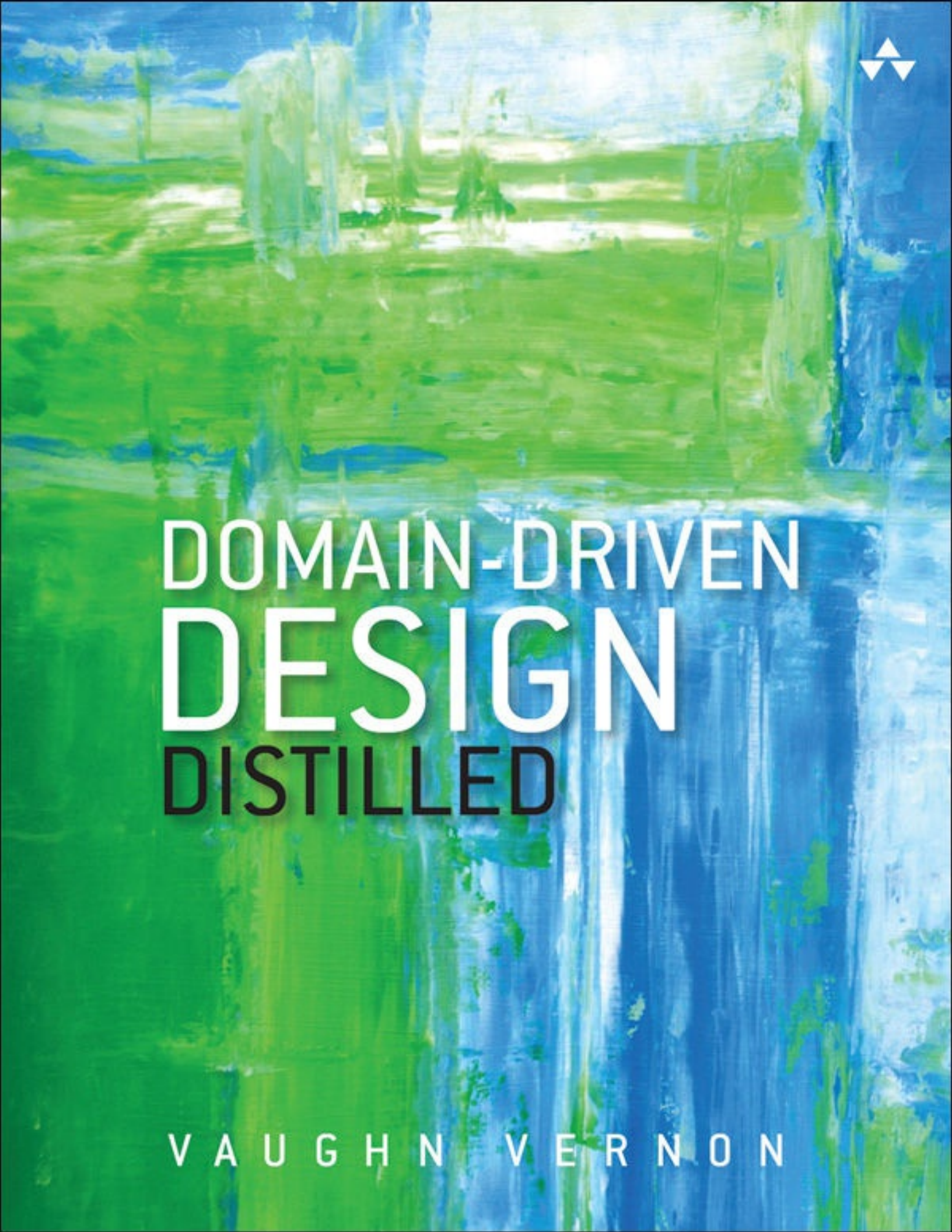

DOMAIN-DRIVEN
DESIGN
DISTILLED
VAUGHN VERNON

About This E-Book

EPUB is an open, industry-standard format for e-books. However, support for EPUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the e-book in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

Domain-Driven Design Distilled

Vaughn Vernon

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2016936587

ISBN-13: 978-0-13-443442-1
ISBN-10: 0-13-443442-0
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, May 2016

Nicole and Tristan
We did it again!

Contents

Preface

Why is model building such a fun and rewarding activity? Ever since I was a kid I have loved to build models. At that time I mostly built models of cars and airplanes. I am not sure where LEGO was in those days. Still, LEGO has been a big part of my son's life since he was very young. It is so fascinating to conceive and build models with those small bricks. It's easy to come up with basic models, and it seems you can extend your ideas almost endlessly.

You can probably relate to some kind of youthful model building.

Models occur in so many situations in life. If you enjoy playing board games, you are using models. It might be a model of real estate and property owners, or models of islands and survivors, or models of territories and building activities, and who knows what all. Similarly, video games are models. Perhaps they model a fantasy world with fanciful characters playing fantastic roles. A deck of cards and related games model power. We use models all the time and probably so often that we don't give most models a well-deserved acknowledgment. Models are just part of our lives.

But why? Every person has a learning style. There are a number of learning styles, but three of the most discussed are auditory, visual, and tactile styles. The auditory learners learn by hearing and listening. The visual learners learn by reading or seeing imagery. The tactile learners learn by doing something that involves touching. It's interesting that each learning style is heavily favored by the individual to the extent that he or she can sometimes have trouble with other types of learning. For example, tactile learners likely remember what they have done but may have problems remembering what was said during the process. With model building, you would think that visual and tactile learners would have a huge advantage over the auditory learners, because model building seems to mostly involve visual and tactile stimulation. However, that might not always hold true, especially if a team of model builders uses audible communication in their building process. In other words, model building holds out the possibility to accommodate the learning style of the vast majority of individuals.

With our natural affinity to learning through model building, why would we not naturally desire to model the software that ever increasingly assists and influences our lives? In fact, to model software appears to be, well, human. And model software we should. It seems to me that humans should be elite software model builders.

It is my strong desire to help you be as human as you can possibly be by modeling software using some of the best software modeling tools available. These tools are packaged under the name "Domain-Driven Design," or DDD. This toolbox, actually a set of patterns, was first codified by Eric Evans in the book *Domain-Driven Design: Tackling Complexity in the Heart of Software* [DDD] . It is my vision to bring DDD to everyone possible. To make my point, if I must say that I want to bring DDD to the masses, then so be it. That is where DDD deserves to be, and DDD is the toolbox that model-oriented humans deserve to use to create their most advanced software models. With this book, I am determined to make learning and using DDD as simple and easy as possible and to bring that to the broadest conceivable audience.

For auditory learners, DDD holds out the prospect of learning through the team communication of building a model based on the development of a *Ubiquitous Language*. For visual and tactile learners, the process of using DDD tools is very visual and hands-on as your team models both strategically and tactically. This is especially true when drawing *Context Maps* and modeling the

business process using *Event Storming*. Thus, I believe that DDD can support everyone who wants to learn and achieve greatness through model building.

Who Is This Book For?

This book is for everyone interested in learning the most important DDD aspects and tools and in learning quickly. The most common readers are software architects and software developers who will put DDD into practice on projects. Very often, software developers quickly discover the beauty of DDD and are keenly attracted to its powerful tooling. Even so, I have made the subject understandable for executives, domain experts, managers, business analysts, information architects, and testers alike. There's really no limit to those in the information technology (IT) industry and research and development (R&D) environments who can benefit from reading this book.

If you are a consultant and you are working with a client to whom you have recommended the use of DDD, provide this book as a way to bring the major stakeholders up to speed quickly. If you have developers—perhaps junior or midlevel or even senior—working on your project who are unfamiliar with DDD but need to use it very soon, make sure that they read this book. By reading this book, at minimum, all the project stakeholders and developers will have the vocabulary and understand the primary DDD tools being used. This will enable them to share things meaningfully as they move the project forward.

Whatever your experience level and role, read this book and then practice DDD on a project. Afterward, reread this book and see what you can learn from your experiences and where you can improve in the future.

What This Book Covers

The first chapter, "DDD for Me," explains what DDD can do for you and your organization and provides a more detailed overview of what you will learn and why it's important.

Chapter 2 , "Strategic Design with Bounded Contexts and the Ubiquitous Language ," introduces DDD strategic design and teaches the cornerstones of DDD, *Bounded Contexts* and the *Ubiquitous Language*. Chapter 3 , "Strategic Design with Subdomains ," explains *Subdomains* and how you can use them to deal with the complexity of integrating with existing legacy systems as you model your new applications. Chapter 4 , "Strategic Design with Context Mapping ," teaches the variety of ways that teams work together strategically and ways that their software can integrate. This is called *Context Mapping*.

Chapter 5 , "Tactical Design with Aggregates ," switches your attention to tactical modeling with *Aggregates*. An important and powerful tactical modeling tool to be used with *Aggregates* is *Domain Events* , which is the subject of Chapter 6 , "Tactical Design with Domain Events ."

Finally, in Chapter 7 , "Acceleration and Management Tools ," the book highlights some project acceleration and project management tools that can help teams establish and maintain their cadence. These two topics are seldom if ever discussed in other DDD sources and are sorely needed by those who are determined to put DDD into practice.

Conventions

There are only a few conventions to keep in mind while reading. All of the DDD tools that I discuss are printed in italics. For example, you will read about *Bounded Contexts* and *Domain Events*.

Another convention is that any source code is presented in a monospaced Courier font. Acronyms and abbreviations for works listed in the References on pages 136 -137 appear in square brackets throughout the chapters.

Even so, what this book emphasizes most, and what your brain should like a lot, is visual learning through lots of diagrams and figures. You will notice that there are no figure numbers in the book, because I didn't want to distract you with so many of those. In every case the figures and diagrams precede the text that discusses them, which means that the graphic visuals introduce thoughts as you work your way through the book. That means that when you are reading text, you can count on referring back to the previous figure or diagram for visual support.

Acknowledgments

This is now my third book within the esteemed Addison-Wesley label. It's also my third time working with my editor, Chris Guzikowski, and developmental editor, Chris Zahn, and I am happy to say that the third time has been as much a charm as the first two. Thanks again for choosing to publish my books.

As always, a book cannot be successfully written and published without critical feedback. This time I turned to a number of DDD practitioners who don't necessarily teach or write about it but are nonetheless working on projects while helping others use the powerful toolbox. I felt that this kind of practitioner was crucial to make sure this aggressively distilled material said exactly what is necessary and in just the right way. It's kind of like, if you want me to talk for 60 minutes, give me 5 minutes to prepare; if you want me to talk for 5 minutes, give me a few hours to prepare.

In alphabetical order by last name, those who helped me the most were Jérémie Chassaing, Brian Dunlap, Yuji Kiriki, Tom Stockton, Tormod J. Varhaugvik, Daniel Westheide, and Philip Windley. Thanks much!

About the Author

Vaughn Vernon is a veteran software craftsman and thought leader in simplifying software design and implementation. He is the author of the best-selling books *Implementing Domain-Driven Design* and *Reactive Messaging Patterns with the Actor Model,* also published by Addison-Wesley. He has taught his IDDD Workshop around the globe to hundreds of software developers. Vaughn is a frequent speaker at industry conferences. He is most interested in distributed computing, messaging, and in particular with the Actor model. Vaughn specializes in consulting around Domain-Driven Design and DDD using the Actor model with Scala and Akka. You can keep up with Vaughn's latest work by reading his blog at www.VaughnVernon.co and by following him on his Twitter account @VaughnVernon.

Chapter 1. DDD for Me

You want to improve your craft and to increase your success on projects. You are eager to help your business compete at new heights using the software you create. You want to implement software that not only correctly models your business's needs but also performs at scale using the most advanced software architectures. Learning *Domain-Driven Design* (DDD), and learning it quickly, can help you achieve all of this and more.

DDD is a set of tools that assist you in designing and implementing software that delivers high value, both strategically and tactically. Your organization can't be the best at everything, so it had better choose carefully at what it must excel. The DDD strategic development tools help you and your team make the competitively best software design choices and integration decisions for your business. Your organization will benefit most from software models that explicitly reflect its core competencies. The DDD tactical development tools can help you and your team design useful software that accurately models the business's unique operations. Your organization should benefit from the broad options to deploy its solutions in a variety of infrastructures, whether in house or in the cloud. With DDD, you and your team can be the ones to bring about the most effective software designs and implementations needed to succeed in today's competitive business landscape.

In this book I have distilled DDD for you, with condensed treatment of both the strategic and tactical modeling tools. I understand the unique demands of software development and the challenges you face as you work to improve your craft in a fast-paced industry. You can't always take months to read up on a subject like DDD, and yet you still want to put DDD to work as soon as possible.

I am the author of the best-selling book *Implementing Domain-Driven Design* [IDDD] , and I have also created and teach the three-day IDDD Workshop. And now I have also written this book to bring you DDD in an aggressively condensed form. It's all part of my commitment to bringing DDD to every software development team, where it deserves to be. My goal, of course, includes bringing DDD to you.

Will DDD Hurt?

You may have heard that DDD is a complicated approach to software development. Complicated? It certainly is not complicated of necessity. Indeed, it is a set of advanced techniques to be used on complex software projects. Due to its power and how much you have to learn, without expert instruction it can be daunting to put DDD into practice on your own. You have probably also found that some of the other DDD books are many hundreds of pages long and far from easy to consume and apply. It required a lot of words for me to explain DDD in great detail in order to provide an exhaustive implementation reference on more than a dozen DDD topics and tools. That effort resulted in *Implementing Domain-Driven Design* [IDDD] . This new condensed book is provided to familiarize you with the most important parts of DDD as quickly and simply as possible. Why? Because some are overwhelmed by the larger texts and need a distilled guide to help them take the initial steps to adoption. I have found that those who use DDD revisit the literature about it several times. In fact, you might even conclude that you will never learn enough, and so you will use this book as a quick reference, and refer to others for more detail, a number of times as your craft is refined. Others have had trouble selling DDD to their colleagues and the all-important management team. This book will help you do that, not only by explaining DDD in a condensed format, but also by showing that tools are available to accelerate and manage its use.

Of course, it is not possible to teach you everything about DDD in this book, because I have purposely distilled the DDD techniques for you. For much more in-depth coverage, see my book *Implementing Domain-Driven Design* [IDDD] , and look into taking my three-day IDDD Workshop. The three-day intensive course, which I have delivered around the globe to a broad audience of hundreds of developers, helps get you up to speed with DDD rapidly. I also provide DDD training

online at http://ForComprehension.com .

The good news is, DDD doesn't have to hurt. Since you probably already deal with complexity on your projects, you can learn to use DDD to reduce the pain of winning over complexity.

Good, Bad, and Effective Design

Often people talk about good design and bad design. What kind of design do you do? Many software development teams don't give design even a passing thought. Instead, they perform what I call "the task-board shuffle." This is where the team has a list of development tasks, such as with a Scrum product backlog, and they move a sticky note from the "To Do" column of their board to the "In Progress" column. Coming up with the backlog item and performing "the task-board shuffle" constitutes the entirety of thoughtful insights, and the rest is left to coding heroics as programmers blast out the source. It rarely turns out as well as it could, and the cost to the business is usually the highest price paid for such nonexistent designs.

This often happens due to the pressure to deliver software releases on a relentless schedule, where management uses Scrum to primarily control timelines rather than allow for one of Scrum's most important tenets: *knowledge acquisition.*

When I consult or teach at individual businesses, I generally find the same situations. Software projects are in peril, and entire teams are hired to keep systems up and running, patching code and data daily. The following are some of the insidious problems that I find, and interestingly ones that DDD can readily help teams avoid. I start with the higher-level business problems and move to the more technical ones:

- Software development is considered a cost center rather than a profit center. Generally this is because the business views computers and software as necessary nuisances rather than sources of strategic advantage. (Unfortunately there may not be a cure for this if the business culture is firmly fixed.)
- Developers are too wrapped up with technology and trying to solve problems using technology rather than careful thought and design. This leads developers to constantly chase after new "shiny objects," which are the latest fads in technology.
- The database is given too much priority, and most discussions about the solutions center around the database and a data model rather than business processes and operations.
- Developers don't give proper emphasis to naming objects and operations according to the business purpose that they fill. This leads to a large chasm between the mental model that the business owns and the software that developers deliver.
- The previous problem is generally a result of poor collaboration with the business. Often the business stakeholders spend too much time in isolated work producing specifications that nobody uses, or that are only partly consumed by developers.
- Project estimates are in high demand, and very frequently producing them can add significant time and effort, resulting in the delay of software deliverables. Developers use the "task-board shuffle" rather than thoughtful design. They produce a *Big Ball of Mud* (discussed in the following chapters) rather than appropriately segregating models according to business drivers.
- Developers house business logic in user interface components and persistence components. Also, developers often perform persistence operations in the middle of business logic.

- There are broken, slow, and locking database queries that block users from performing time-sensitive business operations.
- There are wrong abstractions, where developers attempt to address all current and imagined future needs by overly generalizing solutions rather than addressing actual concrete business needs.
- There are strongly coupled services, where an operation is performed in one service, and that service calls directly to another service to cause a balancing operation. This coupling often leads to broken business processes and unreconciled data, not to mention systems that are very difficult to maintain.

This all seems to happen in the spirit of "no design yields lower-cost software." And all too often it is simply a matter of businesses and the software developers not knowing that there is a much better alternative. "Software is eating the world" [WSJ] , and it should matter to you that software can also eat your profits, or feed your profits a banquet.

It's important to understand that the imagined economy of No Design is a fallacy that has cleverly fooled those who apply the pressure to produce software without thoughtful design. That's because design still flows from the brains of the individual developers through their fingertips as they wrangle with the code, without any input from others, including the business. I think that this quote sums up the situation well:

> Questions about whether design is necessary or affordable are quite beside the point: design is inevitable. The alternative to good design is bad design, not no design at all.
>
> —*Book Design: A Practical Introduction* by Douglas Martin

Although Mr. Martin's comments are not specifically about software design, they are still applicable to our craft, where there is no substitute for thoughtful design. In the situation just described, if you have five software developers working on the project, No Design will actually produce an amalgamation of five different designs in one. That is, you get a blend of five different made-up business language interpretations that are developed without the benefit of the real *Domain Experts*.

The bottom line: we model whether we acknowledge modeling or not. This can be likened to how roads are developed. Some ancient roads started out as cart paths that were eventually molded into well-worn trails. They took unexplained turns and made forks that served only a few who had rudimentary needs. At some point these pathways were smoothed and then paved for the comfort of the increasing number of travelers who used them. These makeshift thoroughfares aren't traveled today because they were well designed, but because they exist. Few of our contemporaries can understand why traveling one of these is so uncomfortable and inconvenient. Modern roads are planned and designed according to careful studies of population, environment, and predictable flow. Both kinds of roads are modeled. One model employed minimal, base intellect. The other model exploited maximum cognition. Software can be modeled from either perspective.

If you are afraid that producing software with thoughtful design is expensive, think of how much more expensive it's going to be to live with or even fix a bad design. This is especially so when we are talking about software that needs to distinguish your organization from all others and yield considerable competitive advantages.

A word closely related to *good* is *effective* , and it possibly more accurately states what we should

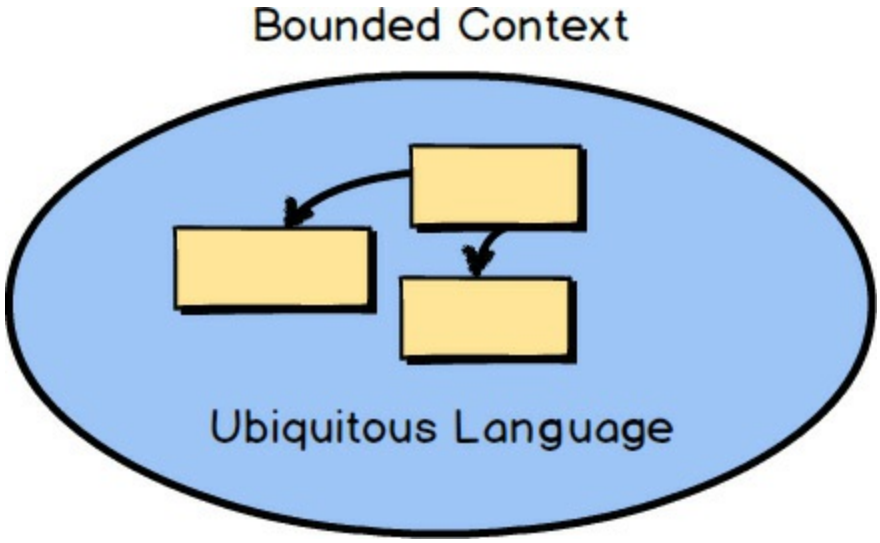
strive for in software design: *effective design*. Effective design meets the needs of the business organization to the extent that it can distinguish itself from its competition by means of software. Effective design forces the organization to understand what it must excel at and is used to guide the creation of the correct software model.

In Scrum, *knowledge acquisition* is done through experimentation and collaborative learning and is referred to as "buying information" [Essential Scrum] . Knowledge is never free, but in this book I do provide ways for you to accelerate how you come by it.

Just in case you still doubt that effective design matters, don't forget the insights of someone who seems to have understood its importance:

> Most people make the mistake of thinking design is what it looks like. People think it's this veneer—that the designers are handed this box and told, "Make it look good!" That's not what we think design is. It's not just what it looks like and feels like. Design is how it works.
>
> —Steve Jobs

In software, effective design matters, most. Given the single alternative, I recommend effective design.



