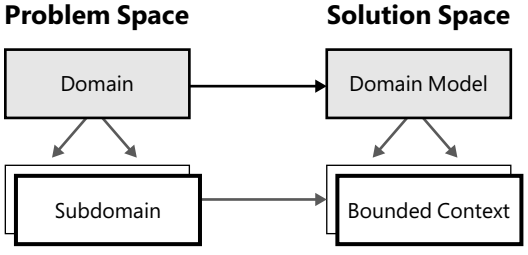

FIGURE 2-3 Domain and subdomains versus domain models and bounded contexts.

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.

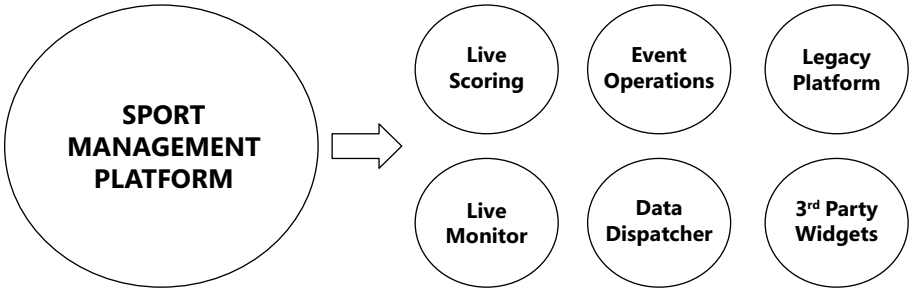

FIGURE 2-4 Breakdown of an example domain model in a sport-tech scenario.

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.)

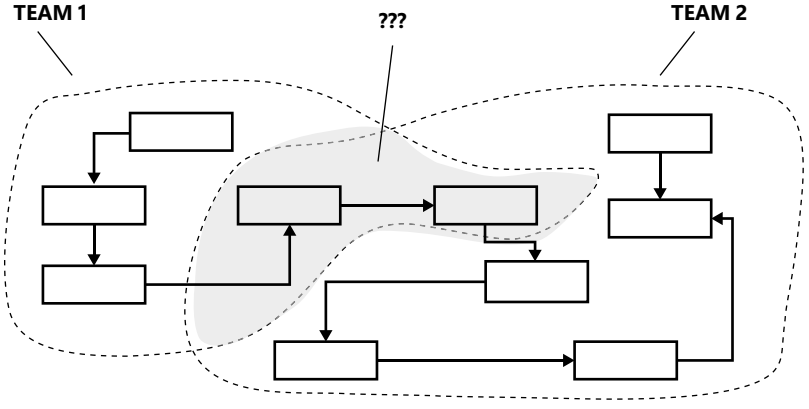

FIGURE 2-5 Discovering a shared kernel.

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.)

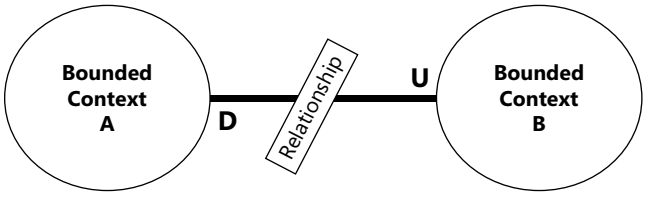

FIGURE 2-6 Graphical notation of a context map relationship.

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.