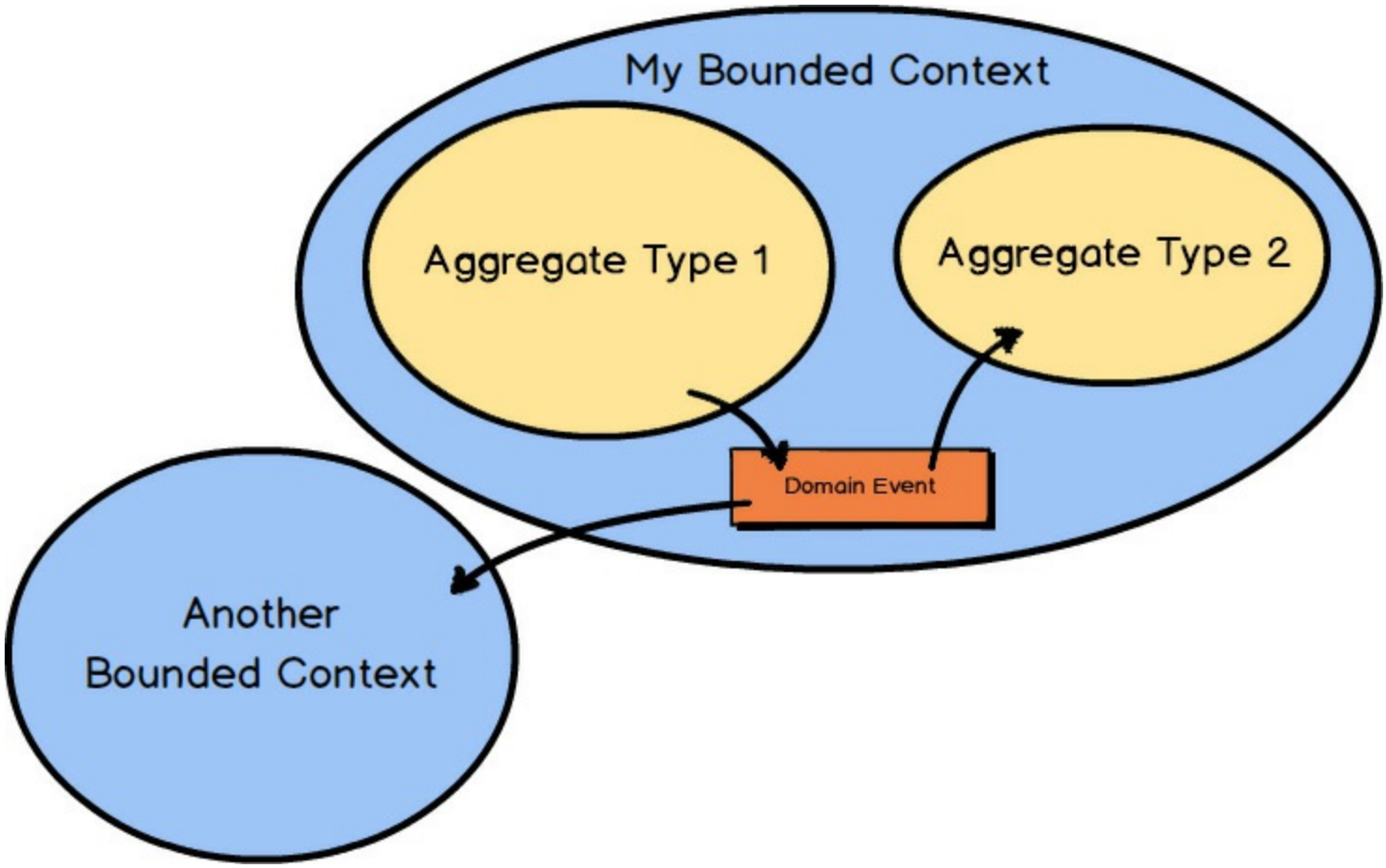
Strategic Design

We begin with the all-important strategic design. You really cannot apply tactical design in an effective way unless you begin with strategic design. Strategic design is used like broad brushstrokes prior to getting into the details of implementation. It highlights what is strategically important to your business, how to divide up the work by importance, and how to best integrate as needed.

First you will learn how to segregate your domain models using the strategic design pattern called *Bounded Contexts*. Hand in glove, you will see how to develop a *Ubiquitous Language* as your domain model within an explicitly *Bounded Context*.

You will learn about the importance of engaging with not only developers but also *Domain Experts* as you develop your model's *Ubiquitous Language*. You will see how a team of both software developers and *Domain Experts* collaborate. This is a vital combination of smart and motivated people who are needed for DDD to produce the best results. The language you develop together through collaboration will become ubiquitous, pervasive, throughout the team's spoken communication and software model.

As you advance further into strategic design, you will learn about *Subdomains* and how these can help you deal with the unbounded complexity of legacy systems, and how to improve your results on greenfield projects. You will also see how to integrate multiple *Bounded Contexts* using a technique called *Context Mapping*. *Context Maps* define both team relationships and technical mechanisms that exist between two integrating *Bounded Contexts*.



Tactical Design

After I have given you a sound foundation with strategic design, you will discover DDD's most prominent tactical design tools. Tactical design is like using a thin brush to paint the fine details of your domain model. One of the more important tools is used to aggregate entities and value objects together into a right-sized cluster. It's the *Aggregate* pattern.

DDD is all about modeling your domain in the most explicit way possible. Using *Domain Events* will help you both to model explicitly and to share what has occurred within your model with the systems that need to know about it. The interested parties might be your own local *Bounded Context* and other remote *Bounded Contexts*.

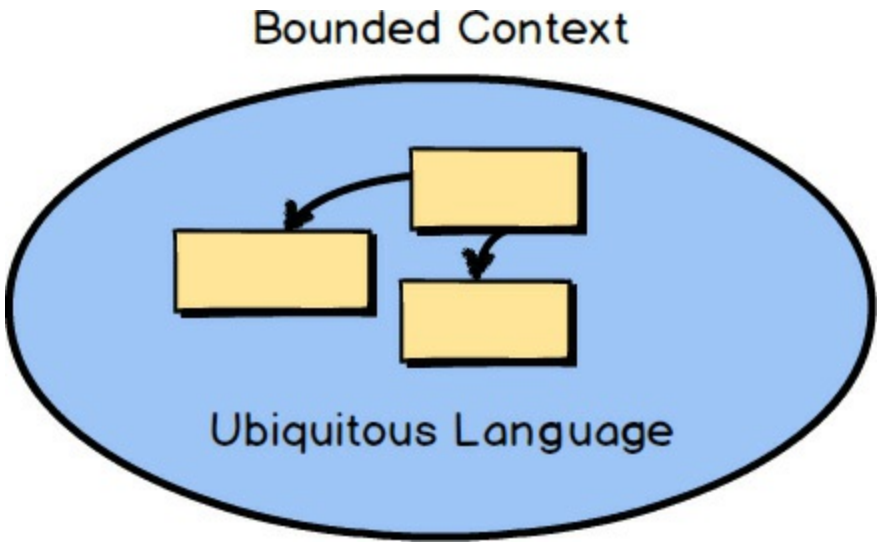
The Learning Process and Refining Knowledge

DDD teaches a way of thinking to help you and your team refine knowledge as you learn about your business's core competencies. This learning process is a matter of discovery through group conversation and experimentation. By questioning the status quo and challenging your assumptions about your software model, you will learn much, and this all-important knowledge acquisition will spread across the whole team. This is a critical investment in your business and team. The goal should be not only to learn and refine, but to learn and refine as quickly as possible. There are additional tools to help with those goals that can be found in Chapter 7 , "Acceleration and Management Tools ."

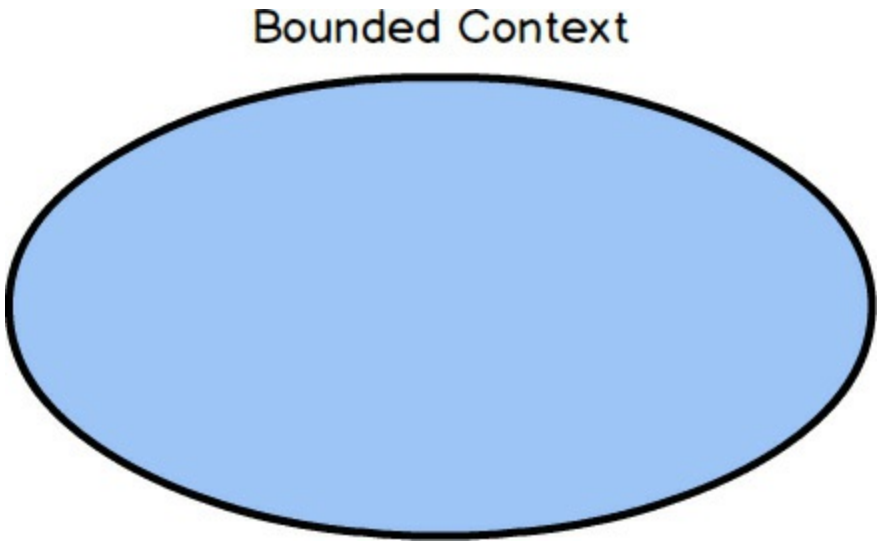
Let's Get Started!

Even in a condensed presentation, there's plenty to learn about DDD. So let's get started with Chapter 2 , "Strategic Design with Bounded Contexts and the Ubiquitous Language ."

Chapter 2. Strategic Design with Bounded Contexts and the Ubiquitous Language



What are these things called *Bounded Contexts* ? What's the *Ubiquitous Language* ? In short, DDD is primarily about modeling a *Ubiquitous Language* in an explicitly *Bounded Context.* While true, that probably wasn't the most helpful description that I could provide. Let me break this down for you.



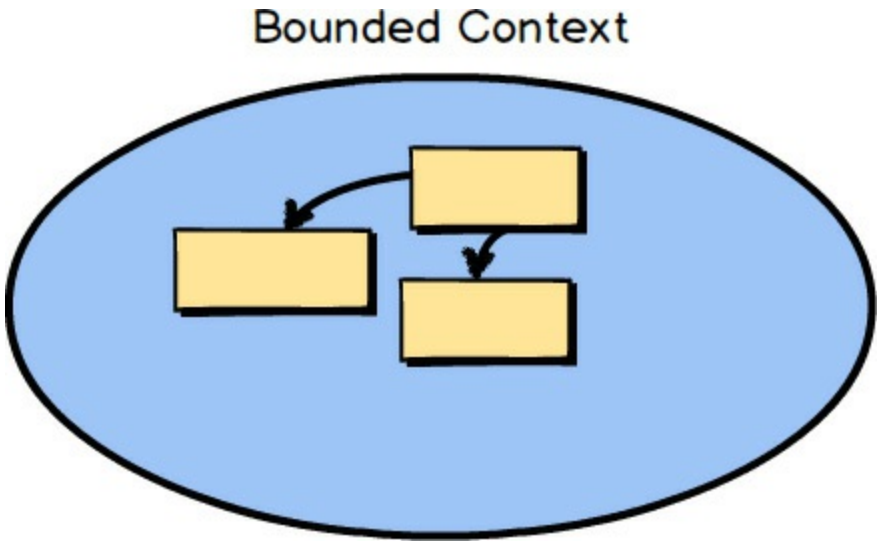
First, a *Bounded Context* is a semantic contextual boundary. This means that within the boundary each component of the software model has a specific meaning and does specific things. The components inside a *Bounded Context* are context specific and semantically motivated. That's simple enough.

When you are just getting started in your software modeling efforts, your *Bounded Context* is somewhat conceptual. You could think of it as part of your *problem space.* However, as your model starts to take on deeper meaning and clarity, your *Bounded Context* will quickly transition to your *solution space* , with your software model being reflected as project source code. (The *problem space* and *solution space* are better explained in the box.) Remember that a *Bounded Context* is where a model is implemented, and you will have separate software artifacts for each *Bounded Context.*

What Is a Problem Space and a Solution Space?

Your problem space is where you perform high-level strategic analysis and design steps within the constraints of a given project. You can use simple diagrams as you discuss the high-level project drivers and note important goals and risks. In practice, *Context Maps* work very well in the problem space. Note too that *Bounded Contexts* may be used in problem space discussions, when needed, but are also closely associated with your solution space.

Your solution space is where you actually implement the solution that your problem space discussions identify as your *Core Domain.* When the *Bounded Context* is being developed as a key strategic initiative of your organization, it's called the *Core Domain.* You develop your solution in the *Bounded Context* as code, both main source and test source. You will also produce code in your solution space that supports integration with other *Bounded Contexts* .



The software model inside the context boundary reflects a language that is developed by the team working in the *Bounded Context* and is spoken by every member of the team that creates the software model that functions within that *Bounded Context.* The language is called the *Ubiquitous Language* because it is both spoken among the team members and implemented in the software model. Thus, it is necessary that the *Ubiquitous Language* be rigorous—strict, exact, stringent, and tight. In the diagram, the boxes inside the *Bounded Context* represent the concepts of the model, which may be implemented as classes. When the *Bounded Context* is being developed as a key strategic initiative of your organization, it's called the *Core Domain.*

When compared with all the software your organization uses, a *Core Domain* is a software model that ranks among the most important, because it is a means to achieve greatness. A *Core Domain* is developed to distinguish your organization competitively from all others. At the very least it addresses a major line of business. Your organization can't excel at everything and shouldn't even try. So you choose wisely what should be part of your *Core Domain* and what should not. This is the primary value proposition of DDD, and you want to invest appropriately by committing your best resources to a *Core Domain.*

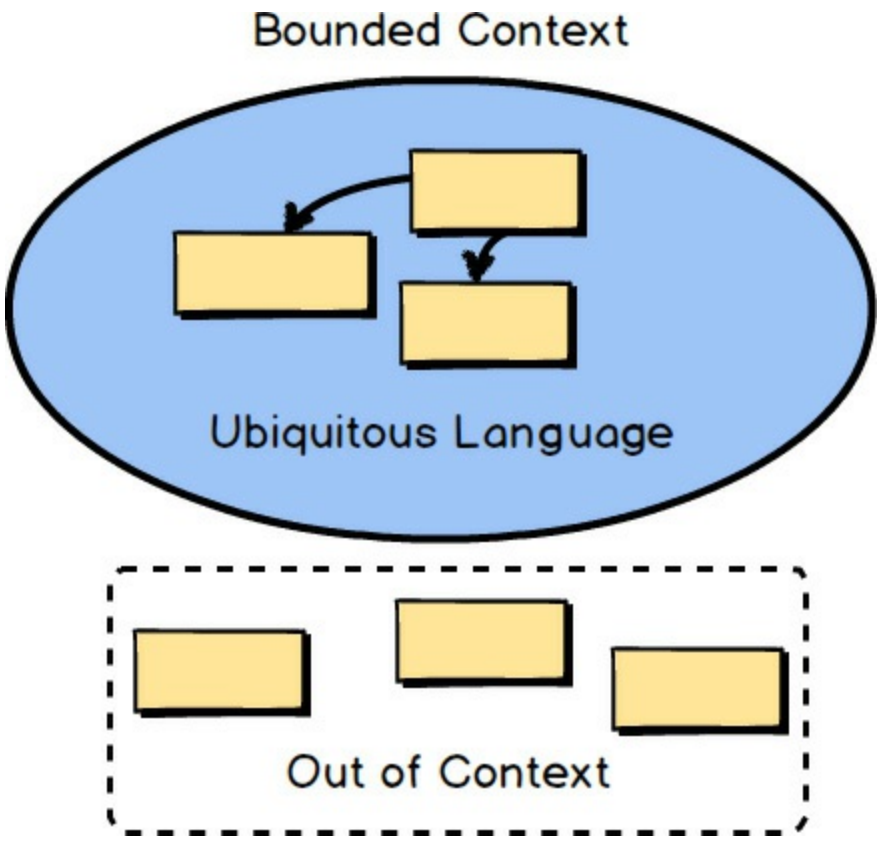
When someone on the team uses expressions from the *Ubiquitous Language* , everyone on the team understands what is meant with precision and constraints. The expression is ubiquitous within the team, as is all language used by the team that defines the software model being developed.

When you consider language in a software model, think of the various nations that make up Europe. Within one of the countries across this space, the official language of each country is clear. Within the boundaries of those nations—for example, Germany, France, and Italy—the official languages are certain. As you cross a boundary, the official language changes. The same goes for Asia, where Japanese is spoken in Japan, and the languages spoken in China and Korea are clearly different across the national boundaries. You can think of *Bounded Contexts* in much the same way, as being language boundaries. In the case of DDD, the languages are those spoken by the team that owns the software model, and a notable written form of the language is the software model's source code.

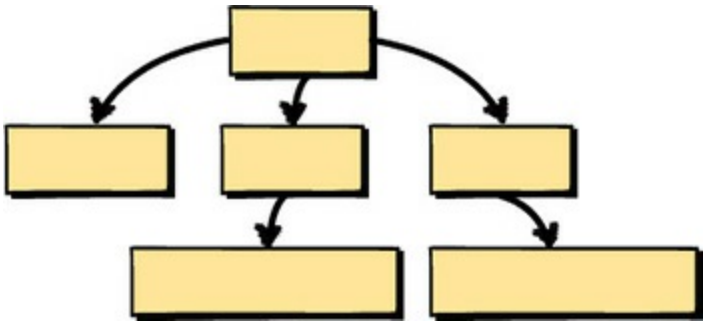
Bounded Contexts, Teams, and Source Code Repositories

There should be one team assigned to work on one *Bounded Context.* There should also be a separate source code repository for each *Bounded Context.* It is possible that one team could work on multiple *Bounded Contexts* , but multiple teams should not work on a single *Bounded Context.* Cleanly separate the source code and database schema for each *Bounded Context* in the same way that you separate the *Ubiquitous Language.* Keep acceptance tests and unit tests together with the main source code.

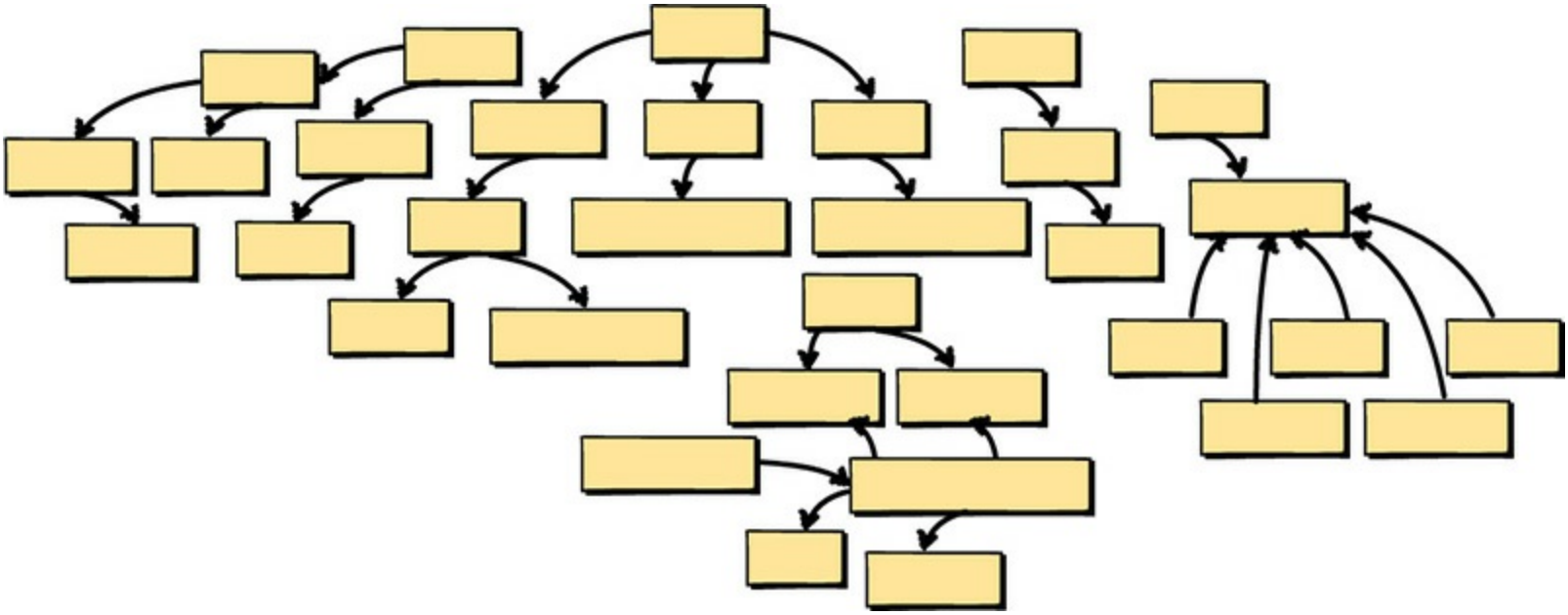
It is especially important to be clear that one team works on a single *Bounded Context.* This completely eliminates the chances of any unwelcome surprises that arise when another team makes a change to your source code. Your team owns the source code and the database and defines the official interfaces through which your *Bounded Context* must be used. It's a benefit of using DDD.



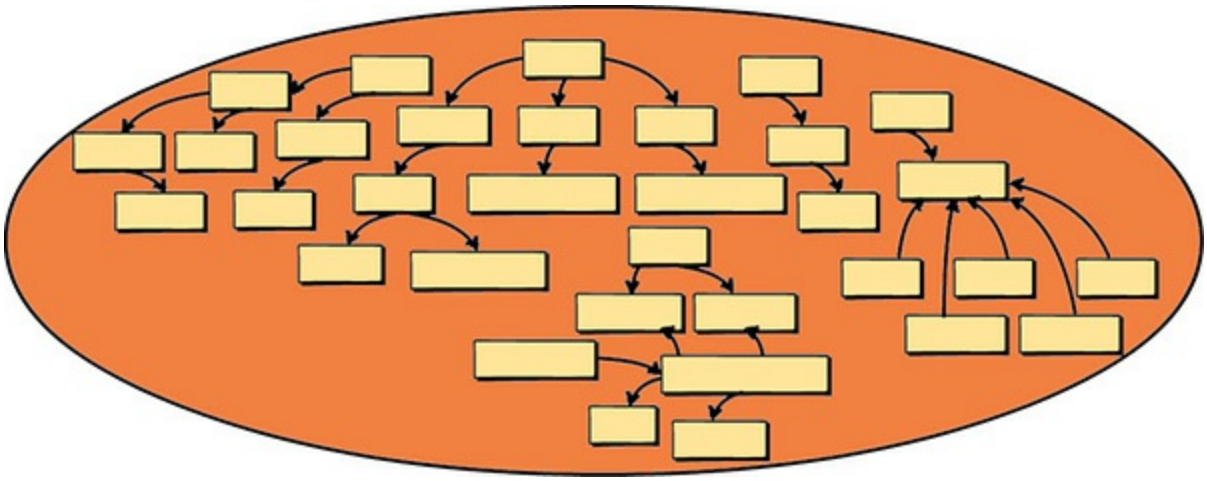
In human languages, terminology evolves over time, and across national boundaries the same or similar words take on nuances of meaning. Think of the differences between Spanish words used in Spain and those same words used in Colombia, where even the pronunciation changes. There is clearly Spain's Spanish and Colombia's Spanish. So too with software model languages. It's possible that people from other teams would have a different meaning for the same terminology, because their business knowledge is within a different context; they are developing a different *Bounded Context*. Any components outside the context are not expected to adhere to the same definitions. In fact, they are probably different, either slightly or vastly, from the components that your team models. That's fine.



To understand one big reason to use *Bounded Contexts* , let's consider a common problem with software designs. Often teams don't know when to stop piling more and more concepts into their domain models. The model may start out small and manageable . . .



But then the team adds more concepts, and more, and still more. This soon results in a big problem. Not only are there too many concepts, but the language of the model becomes blurred, because when you think about it there are actually multiple languages in one large, confusing, unbounded model.

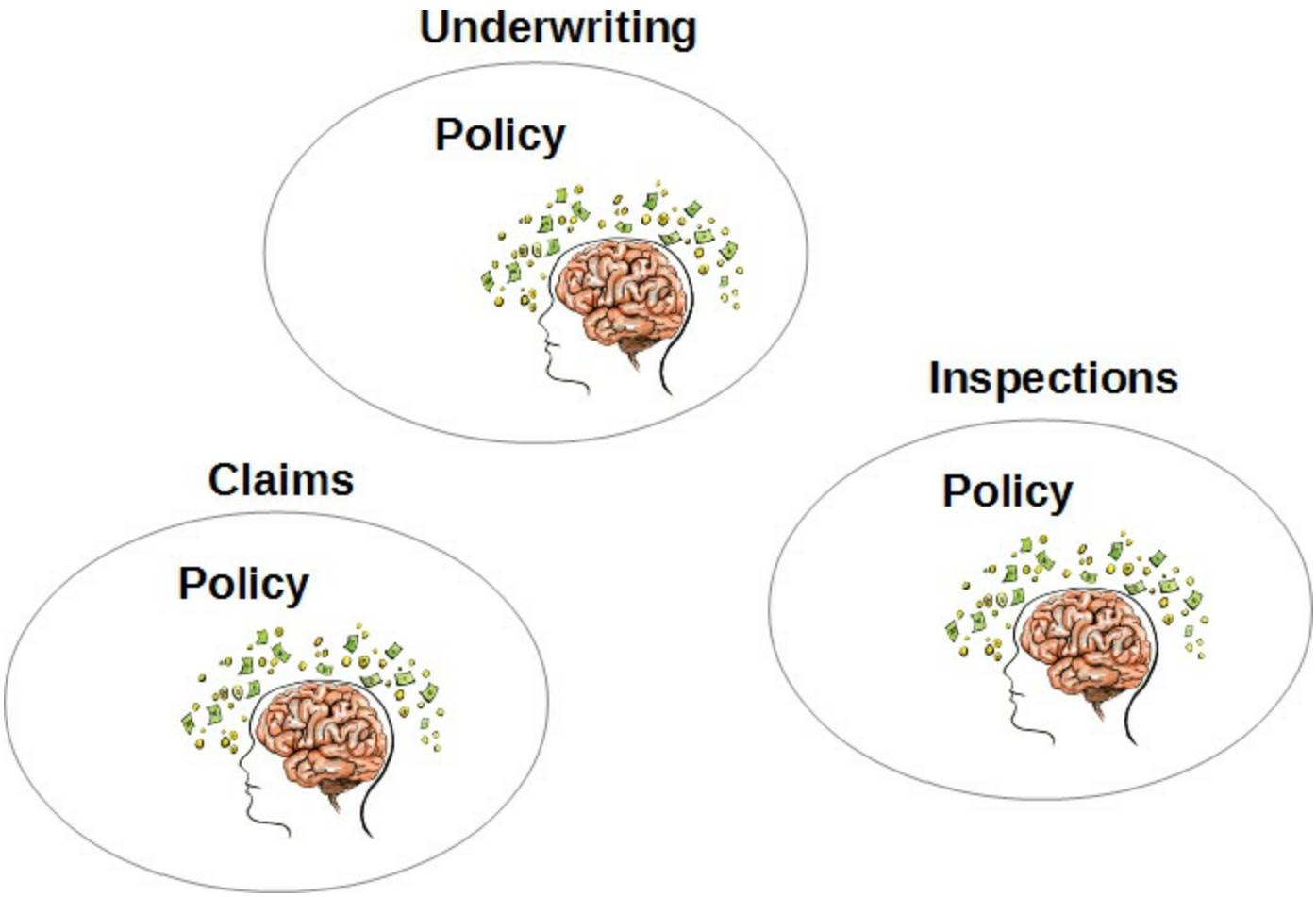


Due to this fault, teams will often turn a brand-new software product into what is called a *Big Ball of Mud*. To be sure, a *Big Ball of Mud* is not something to be proud of. It's a monolith, and worse. This is where a system has multiple tangled models without explicit boundaries. It probably also requires multiple teams to work on it, which is very problematic. Furthermore, various unrelated concepts are blown out over many modules and interconnected with conflicting elements. If this project has tests, it probably takes a very long time to run them, and so the tests may be bypassed at especially important times.

It's the product of trying to do too much, with too many people, in the wrong place. Any attempt to develop and speak a *Ubiquitous Language* will result in a fractured and ill-defined dialect that will soon be abandoned. The language wouldn't even be as well conceived as Esperanto. It's just a mess, like a *Big Ball of Mud*.

Domain Experts and Business Drivers

There may be strong, or at least subtle, hints communicated by business stakeholders that could have been used to help the technical team make better modeling choices. Thus, a *Big Ball of Mud* is often the result of unbridled effort made by a team of software developers who don't listen to the business experts.



The business's department or work group divisions can provide a good indication of where model boundaries should exist. You will tend to find at least one business expert per business function. Lately there is a trend toward grouping people by project, while business divisions or even functional groups under a management hierarchy seem to be less popular. Even in the face of newer business models you will still find that projects are organized according to business drivers and under an area

of expertise. You may need to think of division or function in those terms.

You can determine that this kind of segregation is needed when you consider that each business function likely has different definitions for the same term. Consider the concept named "policy" and how the meaning differs across the various insurance business functions. You can easily imagine that a policy in underwriting is vastly different from a policy in claims and a policy in inspections. See the box for more details.

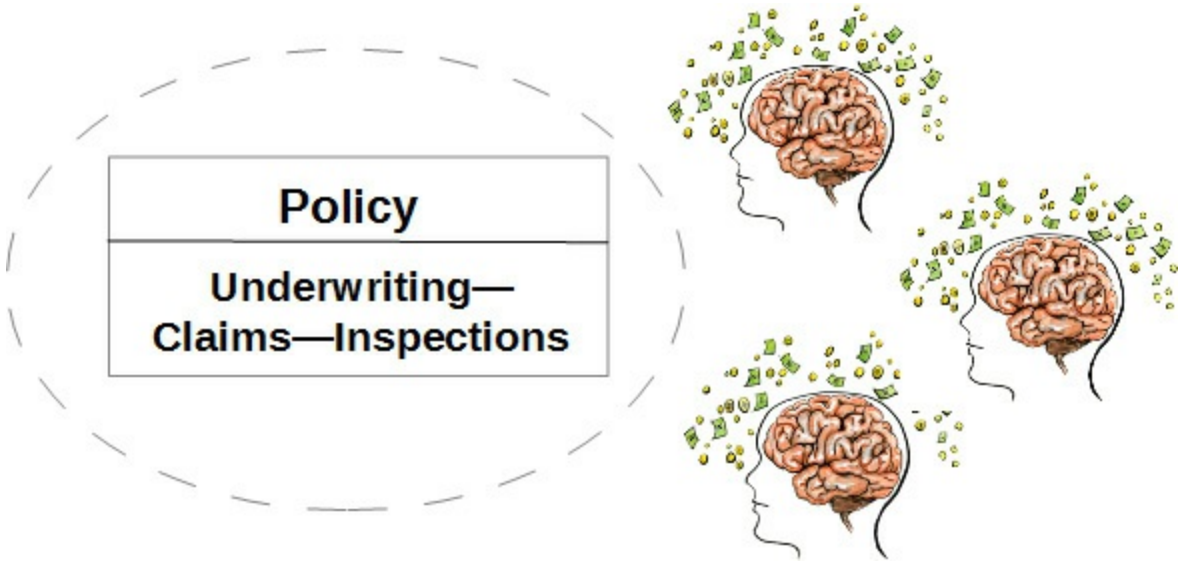
The policy in each of these business areas exists for different reasons. There is no escaping this fact, and no amount of mental gymnastics changes this.

Differences in Policies by Function

Policy in Underwriting: In the area of expertise that is focused on underwriting, a policy is created based on the evaluation of the risks of the insured entity. For example, when working in underwriting for property insurance, the underwriters would assess the risks associated with a given property in order to calculate the premium for the policy to cover the property asset.

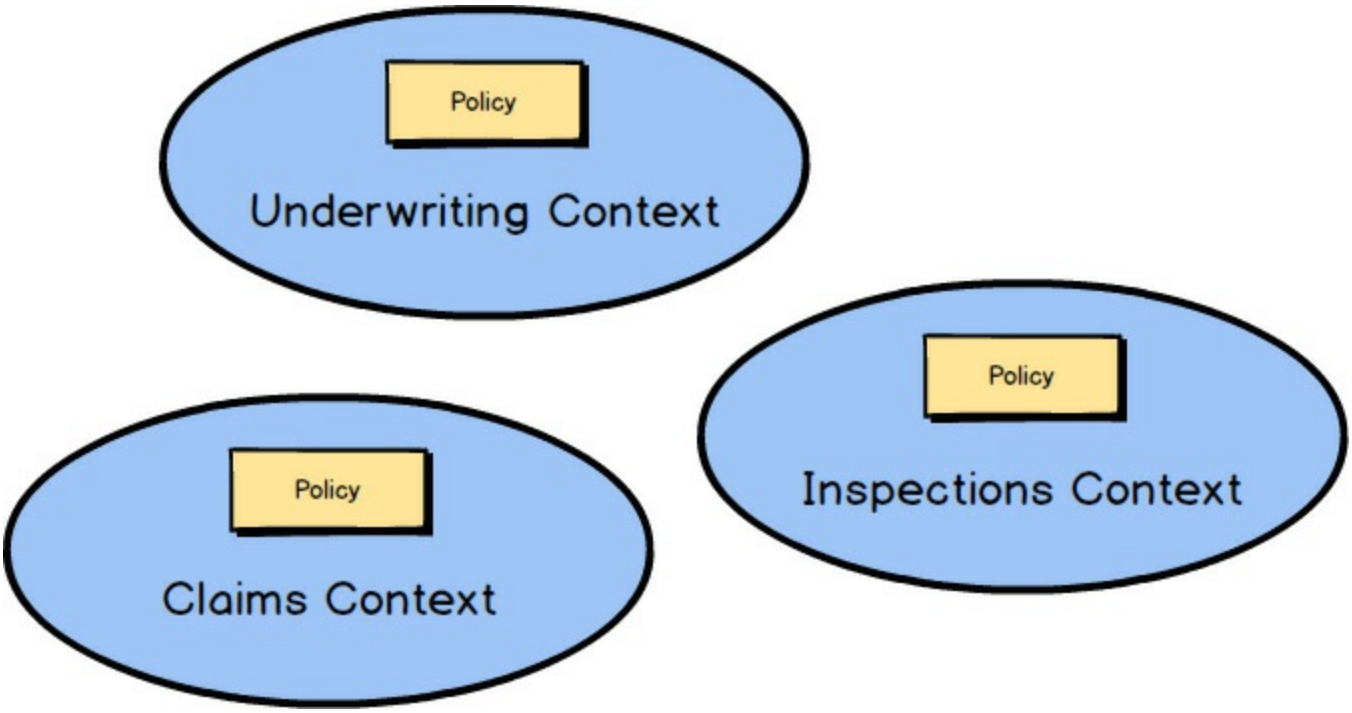
Policy in Inspections: Again, if we are working in the property insurance field, the insurance organization will likely have an inspections area of expertise that is responsible for inspecting a property that is to be insured. The underwriters are somewhat dependent on the information found during inspections, but only from the standpoint that the property is in the condition asserted by the insured. Assuming that a property will be insured, the inspection details—photos and notes—are associated with a policy in the inspections area, and its data can be referenced by underwriting to negotiate the final premium cost in the underwriting area.

Policy in Claims: A policy in the claims area of expertise tracks the request for payment by the insured based on the terms of the policy created by the underwriting area. The claims policy will need to make some references to the underwriting policy but will be focused on, for example, damages to the insured property and reviews performed by the claims personnel to determine the payment, if any, that should be made.



If you try to merge all three of these policy types into a single policy for all three business groups, you will certainly have problems. This would become even more problematic if the already-

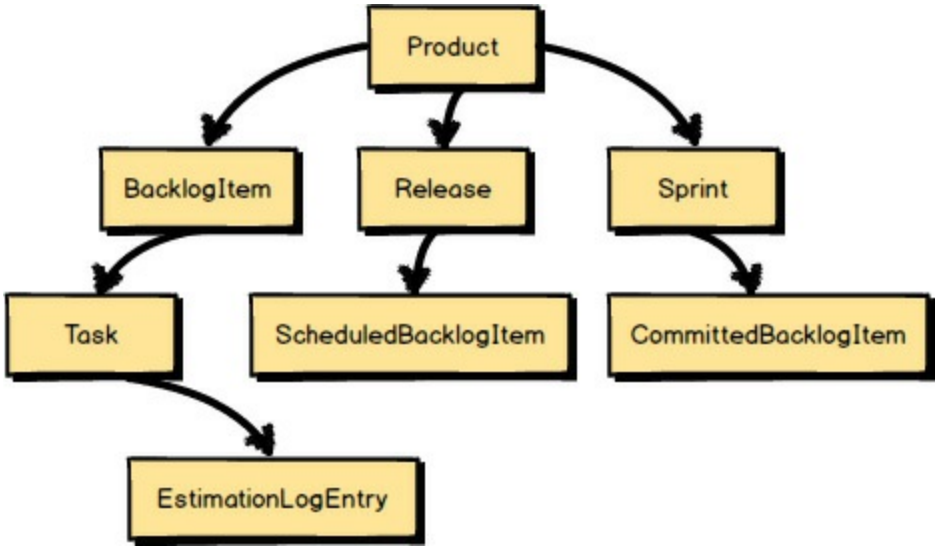
overloaded policy had to support a fourth and fifth business concept in the future. Nobody wins.



On the other hand, DDD emphasizes embracing such differences by segregating the differing types into different *Bounded Contexts*. Admit that there are different languages, and function accordingly. Are there three meanings for policy? Then there are three *Bounded Contexts* , each with its own policy, with each policy having its own unique qualities. There's no need to name these `UnderwritingPolicy` , `ClaimsPolicy` , or `InspectionsPolicy` . The name of the *Bounded Context* takes care of that scoping. The name is simply `Policy` in all three *Bounded Contexts*.

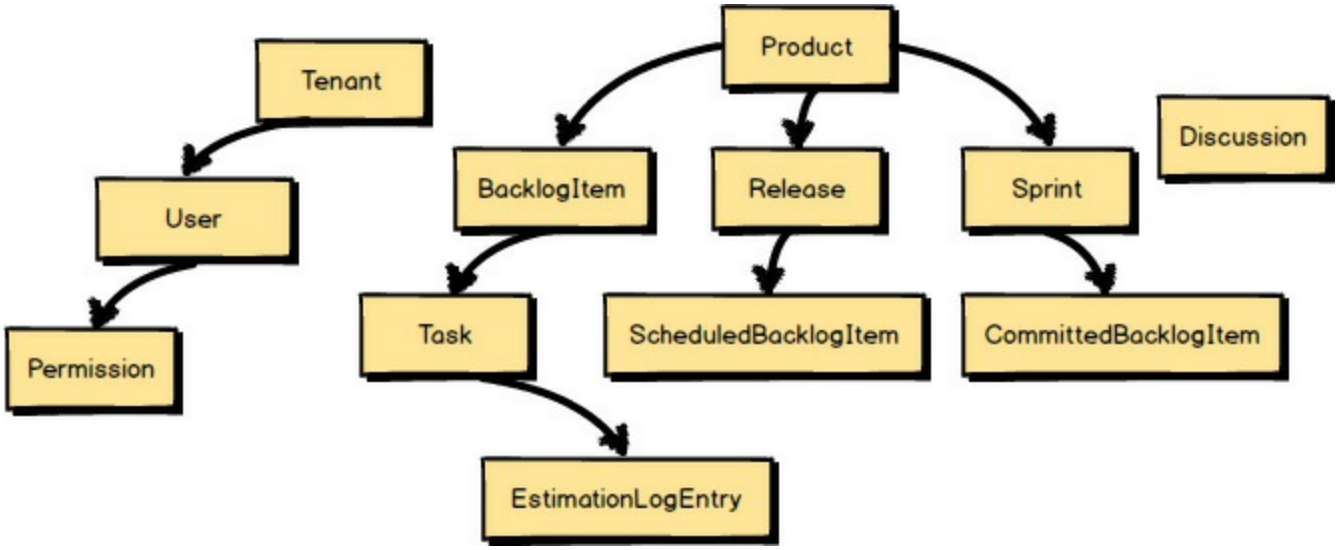
Another Example: What Is a Flight?

In the airline industry, a "flight" can have multiple meanings. There is a flight that is defined as a single takeoff and landing, where the aircraft is flown from one airport to another. There is a different kind of flight that is defined in terms of aircraft maintenance. And there is yet another flight that is defined in terms of passenger ticketing, either nonstop or one-stop. Because each of these uses of "flight" is clearly understood only by its context, each should be modeled in a separate *Bounded Context*. To model all three of these in the same *Bounded Context* would lead to a confusing tangle.

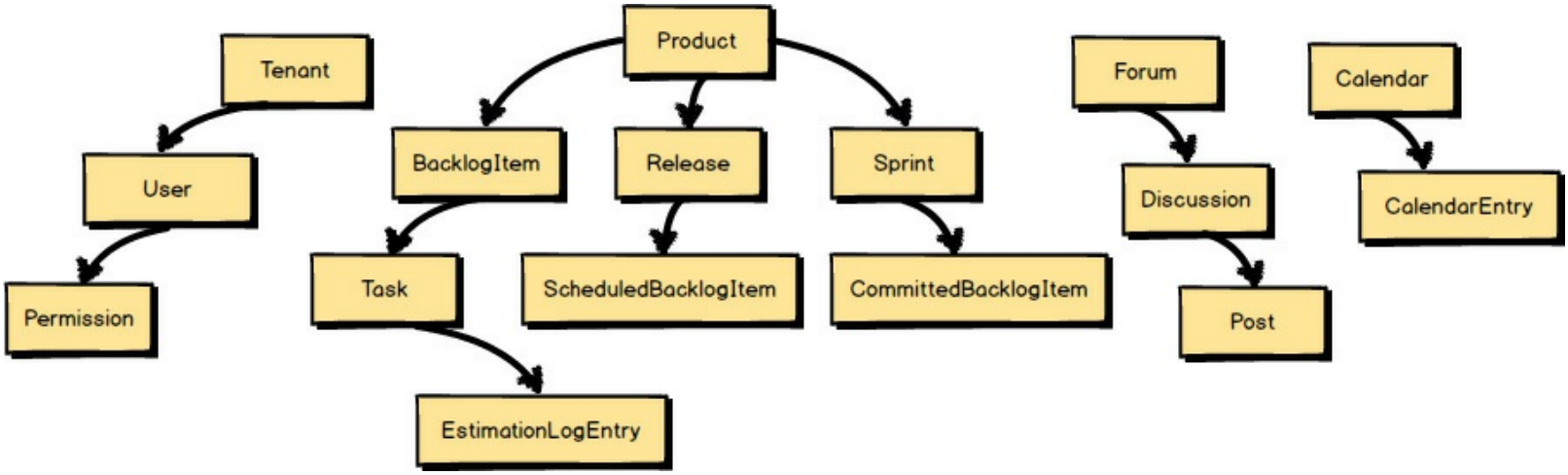


Case Study

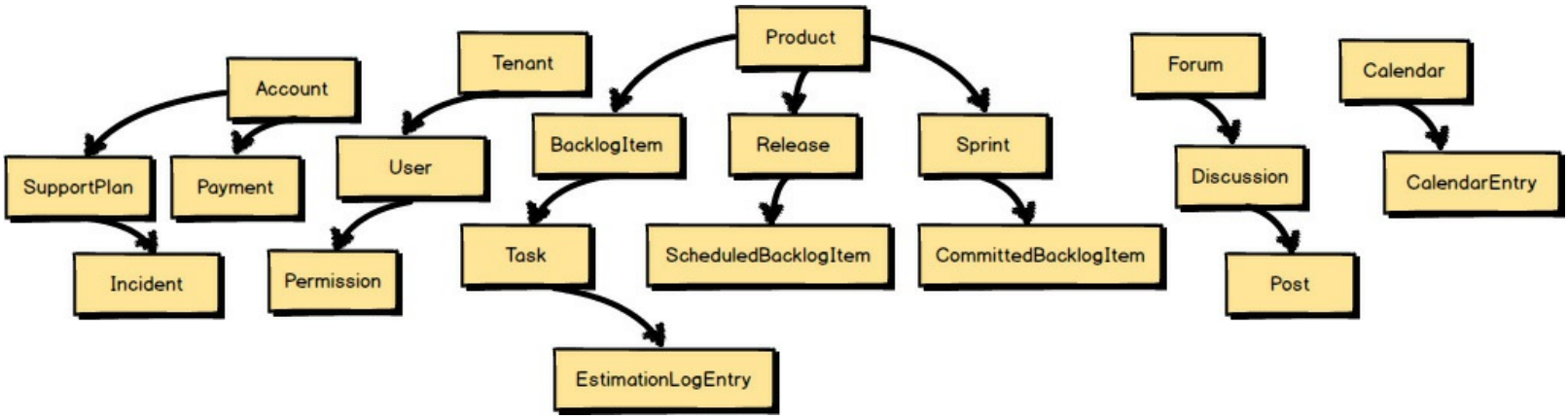
To make the reason to use *Bounded Contexts* more concrete, let me illustrate with a sample domain model. In this case we are working on a Scrum-based agile project management application. So, a central or core concept is Product, which represents the software that is to be built and that will be refined over perhaps years of development. The Product has Backlog Items, Releases, and Sprints. Each Backlog Item has a number of Tasks, and each Task can have a collection of Estimation Log Entries. Releases have Scheduled Backlog Items and Sprints have Committed Backlog Items. So far, so good. We have identified the core concepts of our domain model, and the language is focused and intact.



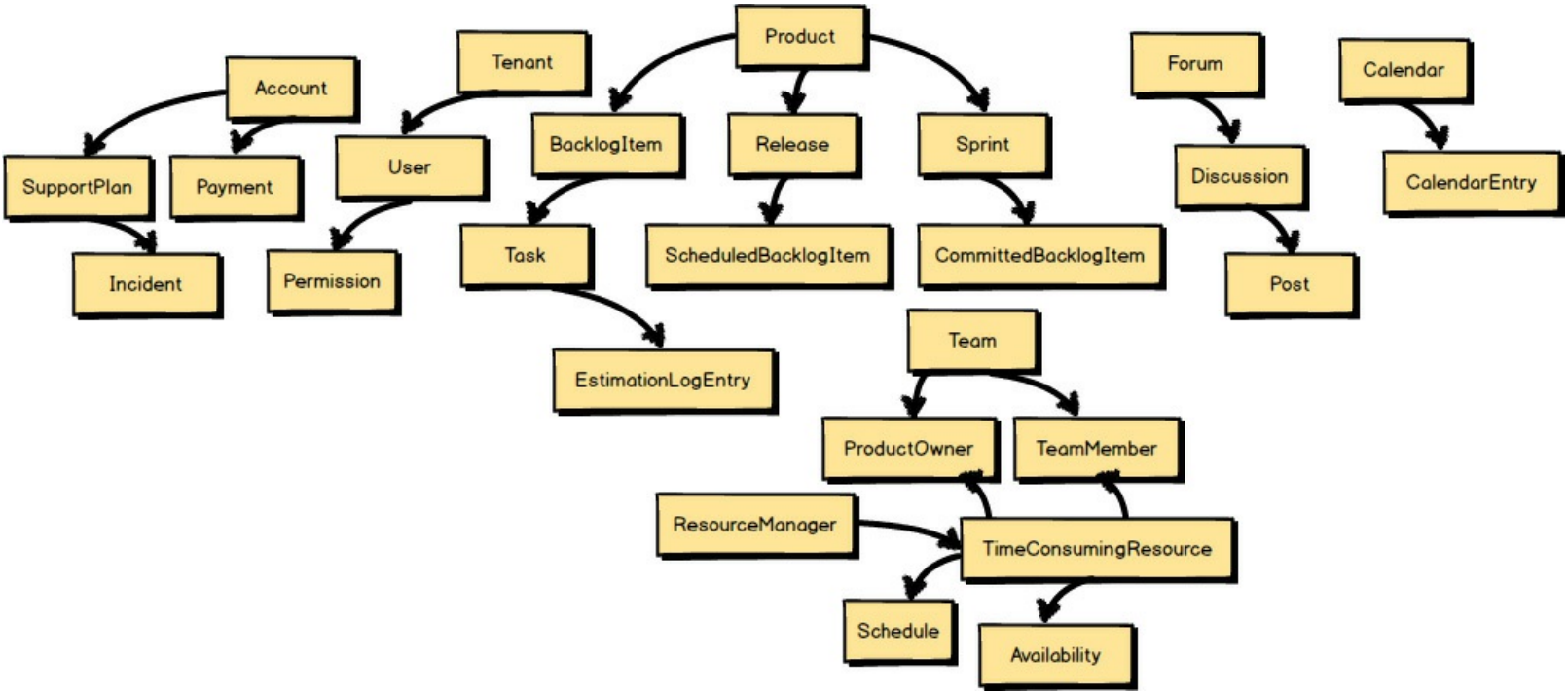
"Oh, yeah," say the team members, "we also need our users. And we want to facilitate collaborative discussions within the product team. Let's represent each subscribing organization as a Tenant. Within Tenants we will allow the registration of any number of Users, and Users will have Permissions. And let's add a concept called Discussion to represent one of the collaborative tools that we will support."