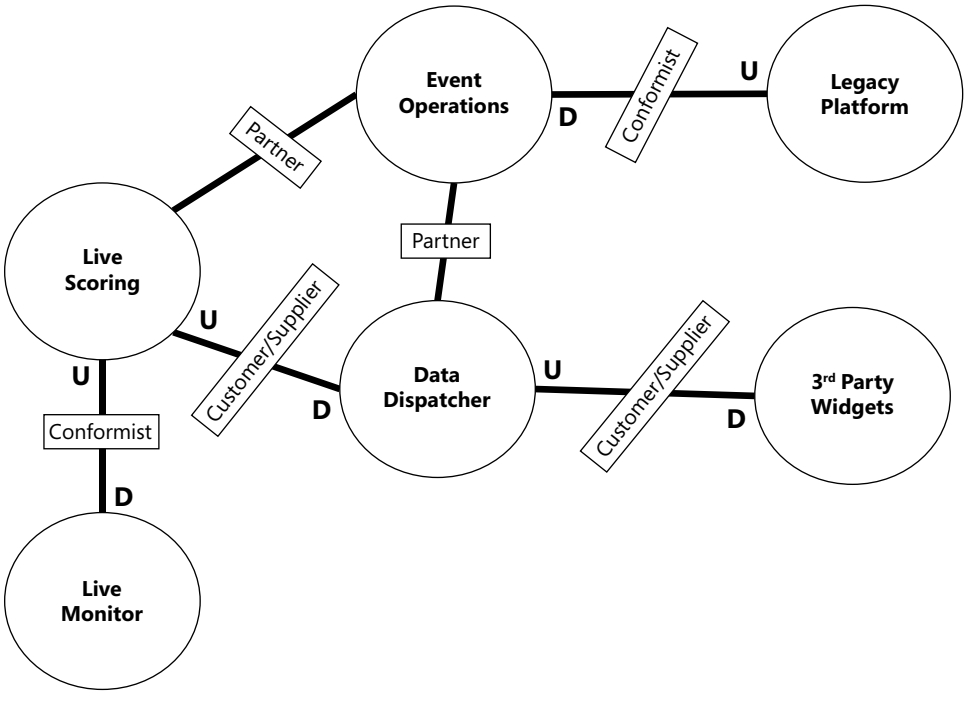

FIGURE 2-7 An example context map for the bounded contexts identified in Figure 2-4.

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.

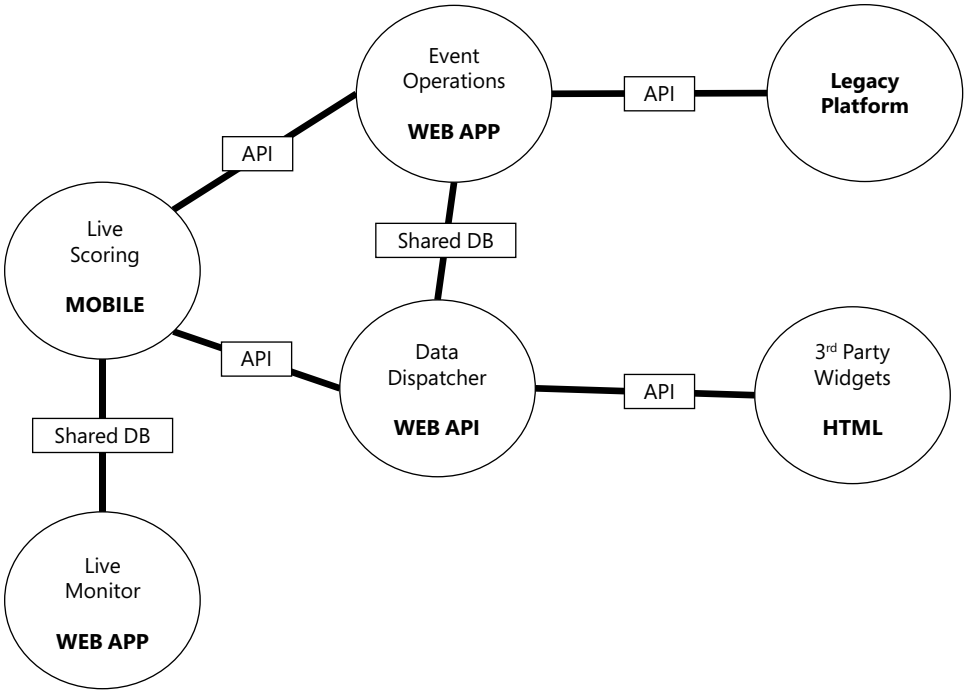

FIGURE 2-8 An example deployment map.

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

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

X - Y

Z