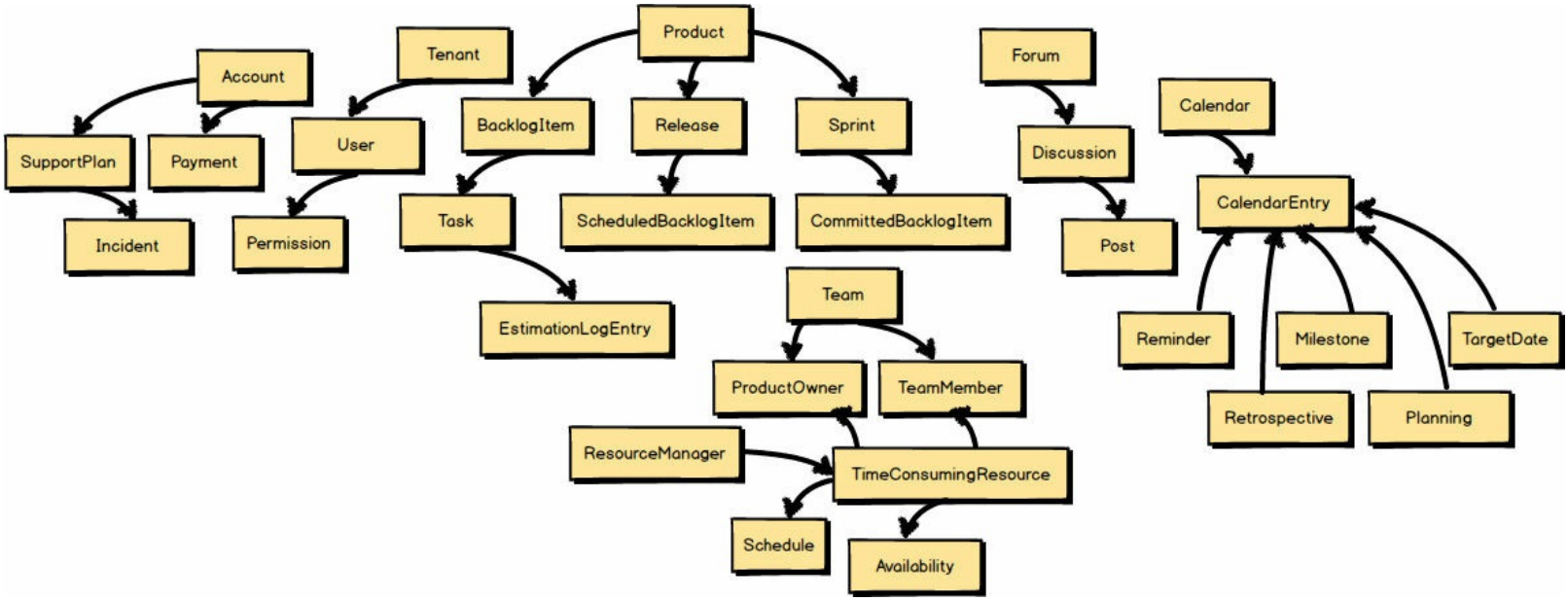
Then the team members add, "Well, there are also other collaboration tools. Discussions belong within Forums and Discussions have Posts. Also we want to support Shared Calendars."



They continue: "And don't forget that we need a way for Tenants to make Payments. We will also sell tiered support plans, so we need a way to track support incidences. Both Support and Payments should be managed under an Account."

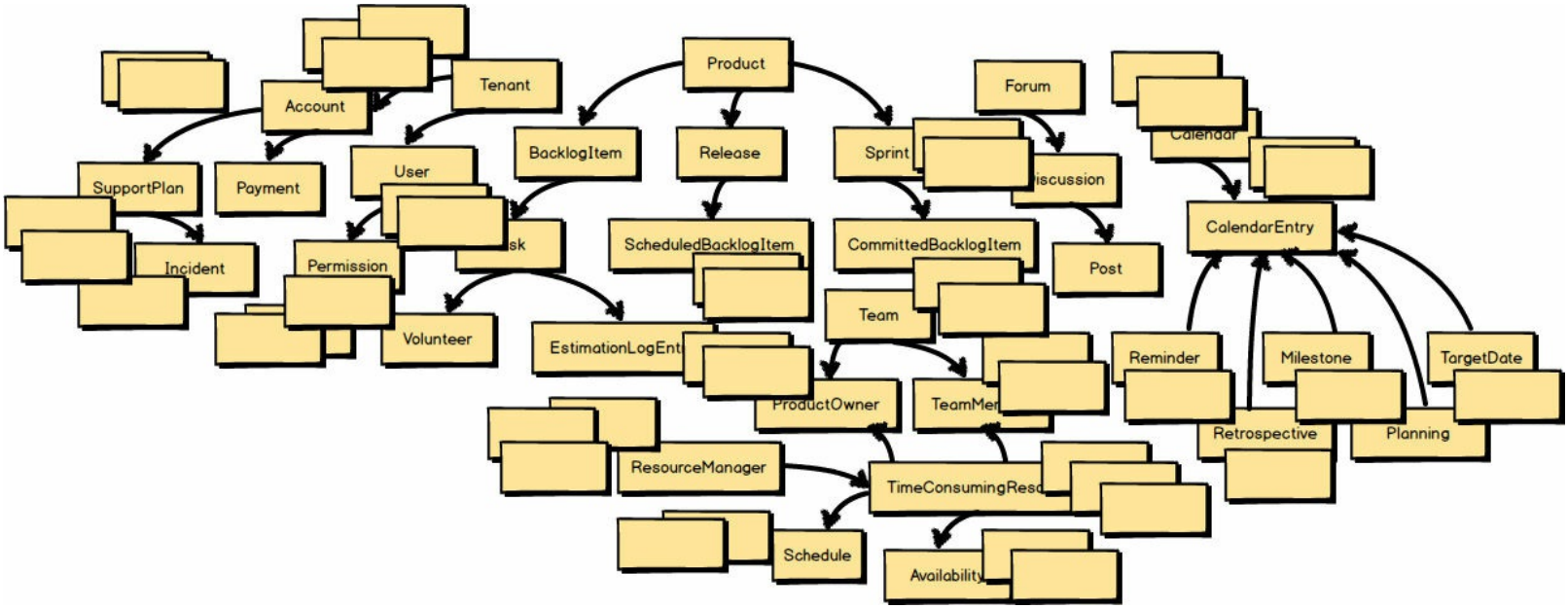


And still more concepts emerge: "Every Scrum-based Product has a specific Team that works on the product. Teams are composed of a single Product Owner and a number of Team Members. But how can we address Human Resource Utilization concerns? Hmmm, what if we modeled the Schedules of Team Members along with their utilization and availability?"

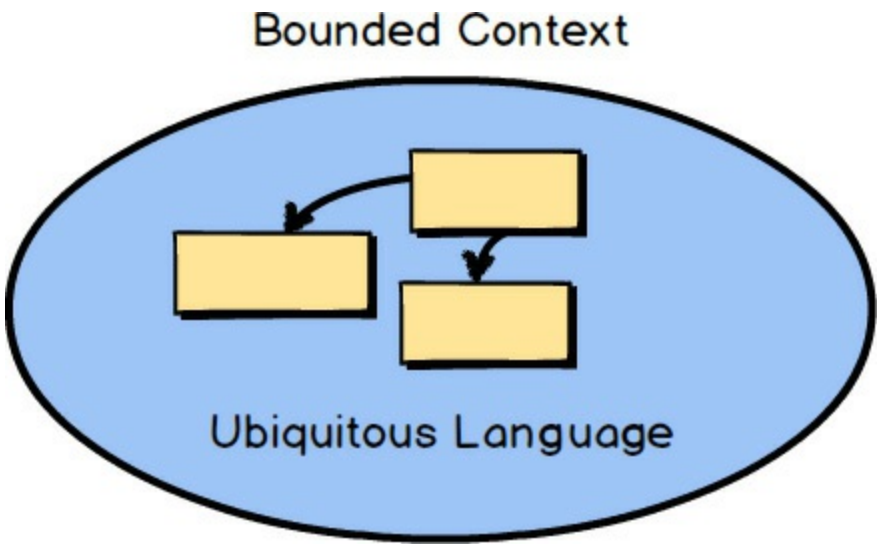


"You know what else?" they ask. "Shared Calendars should not be limited to bland Calendar Entries. We should be able to identify specific kinds of Calendar Entries, such as Reminders, Team Milestones, Planning and Retrospective Meetings, and Target Dates."

Hang on a minute! Do you see the trap that the team is falling into? Look at how far they have strayed from the original core concepts of Product, Backlog Items, Releases, and Sprints. The language is no longer purely about Scrum; it has become fractured and confused.



Don't be fooled by the somewhat limited number of named concepts. For every named element, we might expect to have two or three more concepts to support those that quickly popped into mind. The team is already well on its way to delivering a *Big Ball of Mud* and the project has barely started.

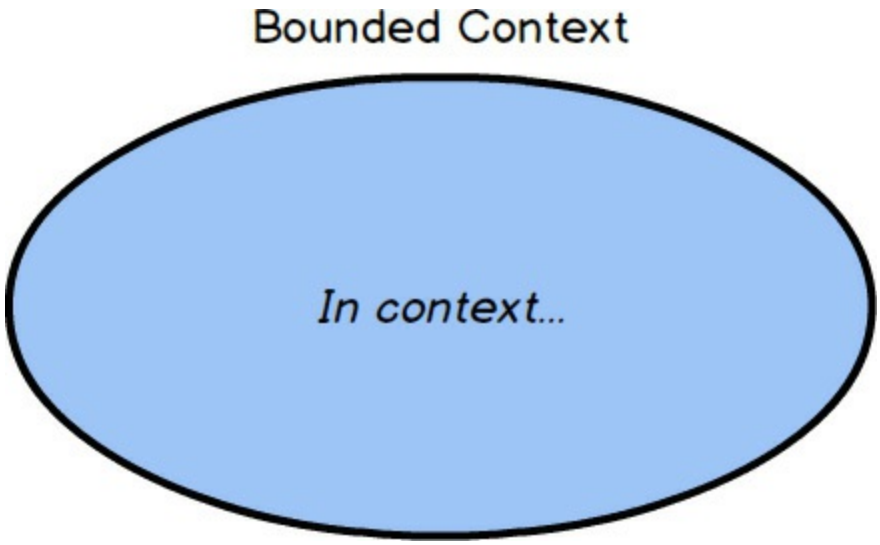


Fundamental Strategic Design Needed

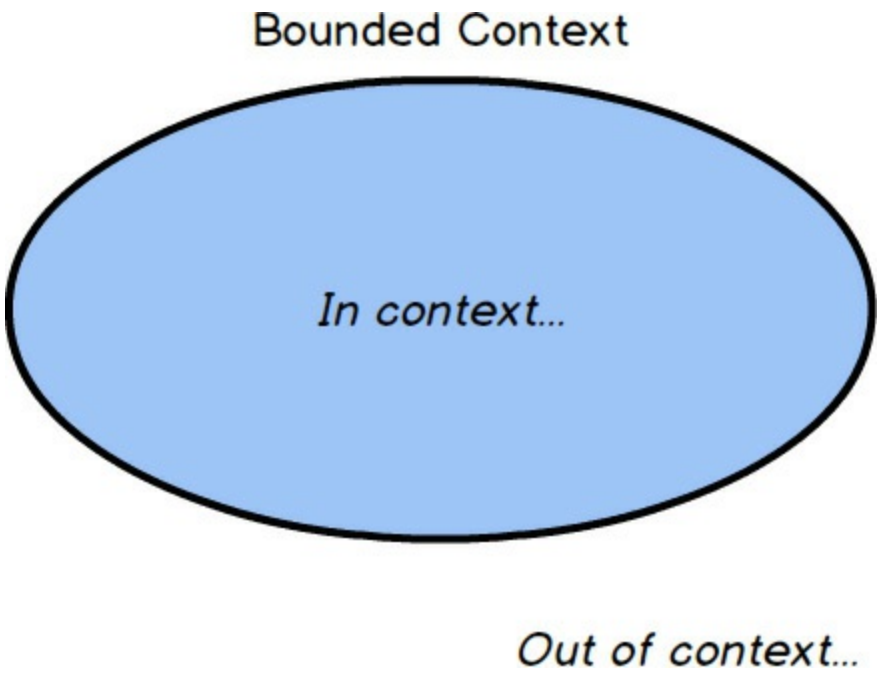
What tools are available with DDD to help us avoid such pitfalls? You need at least two fundamental strategic design tools. One is the *Bounded Context* and the other is the *Ubiquitous Language*. Employing a *Bounded Context* forces us to answer the question "What is core?" The *Bounded Context* should hold closely all concepts that are core to the strategic initiative and push out all others. The concepts that remain are part of the team's *Ubiquitous Language*. You will see how DDD works to avoid the design of monolithic applications.

Testing Benefits

Because *Bounded Contexts* are not monolithic, other benefits are experienced when they are used. One such benefit is that tests will be focused on one model and thus be fewer in number and will run more quickly. Although this isn't the primary motivation to use *Bounded Contexts* , it sure pays off in other ways.



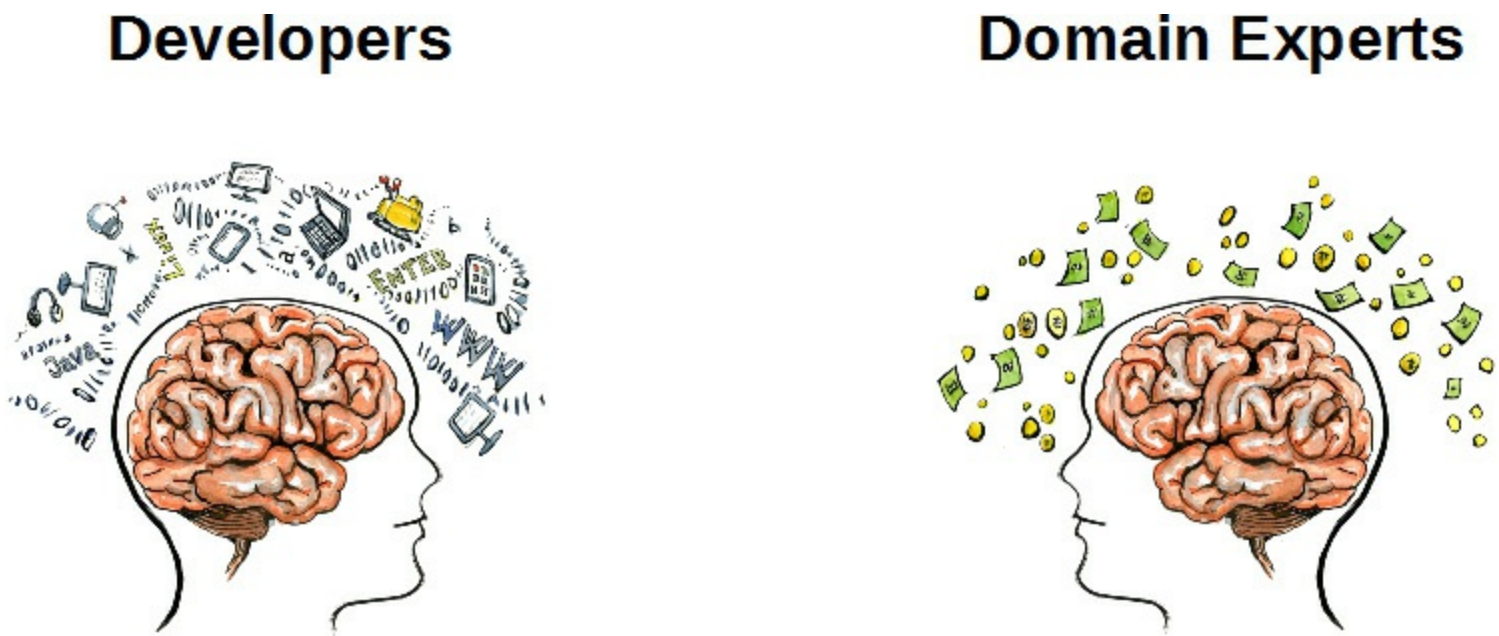
Literally, some concepts will be in context and be clearly included in the team's language.



And other concepts will be out of context. The concepts that survive this stringent application of core-only filtering are part of the *Ubiquitous Language* of the team that owns the *Bounded Context*.

Take Note

The concepts that survive this stringent application of core-only filtering are part of the *Ubiquitous Language* of the team that owns the *Bounded Context*. The boundary emphasizes the rigor inside.



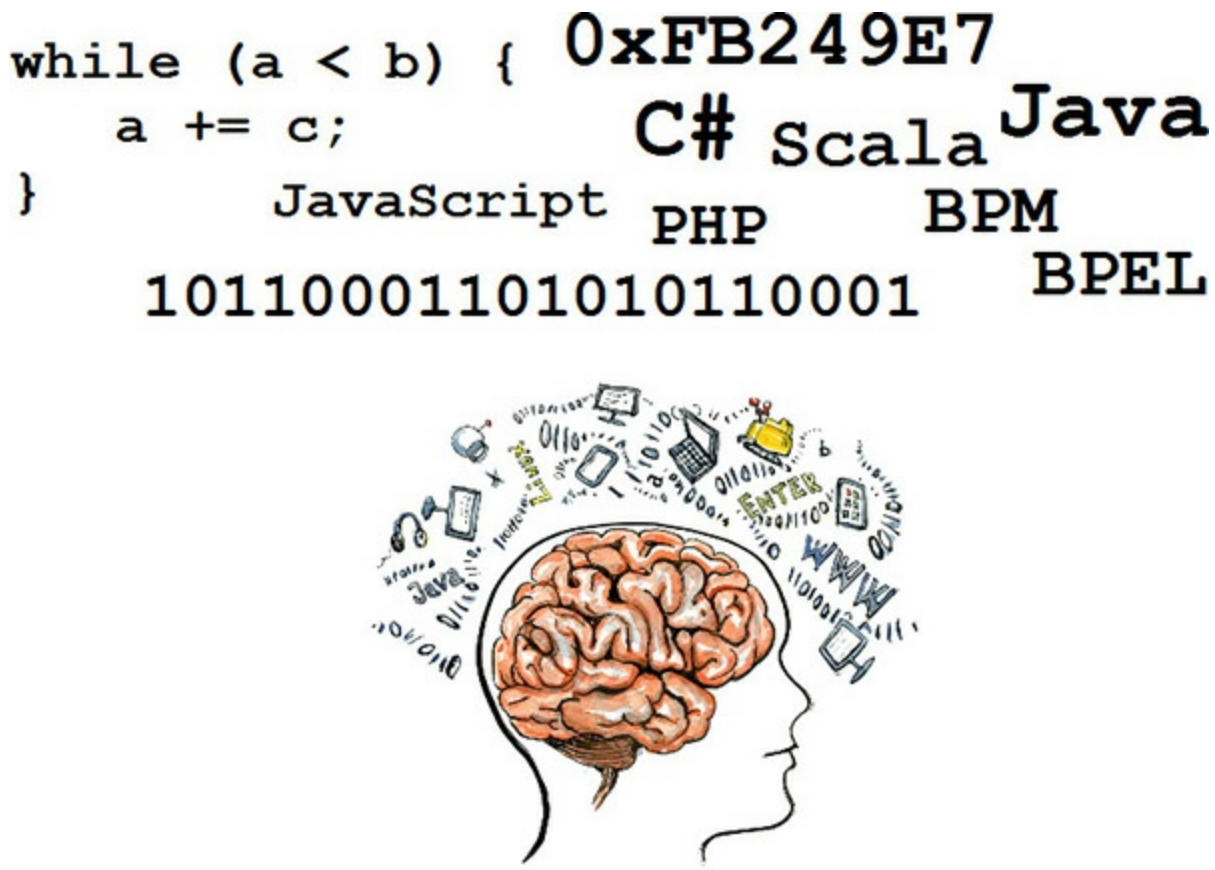
So, how do we know what is core? This is where we have to bring together two vital groups of individuals into one cohesive, collaborative team: *Domain Experts* and software developers.

The *Domain Experts* will naturally be more focused on business concerns. Their thoughts will be centered on their vision of how the business works. In the domain of Scrum, count on the *Domain Expert* being a Scrum Master who thoroughly understands how Scrum is executed on a project.

Product Owner or Domain Expert?

You may wonder what the difference is between a Scrum product owner and a DDD *Domain Expert*. Well, in some cases they might be one and the same, that is, one person capable of filling both roles. Yet it should not be surprising that a product owner is typically more focused on managing and prioritizing the product backlog and seeing to it that the conceptual and technical continuity of the project is maintained. This doesn't mean, however, that the product owner is naturally an expert in the business's core competency in which you are working. Make sure that you have a true *Domain Expert* on the team, and don't substitute a product owner without the necessary know-how instead.

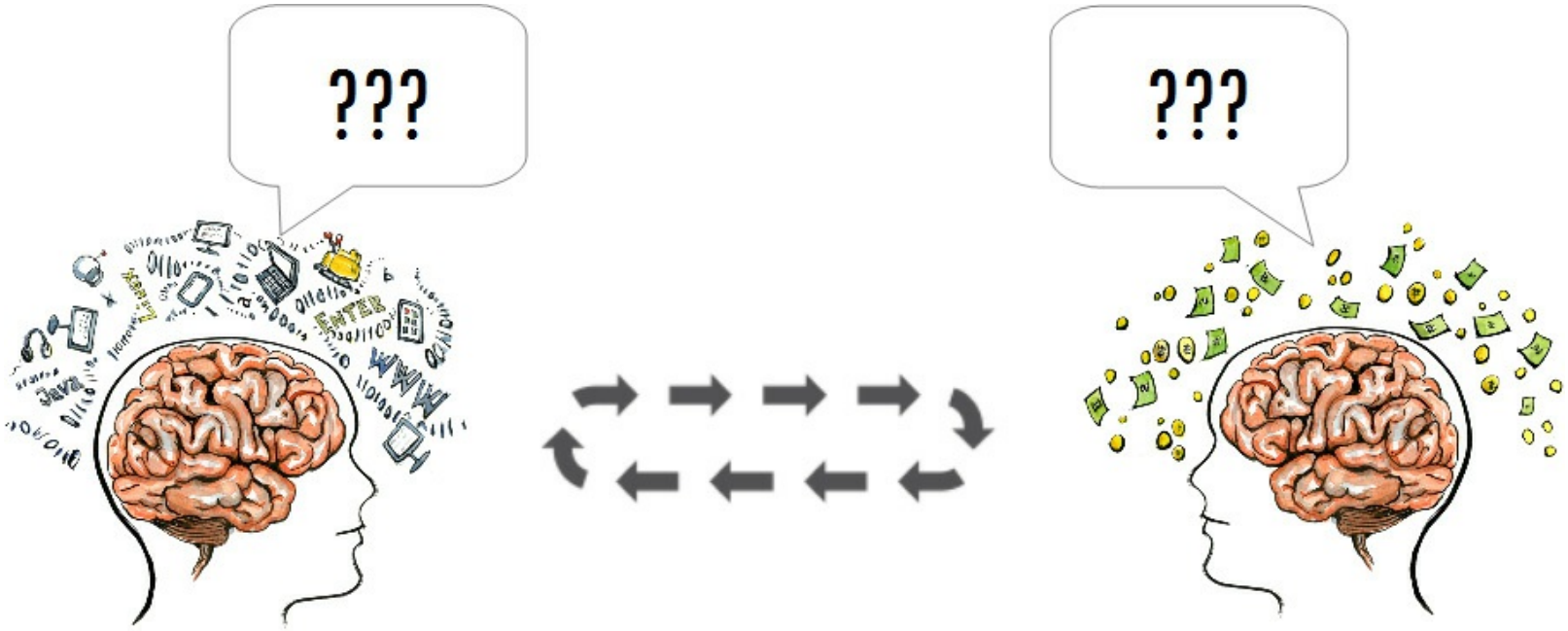
In your particular business, you also have *Domain Experts*. It's not a job title but rather describes those who are primarily focused on the business. It's their mental model that we start with to form the foundation of the team's *Ubiquitous Language*.



On the other hand, developers are focused on software development. As depicted here, developers can become consumed by programming languages and technologies. Yet developers working in a DDD project need to carefully resist the urge to be so technically centered that they cannot accept the business focus of the core strategic initiative. Rather, the developers should reject any uncalled-for terseness and be able to embrace the *Ubiquitous Language* that is gradually developed by the team inside their particular *Bounded Context*.

Focus on Business Complexity, Not Technical Complexity

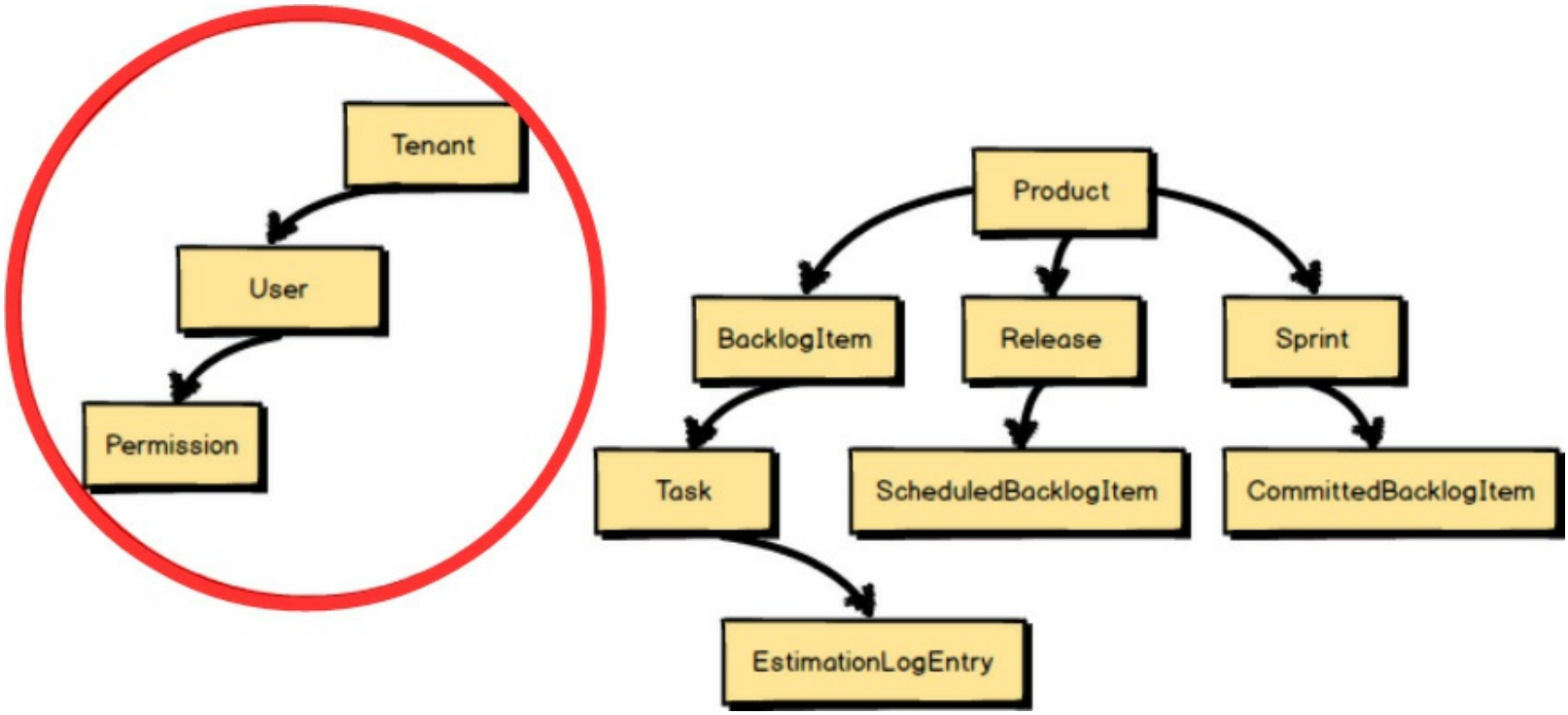
You are using DDD because the business model complexity is high. We never want to make the domain model more complex than it should be. Still, you are using DDD because the business model is more complex than the technical aspects of the project. That's why the developers have to dig into the business model with *Domain Experts* !



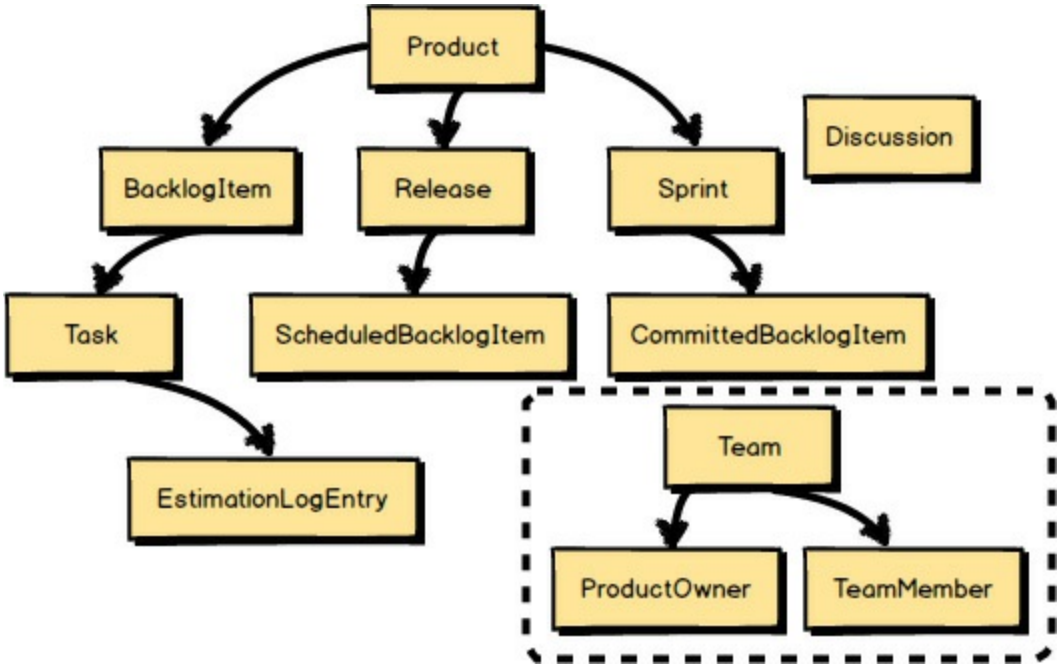
Both developers and *Domain Experts* should reject any tendency to allow documents to rule over conversation. The best *Ubiquitous Language* will be developed by a collaborative feedback loop that drives out the combined mental model of the team. Open conversation, exploration, and challenges to your current knowledge base result in deeper insights about the *Core Domain*.

Challenge and Unify

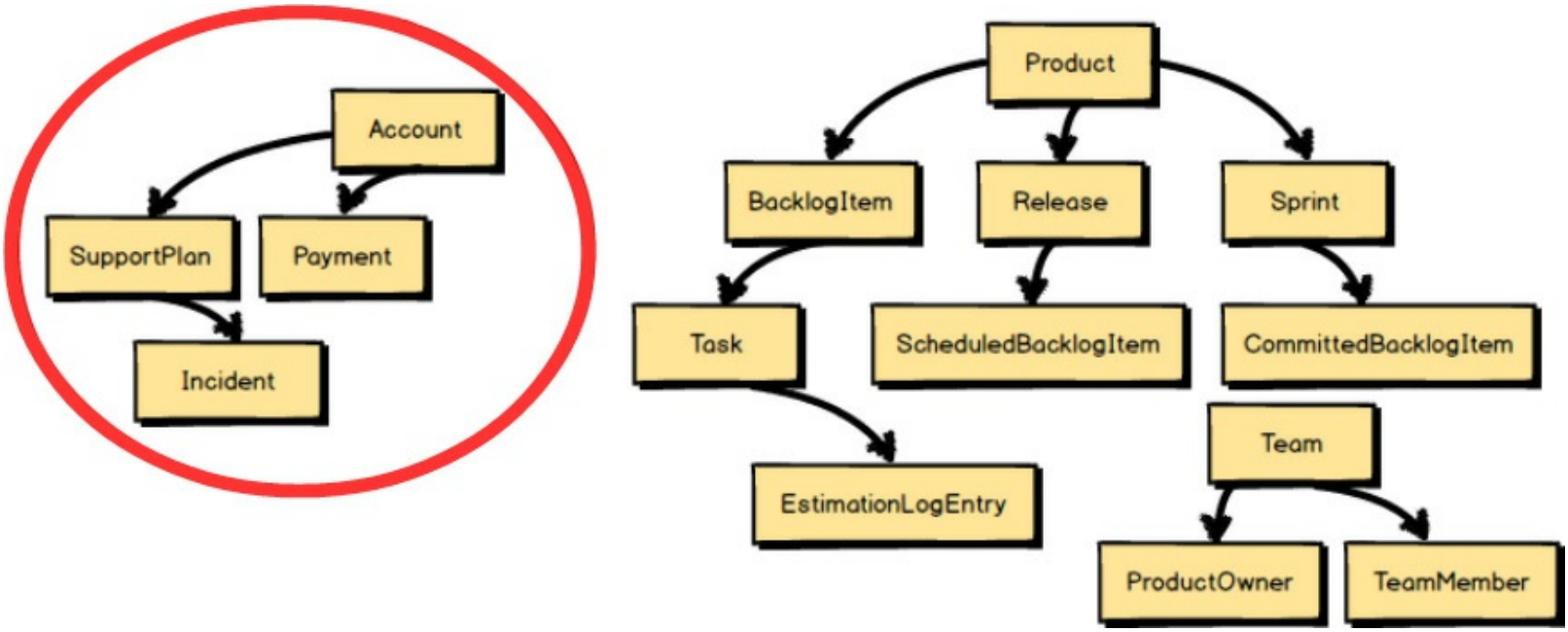
Now back to the question "What is core?" Using the previously out-of-control and ever-expanding model, let's challenge and unify!



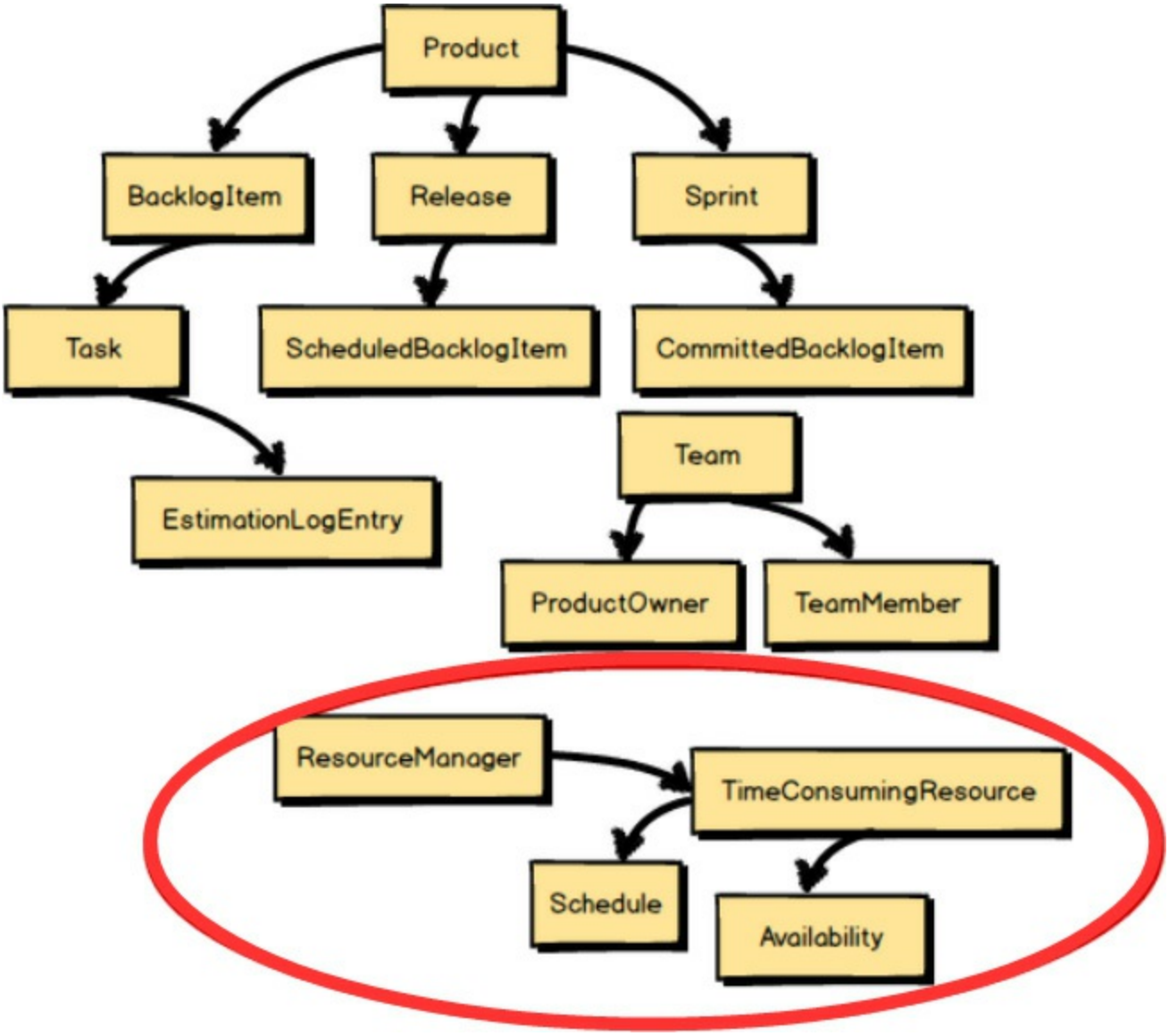
One very simple challenge is to ask whether each of the large-model concepts adheres to the *Ubiquitous Language* of Scrum. Well, do they? For example, `Tenant` , `User` , and `Permission` have nothing to do with Scrum. These concepts should be factored out of our Scrum software model.



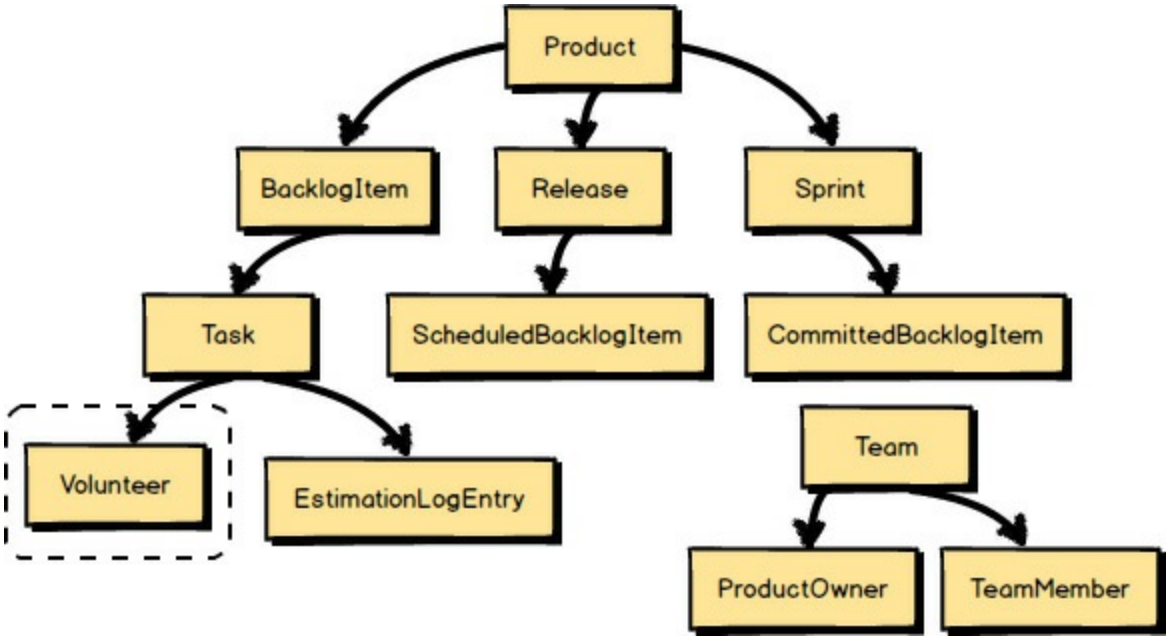
`Tenant` , `User` , and `Permission` should be replaced by `Team` , `ProductOwner` , and `TeamMember` . A `ProductOwner` and a `TeamMember` are actually `Users` in a `Tenancy` , but with `ProductOwner` and `TeamMember` we adhere to the *Ubiquitous Language* of Scrum. They are naturally the terms we use when we have conversations about Scrum products and the work a team does with them.



Are `SupportPlans` and `Payments` really part of Scrum project management? The answer here is clearly "no." True, both `SupportPlans` and `Payments` will be managed under a `Tenant` 's `Account` , but these are not part of our core Scrum language. They are out of context and are removed from this model.

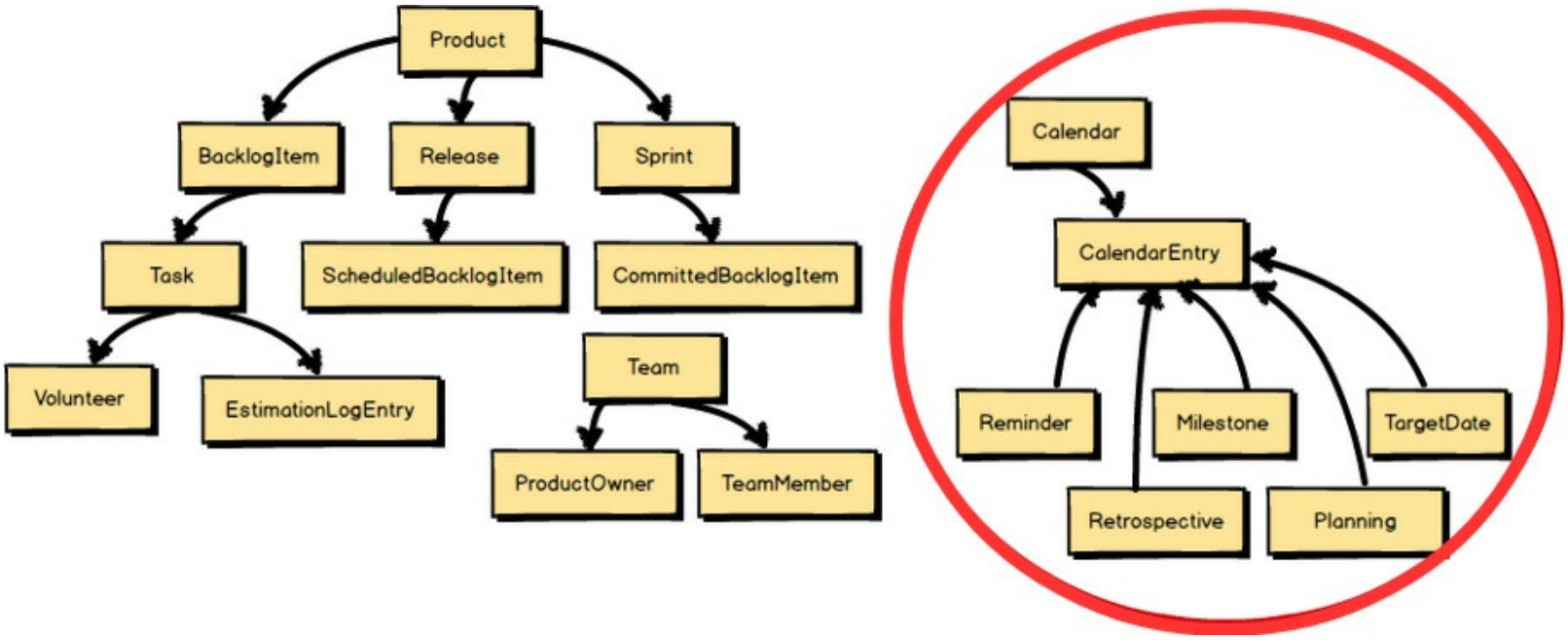


What about introducing Human Resource Utilization concerns? It's probably useful to someone, but it's not going to be directly used by `TeamMember Volunteers` who will work on `BacklogItemTasks` . It's out of context.

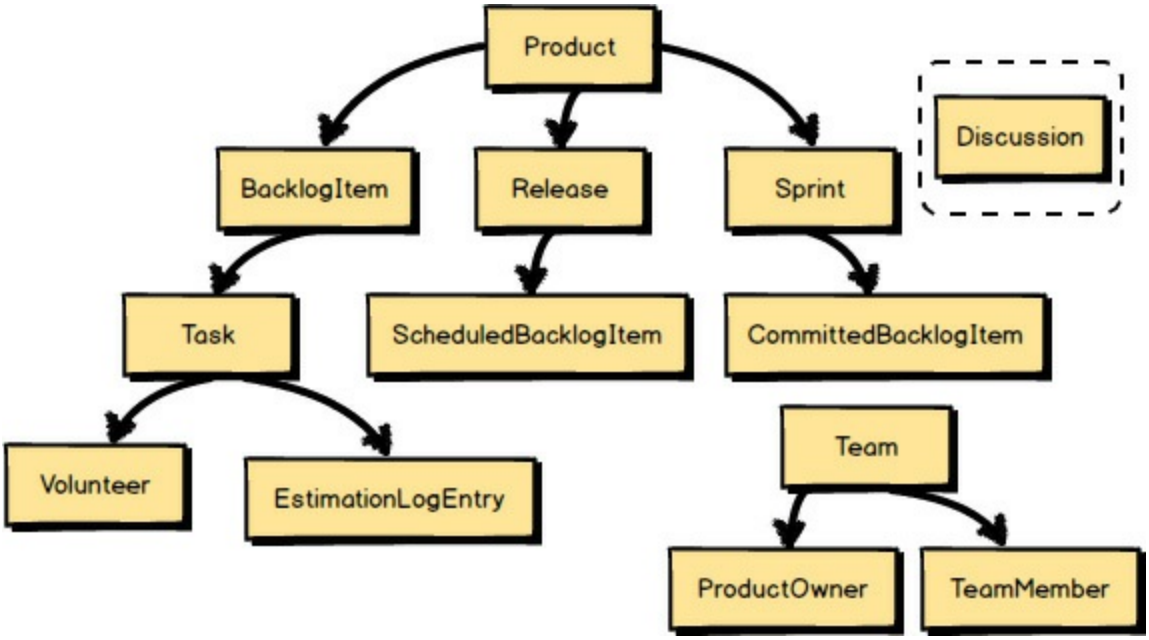


After the addition of the `Team` , `ProductOwner` , and `TeamMember` , the modelers realized that they were missing a core concept to allow `TeamMembers` to work on `Tasks` . In Scrum this is

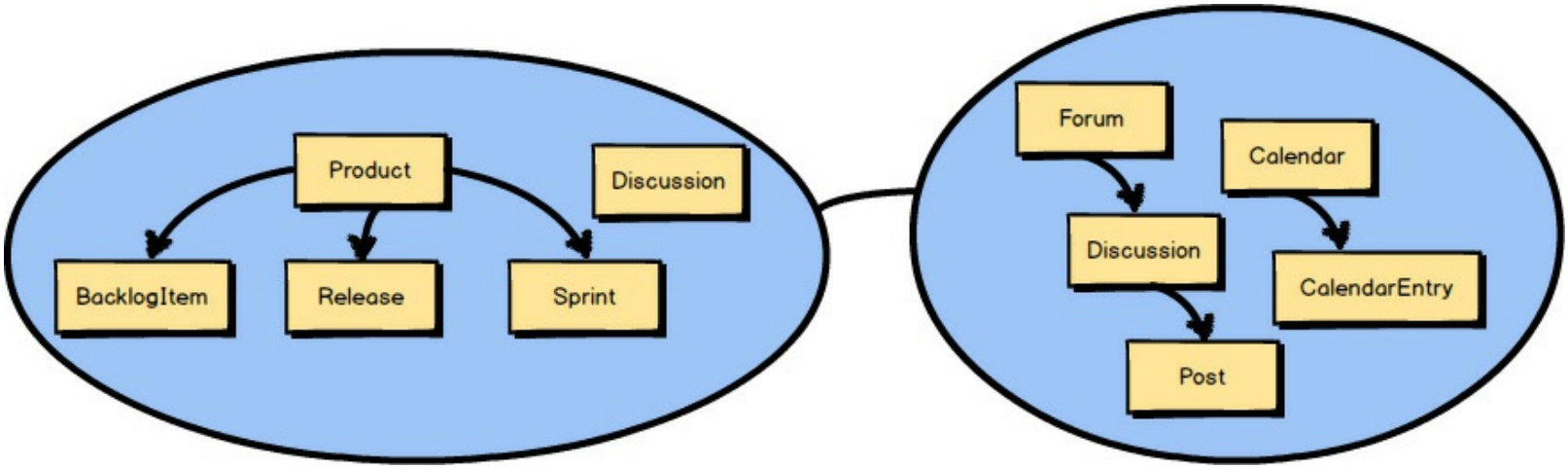
known as a `Volunteer` . So, the `Volunteer` concept is in context and was included in the language of the core model.



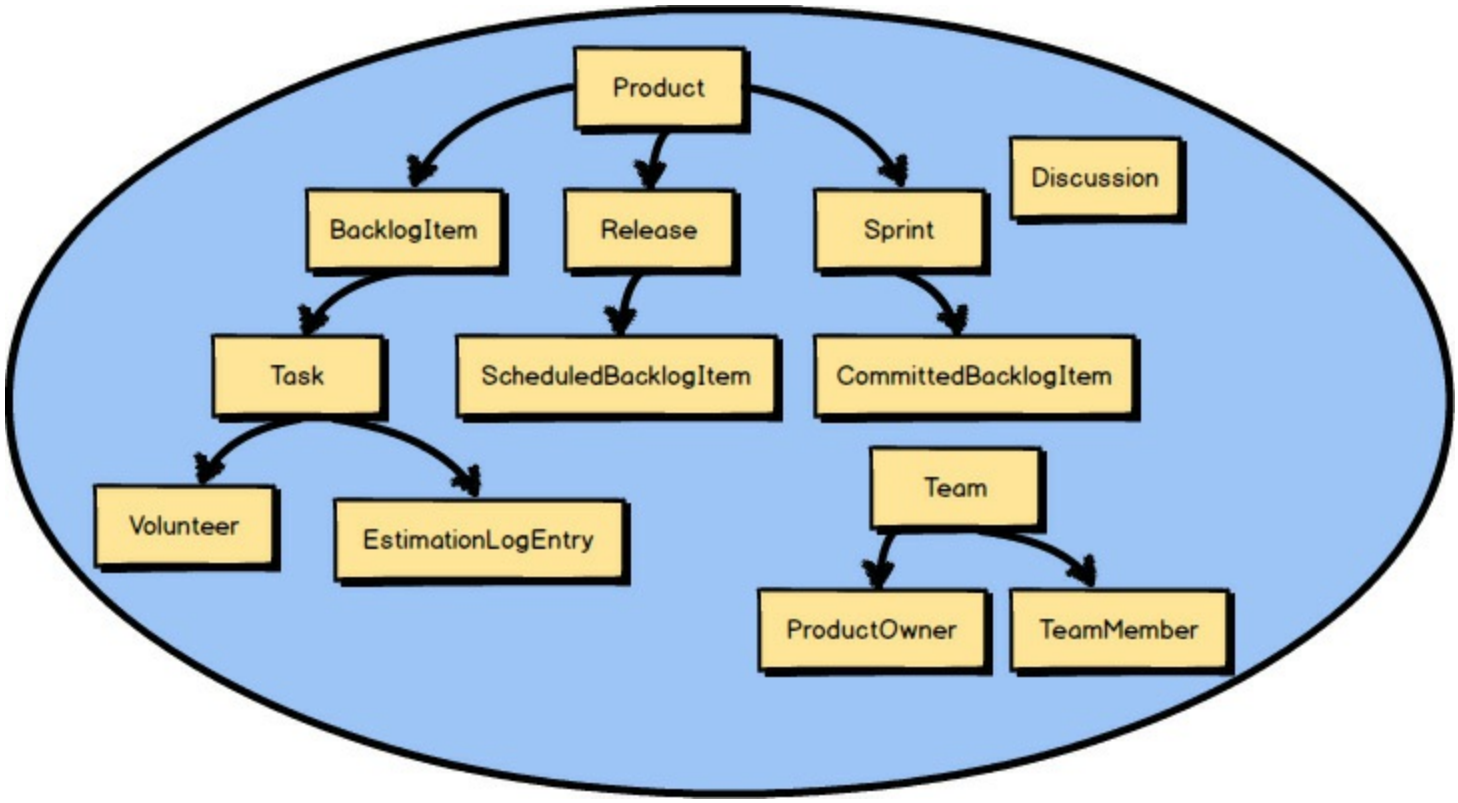
Even though calendar-based `Milestones` , `Retrospectives` , and the like are in context, the team would prefer to save those modeling efforts for a later sprint. They are in context, but for now they are out of scope.



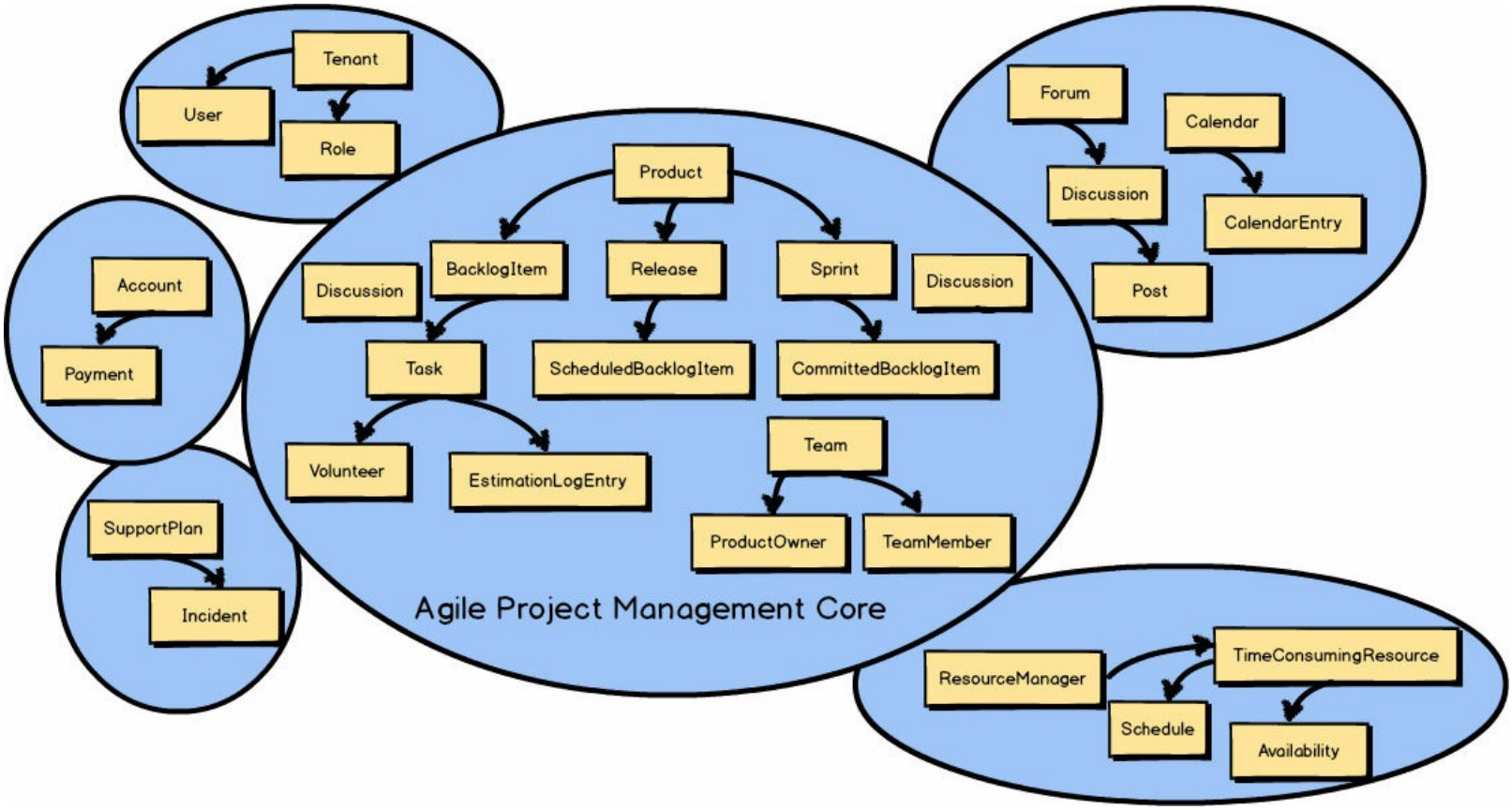
Finally, the modelers want to make sure that they account for the fact that threaded `Discussions` will be part of the core model. So they model a `Discussion` . This means that `Discussion` is part of the team's *Ubiquitous Language* , and thus inside the *Bounded Context*.



These linguistic challenges have resulted in a much cleaner and clearer model of the *Ubiquitous Language*. Yet how will the Scrum model fulfill needed *Discussions* ? It would certainly require a lot of ancillary software component support to make it work, so it seems inappropriate to model it inside our Scrum *Bounded Context*. In fact, the full *Collaboration* suite is out of context. The *Discussion* will be supported by integrating with another *Bounded Context* —the *Collaboration Context*.



After that walk-through, we're left with a much smaller actual *Core Domain*. Of course the *Core Domain* will grow. We already know that `Planning` , `Retrospectives` , `Milestones` , and related calendar-based models must be developed in time. Still, the model will grow only as new concepts adhere to the *Ubiquitous Language* of Scrum.



And what about all the other modeling concepts that have been removed from the *Core Domain* ? It's quite possible that several of the other concepts, if not all, will be composed into their own respective *Bounded Contexts* , each adhering to its own *Ubiquitous Language*. Later you will see how we integrate with them using *Context Mapping*.

Developing a Ubiquitous Language

So how do you actually go about developing a *Ubiquitous Language* within your team as you put into practice one of the chief tools provided by DDD? Is your *Ubiquitous Language* formed from a set of well-known nouns? Nouns are important, but often software developers put too much emphasis on the nouns within a domain model, forgetting that spoken language is composed of far more than nouns alone. True, we have mainly focused on nouns within our previous sample *Bounded Contexts* to this point, but that's because we were interested in another aspect of DDD, that of constraining a *Core Domain* down to essential model elements.

Accelerate Your Discovery

You may want to try a few *Event Storming* sessions as you work on your scenarios. These can help you to quickly understand which scenarios you should be working on, and how they should be prioritized. Likewise, developing concrete scenarios will give you a better idea of the direction that you should take in your *Event Storming* sessions. They are two tools that work well together. I explain the use of *Event Storming* in Chapter 7 , "Acceleration and Management Tools ."

Don't limit your *Core Domain* to nouns alone. Rather, consider expressing your *Core Domain* as a set of concrete scenarios about what the domain model is supposed to do. When I say "scenarios" I don't mean use cases or user stories, such as is common in software projects. I literally mean

scenarios in terms of how the domain model should work—what the various components do. This can be accomplished in the most thorough way only by collaborating as a team of both *Domain Experts* and developers.

Here's an example of a scenario that fits with the *Ubiquitous Language* of Scrum:

> *Allow each backlog item to be committed to a sprint. The backlog item may be committed only if it is already scheduled for release. If it is already committed to a different sprint, it must be uncommitted first. When the commit completes, notify interested parties.*

Notice that this is not just a scenario about how humans use Scrum on a project. We are not talking about human procedures. Rather, this scenario is a description of how the very real software model components are used to support the management of a Scrum-based project.

The previous scenario is not a perfectly stated one, and a perk of using DDD is that we are constantly on the lookout for ways to improve the model. Yet this is a decent start. We hear nouns spoken, but our scenario doesn't limit us to nouns. We also hear verbs and adverbs, and other kinds of grammar. You also hear that there are constraints—conditions that must be met before the scenario can be completed to its successful end. The most important benefit and empowering feature is that you can actually have conversations about how the domain model works—its design.

We can even draw simple pictures and diagrams. It's all about doing whatever is needed to communicate well on the team. One word of warning is appropriate here. Be careful about the time spent in your domain-modeling efforts when it comes to keeping documents with written scenarios and drawings and diagrams up-to-date over the long haul. Those things are not the domain model. Rather, they are just tools to help you develop a domain model. In the end the code is the model and the model is the code. Ceremony is for distinguished observances, like weddings, not domain models. This doesn't mean that you forgo any efforts to freshen scenarios, but only do so as long as it is helpful rather than burdensome.

What would you do to improve a part of the *Ubiquitous Language* in our previous example? Think about it for just a minute. What's missing? Before too long you probably wish for an understanding of *who* does the committing of backlog items to a sprint. Let's add the *who* and see what happens:

> *The product owner commits each backlog item to a sprint . . .*

You will find in many cases that you should name each persona involved in the scenario and give some distinguishing attribute to other concepts such as to the backlog item and sprint. This will help to make your scenario more concrete and less like a set of statements about acceptance criteria. Still, in this particular case there isn't a strong reason to name the product owner or further describe the backlog item and sprint involved. In this case all product owners, backlog items, and sprints will work the same way whether or not they have a concrete persona or identity. In cases where giving names or other distinguishing identities to concepts in the scenario helps, use them:

> *The product owner Isabel commits the View User Profile backlog item to the Deliver User Profiles sprint . . .*

Now let's pause for a moment. It's not that the product owner is the sole individual responsible for deciding that a backlog item will be committed to a sprint. Scrum teams wouldn't like that very much, because they would be committed to delivering software within some time frame that they had no say in determining. Still, for our software model it may be most practical for a single person to have the

responsibility to carry out this particular action on the model. So in this case we have stated that it's the product owner role that does this. Even so, the nature of Scrum teams forces the question "Is there anything that must be done by the remainder of the team to enable the product owner to perform the commitment?"

Do you see what has happened? By challenging the current model with the *who* question, we have been led to an opportunity for deeper insight into the model. Perhaps we should require at least some team consensus that a backlog item can be committed before actually allowing the product owner to carry out the commit operation. This could lead to the following refined scenario:

> *The product owner commits a backlog item to a sprint. The backlog item may be committed only if it is already scheduled for release, and if a quorum of team members have approved commitment . . .*

OK, now we have a refined *Ubiquitous Language* , because we have identified a new model concept called a *quorum.* We decided that there must be a *quorum* of team members who agree that a backlog item should be committed, and there must be a way for them to *approve* commitment. This has now introduced a new modeling concept and some idea that the user interface will have to facilitate these team interactions. Do you see the innovation unfolding?

There is another *who* missing from the model. Which one? Our opening scenario concluded:

> *When the commit completes, notify interested parties.*

Who or what are the interested parties? This question and challenge further lead to modeling insights. Who needs to know when a backlog item has been committed to a sprint? Actually one important model element is the sprint itself. The sprint needs to track total sprint commitment, and what effort is already required to deliver all the sprint's tasks. However you decide to design the sprint to track that, the important point now is for the sprint to be notified when a backlog item is committed to it:

> *If it is already committed to a different sprint, it must be uncommitted first. When the commitment completes, notify the sprint from which it was uncommitted and the sprint to which it is now committed.*

Now we have a fairly decent domain scenario. This concluding sentence has also led us to an understanding that the backlog item and the sprint may not necessarily be aware of commitment at the same time. We need to ask the business to be certain, but it sounds like a great place to introduce *eventual consistency.* You will see why that is important and how it is accomplished in Chapter 5 , "Tactical Design with Aggregates ."

The refined scenario in its entirety looks like this:

> *The product owner commits a backlog item to a sprint. The backlog item may be committed only if it is already scheduled for release, and if a quorum of team members have approved commitment. If it is already committed to a different sprint, it must be uncommitted first. When the commitment completes, notify the sprint from which it was uncommitted and the sprint to which it is now committed.*

How would a software model actually work in practice? You can well imagine a very innovative user interface supporting this software model. As a Scrum team is participating in a sprint planning

session, team members use their smartphones or other mobile devices to add their approval to each backlog item as it is discussed and agreed upon to work on during the next sprint. The consensus of the quorum of team members approving each of the backlog items gives the product owner the ability to commit all of the approved backlog items to the sprint.

Putting Scenarios to Work

You may be wondering how you can make the transition from a written scenario to some sort of artifact that can be used to validate your domain model against the team's specifications. There is a technique named *Specification by Example* [Specification] that can be used; it's also called *Behavior-Driven Development* [BDD] . What you are trying to achieve with this approach is to collaboratively develop and refine a *Ubiquitous Language* , model with a shared understanding, and determine whether your model adheres to your specifications. You will do this by creating acceptance tests. Here is how we might restate the preceding scenario as an executable specification:

Click here to view code image

```
Scenario:
  The product owner commits a backlog item to a sprint
   Given a backlog item that is scheduled for release
   And the product owner of the backlog item
   And a sprint for commitment
   And a quorum of team approval for commitment
   When the product owner commits the backlog item to the sprint
   Then the backlog item is committed to the sprint
   And the backlog item committed event is created
```

With a scenario written in this form, you can create some backing code and use a tool to execute this specification. Even without a tool, you may find that this form of scenario authoring with its given/when/then approach works better than the previous scenario authoring example. Yet executing your specifications as a means of validating the domain model may be hard to resist. I comment on this further in Chapter 7 , "Acceleration and Management Tools ."

You don't have to use this form of executable specification in order to validate your domain model against your scenarios. You can use a unit testing framework to accomplish much the same thing, where you create acceptance tests (not unit tests) that validate your domain model:

Click here to view code image

```
/*
 The product owner commits a backlog item to a sprint.
 The backlog item may be committed only if it is already
 scheduled for release, and if a quorum of team members
 have approved commitment. When the commitment completes,
 notify the sprint to which it is now committed.
*/

[Test]
public void ShouldCommitBacklogItemToSprint()
{
  // Given
  var backlogItem = BacklogItemScheduledForRelease();

  var productOwner = ProductOwnerOf(backlogItem);
```

```
    var sprint = SprintForCommitment();

    var quorum = QuorumOfTeamApproval(backlogItem, sprint);

    // When
    backlogItem.CommitTo(sprint, productOwner, quorum);

    // Then
    Assert.IsTrue(backlogItem.IsCommitted());

    var backlogItemCommitted =
      backlogItem.Events.OfType<BacklogItemCommitted>().SingleOrDefault();

    Assert.IsNotNull(backlogItemCommitted);
  }
```

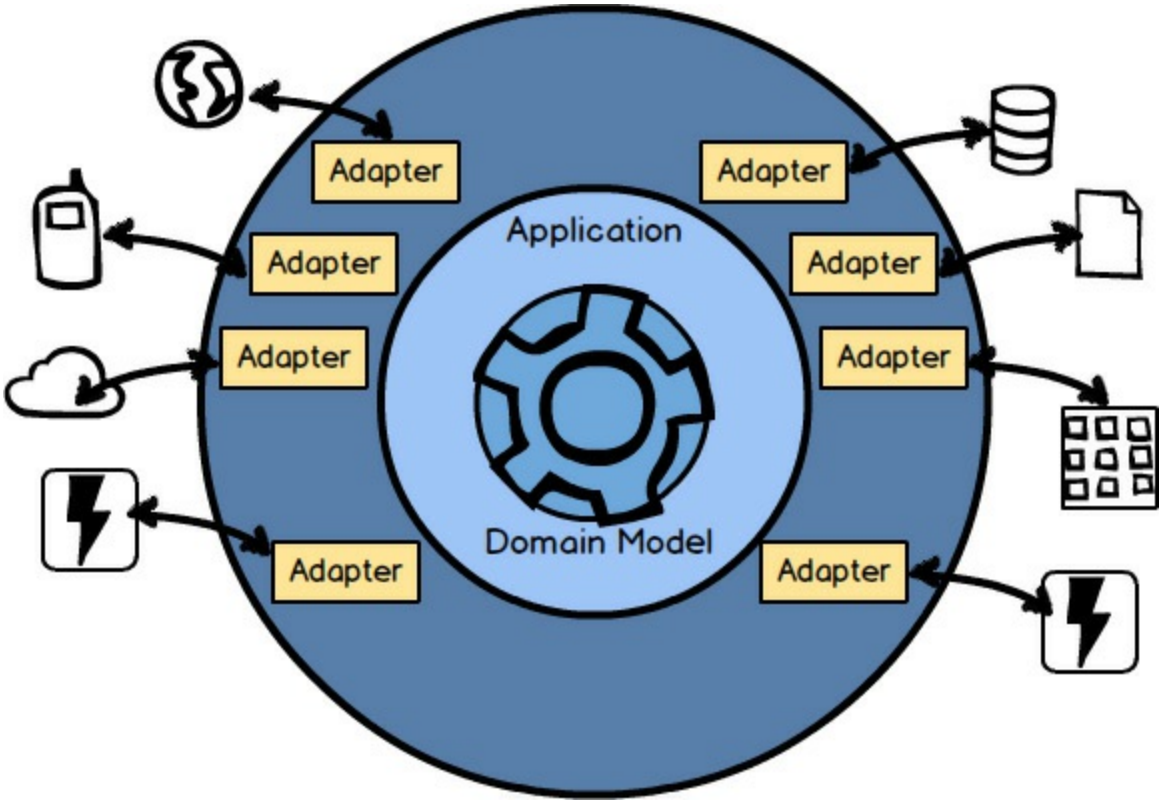
This unit-test-based approach to acceptance testing accomplishes the same goal as the executable specification. The advantage here may be the ability to write this kind of scenario validation more rapidly but at the cost of some readability. Still, most *Domain Experts* should be able to follow this code with some help from the developer. When using this approach it would probably work best to maintain the document form of the scenario associated with the validation code in comments, as seen in this example.

Whichever approach you decide on, both will generally be used in a red-green (fail-pass) fashion, where your specification will first fail when run, because there is no implementation of domain model concepts yet to be validated. You stepwise refine your domain model through a series of red results until you fully support your specification and the validations pass (you see all green). These acceptance tests will be directly associated with your *Bounded Context* and kept in its source code repository.

What about the Long Haul?

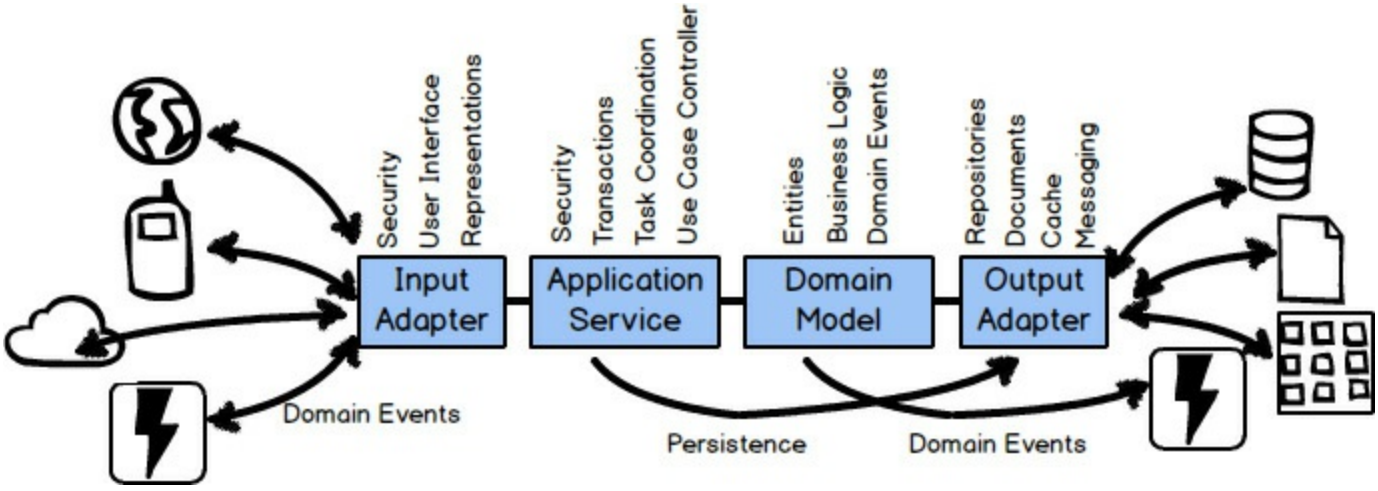
Now you may be wondering how we should support the *Ubiquitous Language* once the innovation has ceased and maintenance sets in. Actually, some of the best learning, or knowledge acquisition, takes place over a long period of time, even during what some might refer to as “maintenance.” It is a mistake for teams to take the view that innovation ends when maintenance begins.

Perhaps the worst thing that could happen is for the label “maintenance phase” to be attached to a *Core Domain.* A continuous learning process is not a phase at all. The *Ubiquitous Language* that was developed early on must continue to thrive as years pass. True, it may eventually be less significant, but probably not for quite a while. It is all part of your organization’s commitment to a core initiative. If this long-term commitment cannot be made, is this model that you are working on today truly a strategic differentiator, a *Core Domain* ?



Architecture

There is another question that you may have wondered about. What's inside a *Bounded Context* ? Using this *Ports and Adapters* [IDDD] architecture diagram, you can see that a *Bounded Context* is composed of more than a domain model.



These layers are common in a *Bounded Context: Input Adapters* , such as user interface controllers, REST endpoints, and message listeners; *Application Services* that orchestrate use cases and manage transactions; the domain model that we've been focusing on; and *Output Adapters* such as persistence management and message senders. There is much to be said about the various layers in this architecture, and it is too elaborate to state in this distilled book. See Chapter 4 of *Implementing Domain-Driven Design* [IDDD] for an exhaustive discussion.

Technology-Free Domain Model

Although there will be technology scattered throughout your architecture, the domain model should be free of technology. For one thing, that's why transactions are managed by the application services and not by the domain model.

Ports and Adapters can be used as a foundational architecture, but it's not the only one that can be used along with DDD. Along with Ports and Adapters, you can use DDD with any of these architectures or architecture patterns (and others), mixing and matching them as needed:

- Event-Driven Architecture; *Event Sourcing* [IDDD] . Note that *Event Sourcing* is discussed in this book in Chapter 6 , "Tactical Design with Domain Events ."
- Command Query Responsibility Segregation (CQRS) [IDDD] .
- Reactive and Actor Model; see *Reactive Messaging Patterns with the Actor Model* [Reactive] , which also elaborates on the use of the Actor model with DDD.
- Representational State Transfer (REST) [IDDD] .
- Service-Oriented Architecture (SOA) [IDDD] .
- Microservices are explained in *Building Microservices* [Microservices] as essentially equivalent to DDD *Bounded Contexts* , so both the book you are reading and *Implementing Domain-Driven Design* [IDDD] discuss the development of microservices from that perspective.
- Cloud computing is supported in much the same way as microservices, such that anything you read in this book, in *Implementing Domain-Driven Design* [IDDD] , and in *Reactive Messaging Patterns with the Actor Model* [Reactive] is applicable.

Another comment on microservices is in order. Some consider a microservice to be much smaller than a DDD *Bounded Context*. Using that definition, a microservice models only one concept and manages one narrow type of data. An example of such a microservice is a `Product` and another is a `BacklogItem` . If this is the granularity that you consider a worthy microservice, understand that both the `Product` microservice and the `BacklogItem` microservice will still be in the same larger, logical *Bounded Context*. The two small microservice components have only different deployment units, which may also have an impact on how they interact (see *Context Mapping*). Linguistically they are still within the same Scrum-based contextual and semantic boundary.

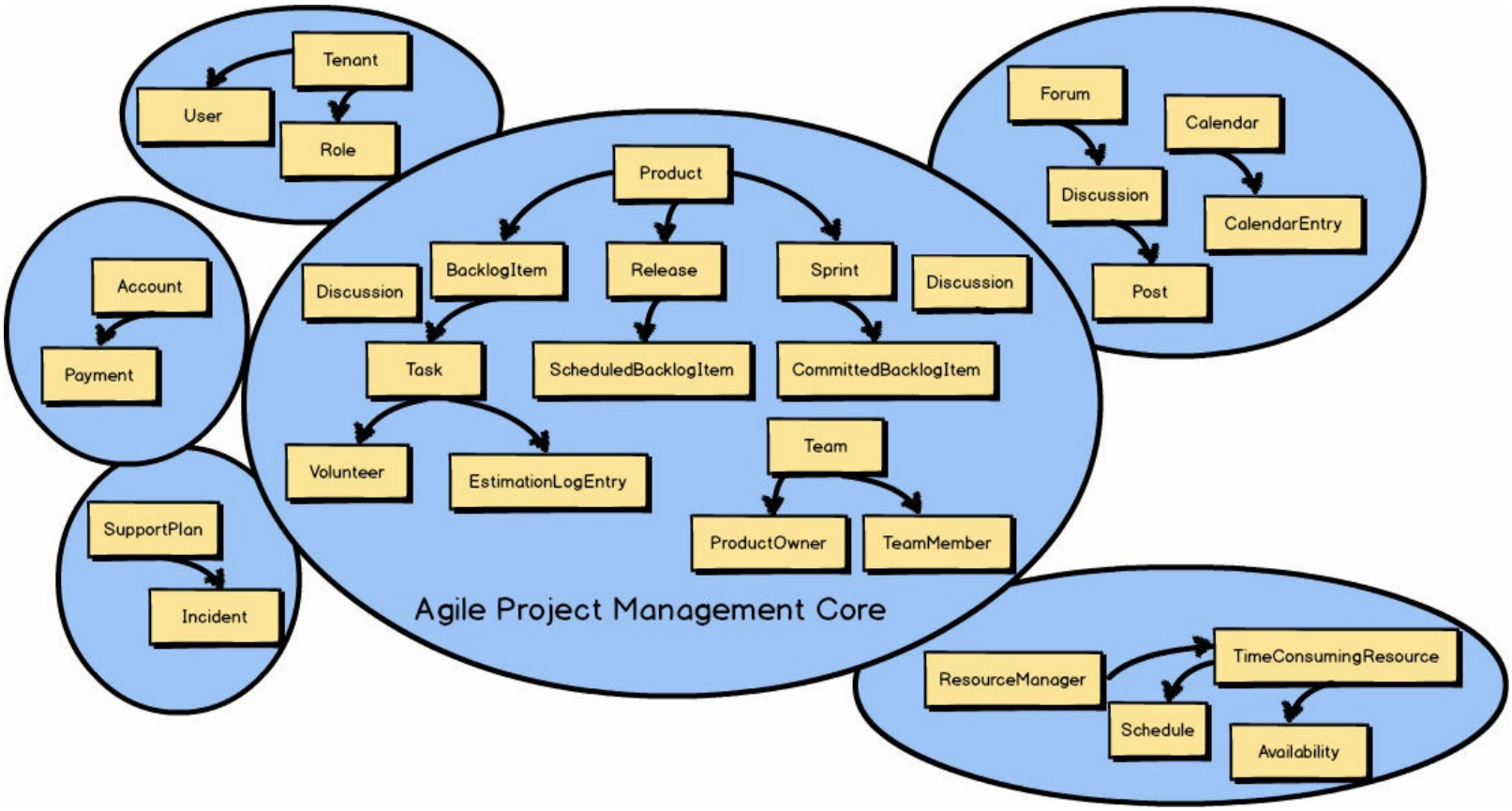
Summary

In summary you have learned:

- Some of the major pitfalls of putting too much into one model and creating a *Big Ball of Mud*
- The application of DDD strategic design
- The use of *Bounded Context* and *Ubiquitous Language*
- How to challenge your assumptions and unify mental models
- How to develop a *Ubiquitous Language*
- About the architectural components found inside a *Bounded Context*
- That DDD is not too difficult to put into practice yourself!

For a more in-depth treatment of *Bounded Contexts* , see Chapter 2 of *Implementing Domain-Driven Design* [IDDD] .

Chapter 3. Strategic Design with Subdomains



When you work on a DDD project, there are always multiple *Bounded Contexts* in play. One of the *Bounded Contexts* will be the *Core Domain* , and there will also be various *Subdomains* in other *Bounded Contexts*. In the previous chapter you saw the importance of dividing different models by their specific *Ubiquitous Language* and forming multiple *Bounded Contexts* . There are six *Bounded Contexts* and six *Subdomains* in the preceding diagram. Because DDD strategic design was used, the teams achieved the most optimal modeling composition: one *Subdomain* per *Bounded Context* , and one *Bounded Context* per *Subdomain*. In other words, the *Agile Project Management Core* is both one clean *Bounded Context* and one clean *Subdomain*. In some situations, there may be multiple *Subdomains* in one *Bounded Context* , but that doesn't achieve the most optimal modeling outcome.

What Is a Subdomain?

Simply stated, a *Subdomain* is a sub-part of your overall business domain. You can think of a *Subdomain* as representing a single, logical domain model. Most business domains are usually too large and complex to reason about as a whole, so we generally concern ourselves only with the *Subdomains* that we must use within a single project. *Subdomains* can be used to logically break up your whole business domain so that you can understand your *problem space* on a large, complex project.

Another way to think of a *Subdomain* is that it is a clear area of expertise, assuming that it is responsible for providing a solution to a core area of your business. This implies that the particular *Subdomain* will have one or more *Domain Experts* who understand very well the aspects of the business that a specific *Subdomain* facilitates. The *Subdomain* also has greater or lesser strategic significance to your business.

If DDD had been used to develop it, the *Subdomain* would have been implemented as a clean

Bounded Context. The *Domain Experts* who specialize in that particular area of the business would have been members of the team that developed the *Bounded Context*. Although using DDD to develop a clean *Bounded Context* is the optimal choice, sometimes we can only wish that had been the case.

Types of Subdomains

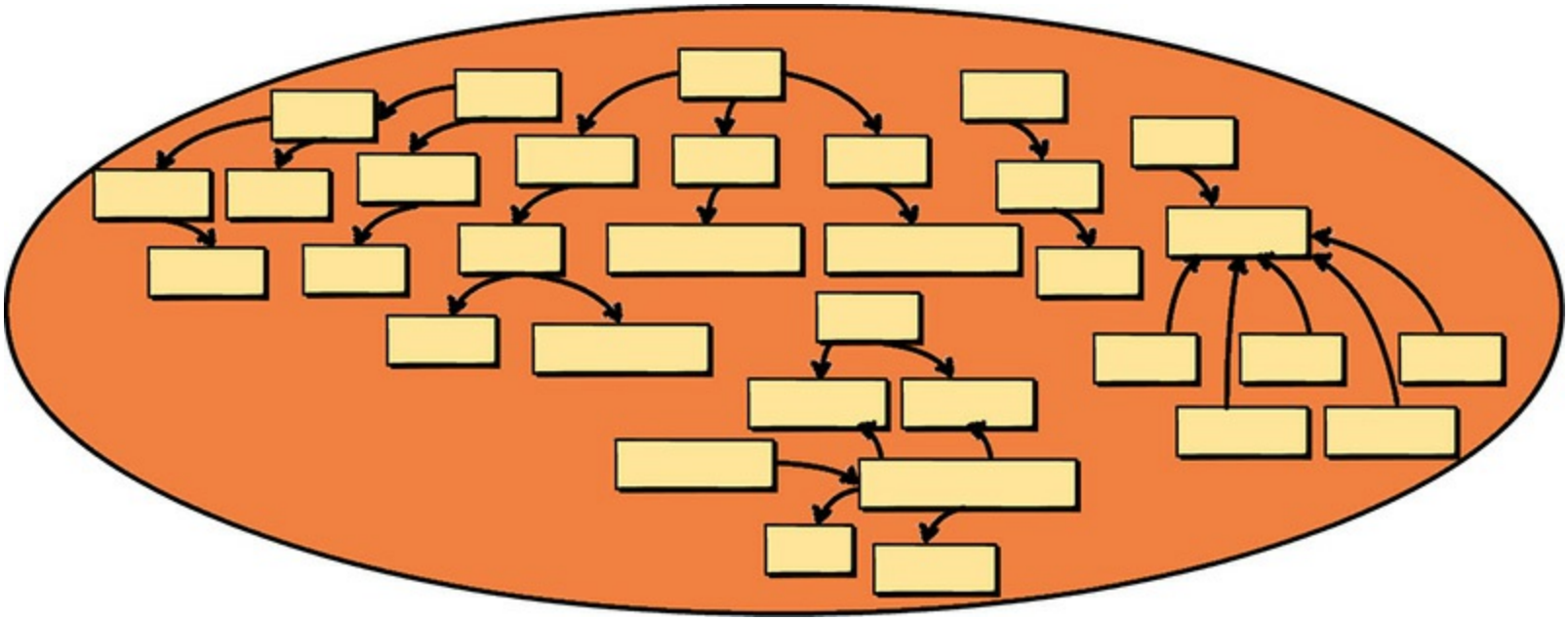
There are three primary types of *Subdomains* within a project:

- *Core Domain:* This is where you are making a strategic investment in a single, well-defined domain model, committing significant resources for carefully crafting your *Ubiquitous Language* in an explicit *Bounded Context*. This is very high on your organization's list of projects because it will distinguish it from all competitors. Since your organization can't be distinguished in everything that it does, your *Core Domain* demarcates where it must excel. Achieving the level of deep learning and understanding required to make such a determination requires commitment, collaboration, and experimentation. It's where the organization needs to invest most liberally in software. I provide the means to accelerate and manage such projects efficiently and effectively later in this book.
- *Supporting Subdomain:* This is a modeling situation that calls for custom development, because an off-the-shelf solution doesn't exist. However, you will still not make the kind of investment that you have made for your *Core Domain*. You may want to consider outsourcing this kind of *Bounded Context* to avoid mistaking it for something strategically distinguishing, and thus investing heavily in it. This is still an important software model, because your *Core Domain* cannot be successful without it.
- *Generic Subdomain:* This kind of solution may be available for purchase off the shelf but may also be outsourced or even developed in house by a team that doesn't have the kind of elite developers that you assign to your *Core Domain* or even a lesser *Supporting Subdomain*. Be careful not to mistake a *Generic Subdomain* for a *Core Domain*. You don't want to make that kind of investment here.

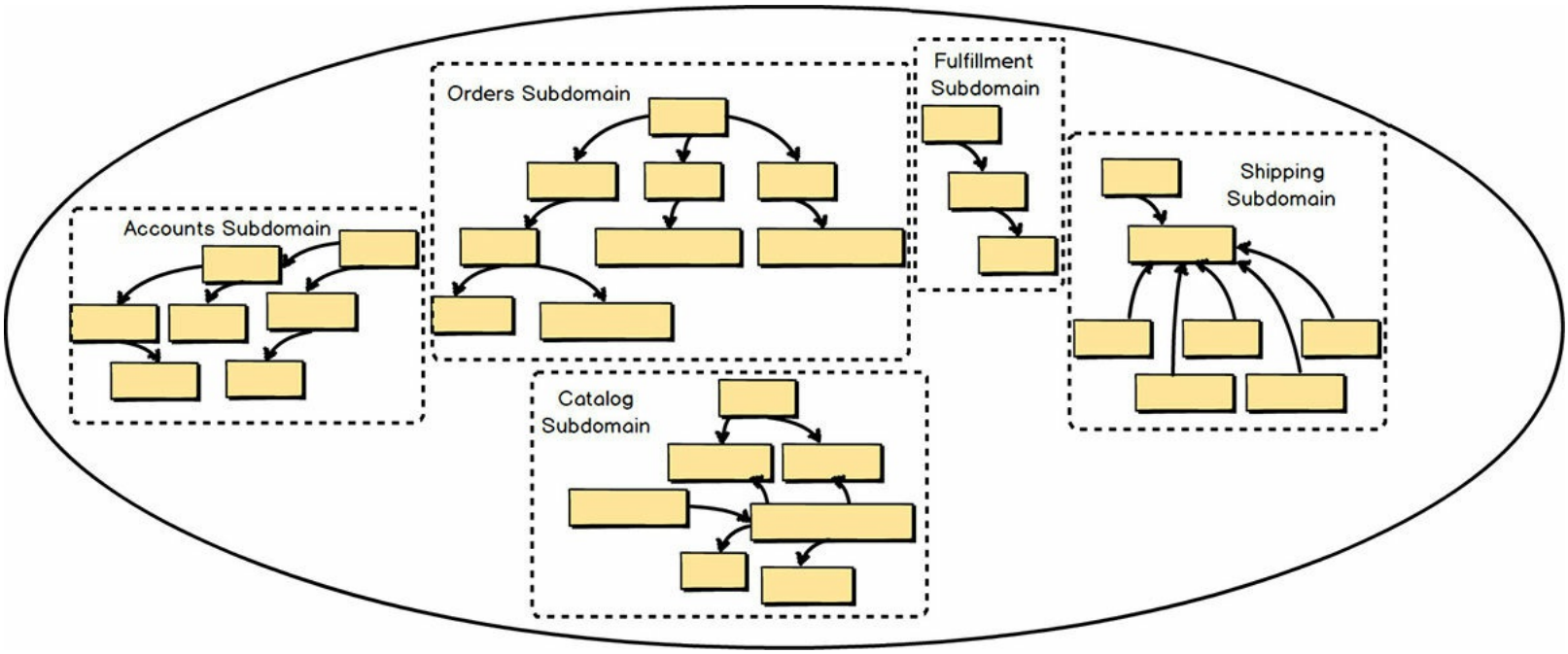
When discussing a project where DDD is being employed, we are most likely discussing a *Core Domain*.

Dealing with Complexity

Some of the system boundaries within a business domain will very likely be legacy systems, perhaps those that your organization has created or those that have been purchased through software licensing. At this point you may not be able to do much about improving those legacy systems, but you still need to reason about them when they have an impact on your *Core Domain* project. To do so, use *Subdomains* as a tool for discussing your *problem space*.



Unfortunately, but true all the same, some legacy systems are so counter to the DDD way of designing with *Bounded Contexts* that you might even refer to them as *unbounded* legacy systems. That's because such a legacy system is what I've already referred to as a *Big Ball of Mud.* In reality the one system is full of multiple tangled models that should have been separately designed and implemented but were jumbled together into one very complex and intertwined mess.

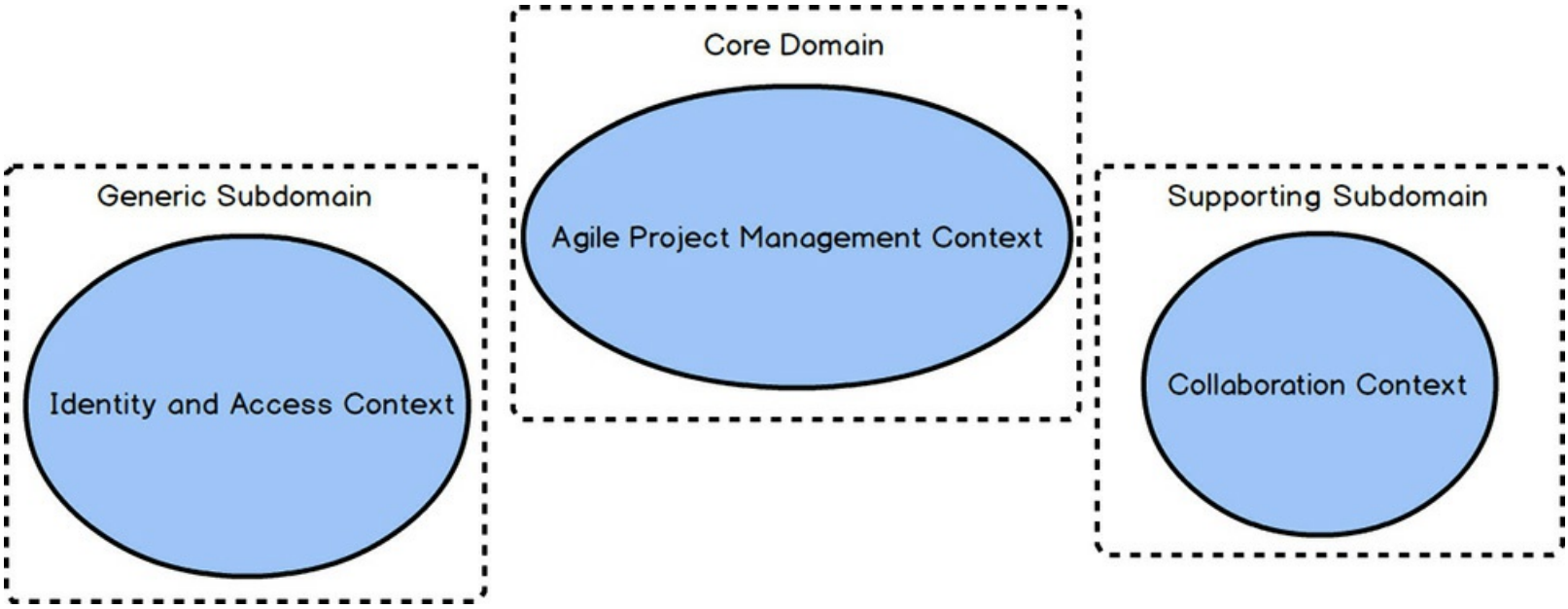


Stated another way, when we are discussing a legacy system there are probably some, even many, *logical* domain models that exist inside that one legacy system. Think of each of those logical domain models as a *Subdomain.* In the diagram, each logical *Subdomain* in the unbounded legacy monolithic *Big Ball of Mud* is marked off by a dashed box. There are five logical models or *Subdomains*. Treating the logical *Subdomains* as such helps us grapple with the complexity of large systems. This makes a lot of sense because it allows us to treat the *problem space* as if it had been developed using DDD and multiple *Bounded Contexts*.

The legacy system seems less monolithic and muddy if we imagine separate *Ubiquitous Languages* , at least for the sake of understanding how we must integrate with it. Thinking about and discussing such legacy systems using *Subdomains* helps us cope with the harsh realities of a large entangled model. And as we reason using this tool, we can determine the *Subdomains* that are more

valuable to the business and necessary for our project, and those that can be relegated to lesser status.

With that in mind, you can even show the *Core Domain* that you are working on, or are about to work on, right in the same simple diagram. This will help you understand the associations and dependencies between *Subdomains*. But I will save the details of that discussion for *Context Mapping*.



When using DDD, a *Bounded Context* should align one-to-one (1:1) with a single *Subdomain*. That is, when using DDD, if there is one *Bounded Context* , there is, as a goal, one *Subdomain* model in that *Bounded Context*. It may not always be possible or practical to achieve, but where possible it is important to design in that way. This will keep your *Bounded Contexts* clean and focused on the core strategic initiative.

If you must create a second model in the same *Bounded Context* (within your *Core Domain*), you should segregate the secondary model from your *Core Domain* using a completely separate *Module* [IDDD] . (A DDD *Module* is basically a package in Scala and Java, and a namespace in F# and C#.) This makes a clear linguistic statement that one model is core and the other is merely supporting. This particular use of segregating a *Subdomain* is one that you would employ in your *solution space*.

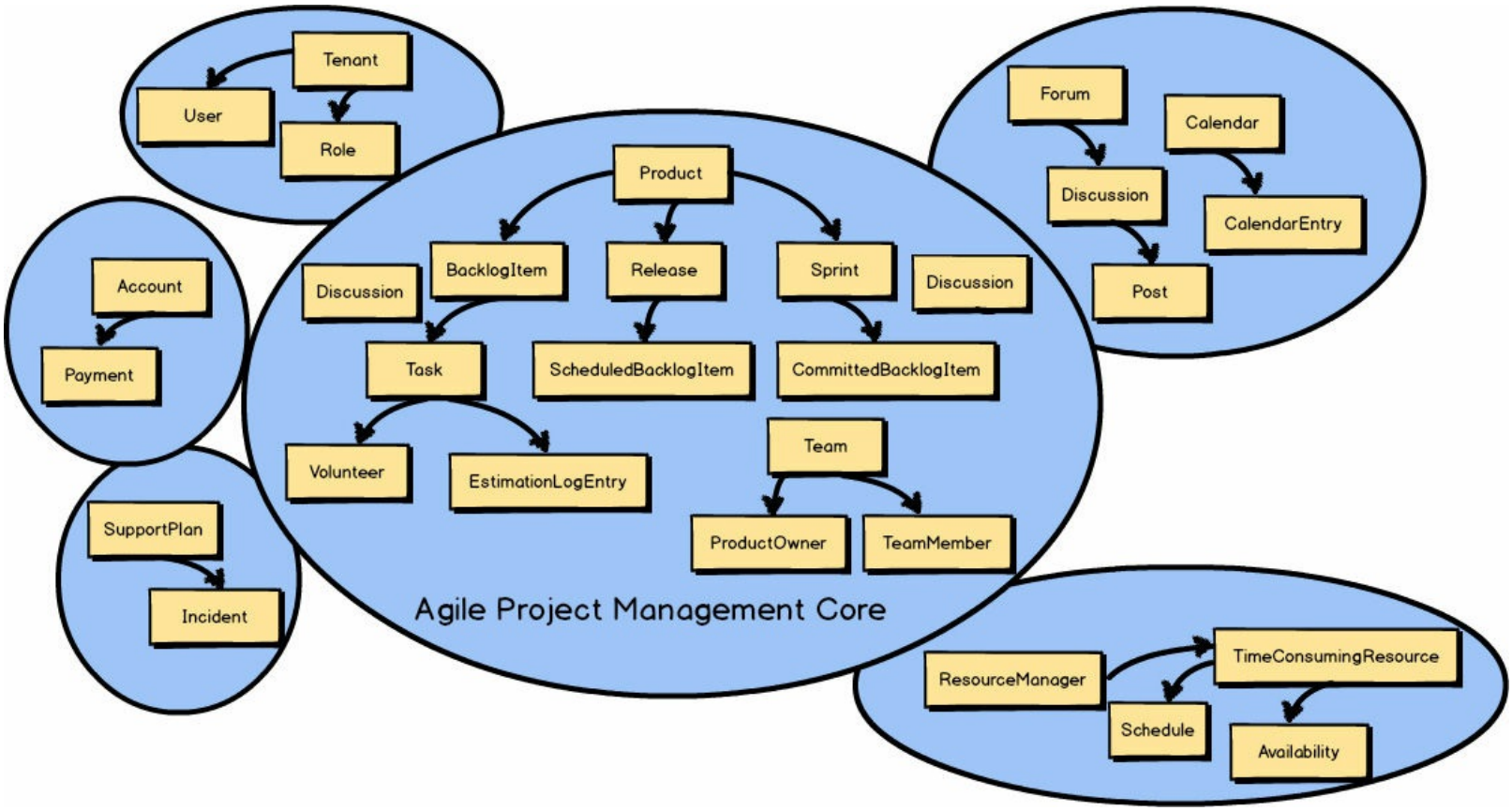
Summary

In summary you have learned:

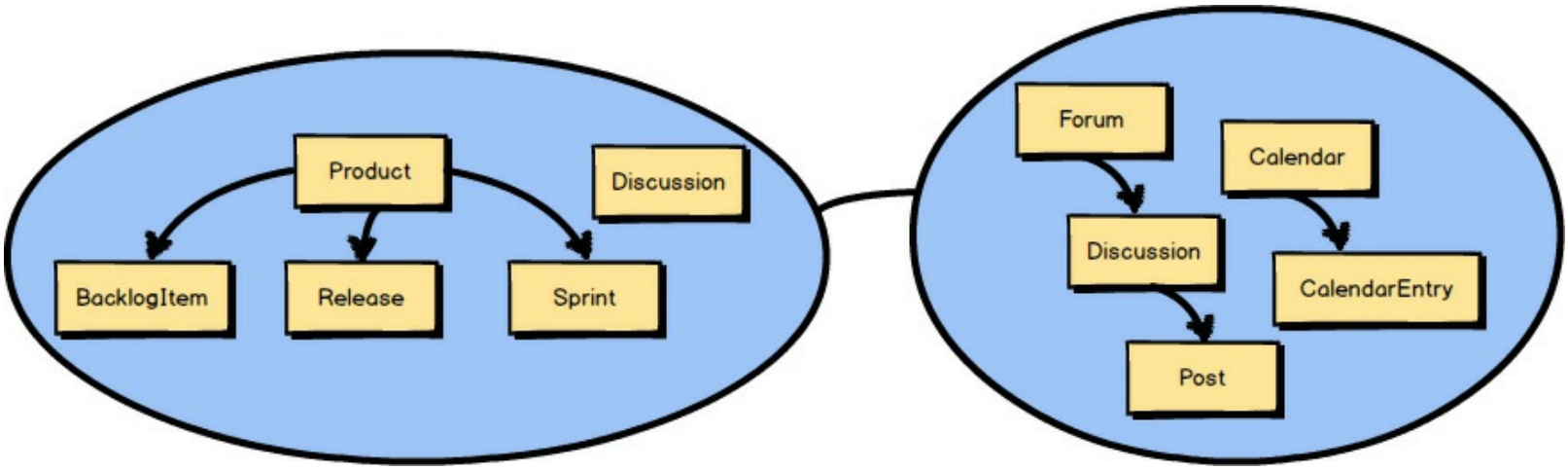
- What *Subdomains* are and how they are used, both in the *problem space* and in the *solution space*
- The difference between a *Core Domain* , a *Supporting Subdomain* , and a *Generic Subdomain*
- How you can make use of *Subdomains* while reasoning about integration with a *Big Ball of Mud* legacy system
- The importance of aligning your DDD *Bounded Context* one-to-one with a single *Subdomain*
- How you should segregate a *Supporting Subdomain* model from your *Core Domain* model using a DDD *Module* when it is impractical to separate the two in different *Bounded Contexts*

For exhaustive coverage of *Subdomains* , see Chapter 2 of *Implementing Domain-Driven Design* [IDDD] .

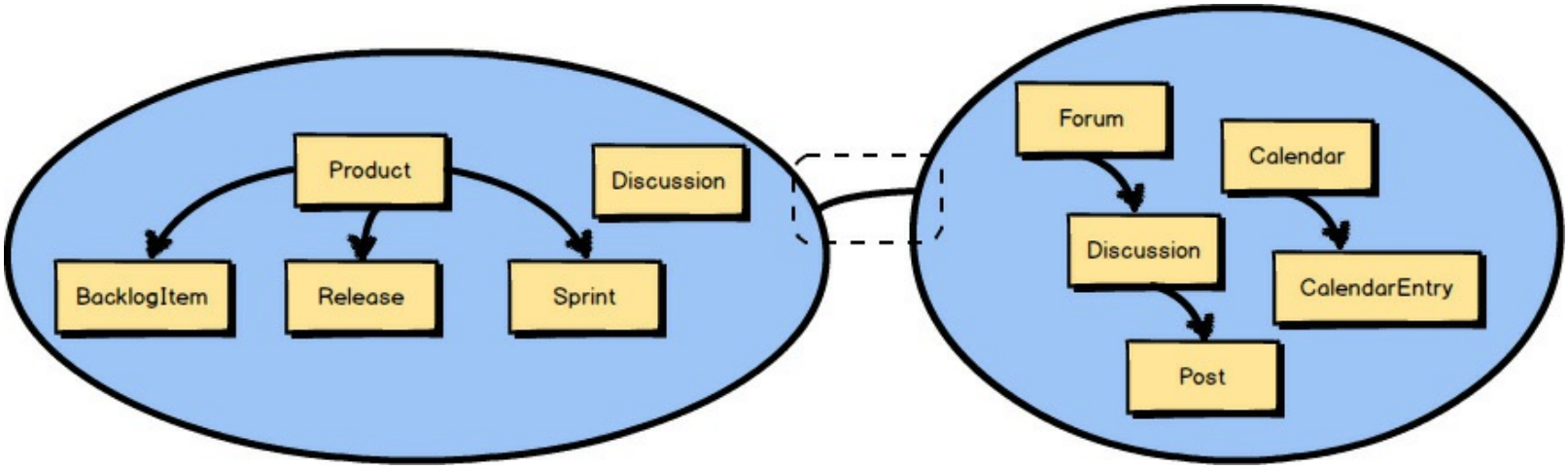
Chapter 4. Strategic Design with Context Mapping



In previous chapters you learned that in addition to the *Core Domain* , there are multiple *Bounded Contexts* associated with every DDD project. All concepts that didn't belong in the *Agile Project Management Context* —the *Core Domain* —were moved to one of several other *Bounded Contexts*.



You also learned that the Agile Project Management *Core Domain* would have to integrate with other *Bounded Contexts*. That integration is known in DDD as *Context Mapping*. You can see in the previous *Context Map* that `Discussion` exists in both *Bounded Contexts*. Recall that this is because the *Collaboration Context* is the source of the `Discussion` , and that the *Agile Project Management Context* is the consumer of the `Discussion` .



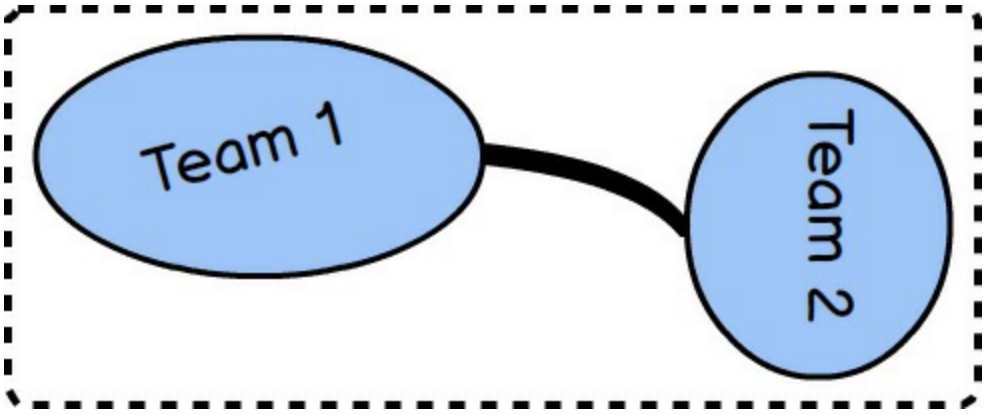
A *Context Mapping* is highlighted in this diagram by the line inside the dashed box. (The dashed box is not part of the *Context Mapping* but is used only to highlight the line.) It's actually this line between the two *Bounded Contexts* that represents a *Context Mapping*. In other words, the line indicates that the two *Bounded Contexts* are mapped in some way. There will be some inter-team dynamic between the two *Bounded Contexts* as well as some integration.

Considering that in two different *Bounded Contexts* there are two *Ubiquitous Languages* , this line represents the translation that exists between the two languages. By way of illustration, imagine that two teams need to work together, but they work across national boundaries and don't speak the same language. Either the teams would need an interpreter, or one or both teams would have to learn a great deal about the other's language. Finding an interpreter would be less work for both teams, but it could be expensive in various ways. For example, imagine the extra time needed for one team to talk to the interpreter, and then for the interpreter to relay the statements to the other team. It works fine for the first few moments but then becomes cumbersome. Still, the teams might find this a better solution than learning and embracing a foreign language and constantly switching between languages. And, of course, this describes the relationship only between two teams. What if there are a few other teams involved? Similarly, the same trade-offs would exist when translating one *Ubiquitous Language* into another, or in some other way adapting to another *Ubiquitous Language*.

When we talk about *Context Mapping* , what is of interest to us is *what kind* of inter-team relationship and integration is represented by the line between any two *Bounded Contexts*. Well-defined boundaries and contracts between them support controlled changes over time. There are several kinds of *Context Mappings* , both team and technical, that can be represented by the line. In some cases both an inter-team relationship and an integration mapping will be blended.

Kinds of Mappings

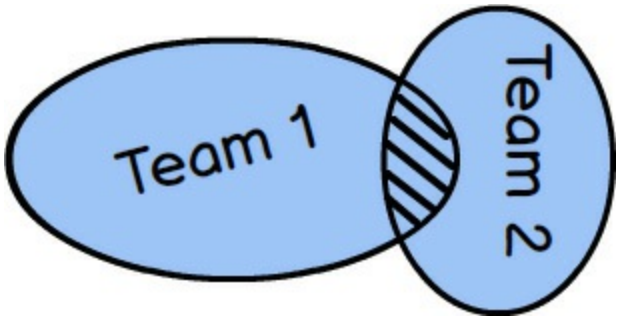
What relationships and integrations can be represented by the *Context Mapping* line? I will introduce them to you now.



Partnership

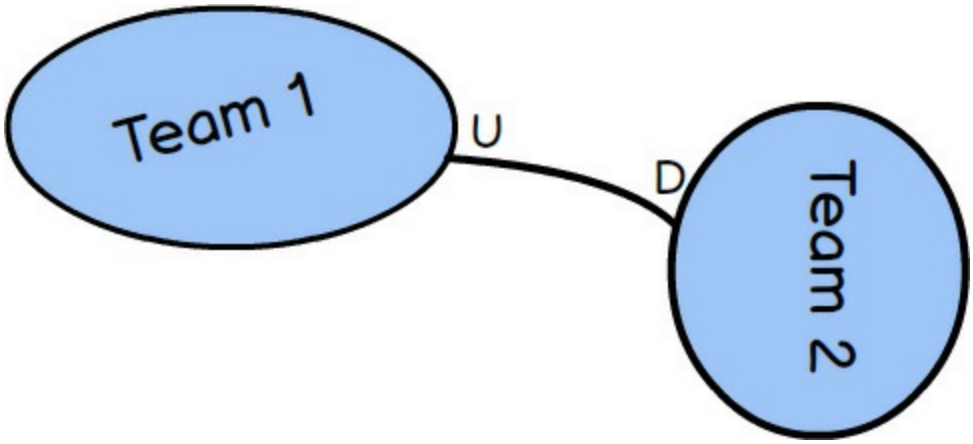
A *Partnership* relationship exists between two teams. Each team is responsible for one *Bounded Context*. They create a *Partnership* to align the two teams with a dependent set of goals. It is said that the two teams will succeed or fail together. Since they are so closely aligned, they will meet frequently to synchronize schedules and dependent work, and they will have to use continuous integration to keep their integrations in harmony. The synchronization is represented by the thick mapping line between the two teams. The thick line indicates the level of commitment required, which is quite high.

It can be challenging to maintain a *Partnership* over the long term, so many teams that enter a *Partnership* may do best to set limits on the term of the relationship. The *Partnership* should last only as long as it provides an advantage, and it should be remapped to a different relationship when the advantage is trumped by the drain of commitment.



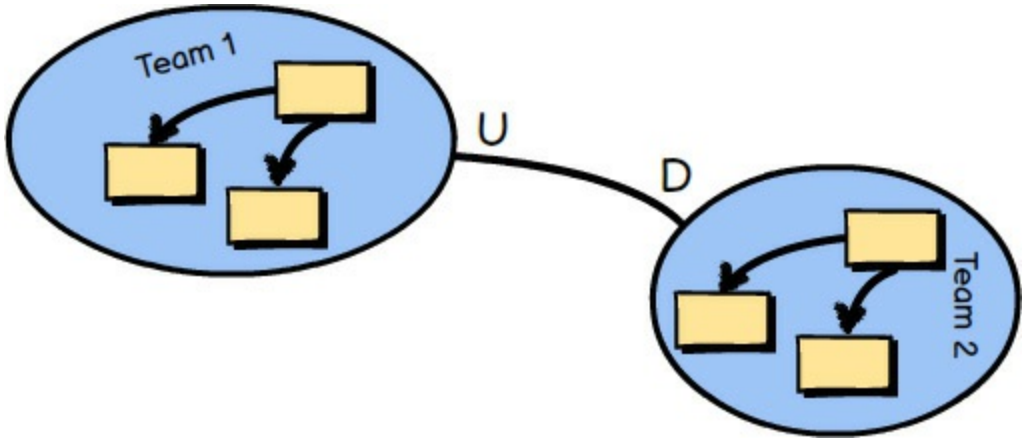
Shared Kernel

A *Shared Kernel* , depicted on page 54 by the intersection of the two *Bounded Contexts* , describes the relationship between two (or more) teams that share a small but common model. The teams must agree on what model elements they are to share. It's possible that only one of the teams will maintain the code, build, and test for what is shared. A *Shared Kernel* is often very difficult to conceive in the first place, and difficult to maintain, because you must have open communication between teams and constant agreement on what constitutes the model to be shared. Still, it is possible to be successful if all involved are committed to the idea that the kernel is better than going *Separate Ways* (see the later section).



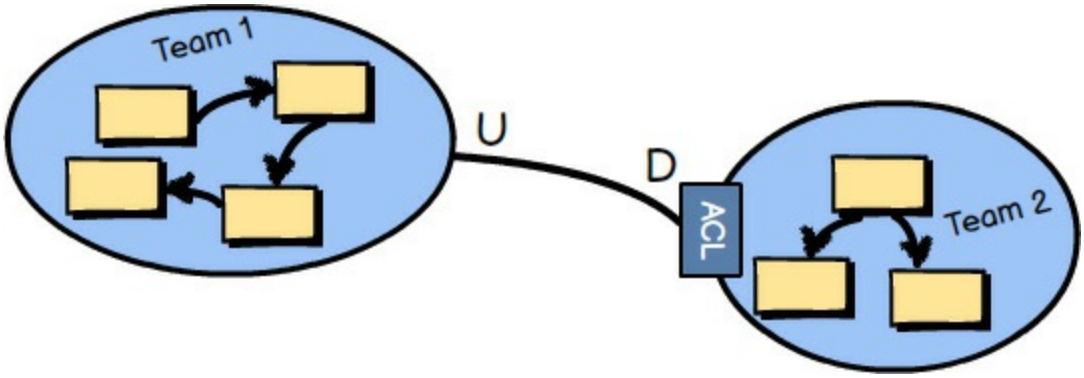
Customer-Supplier

A *Customer-Supplier* describes a relationship between two *Bounded Contexts* and respective teams, where the *Supplier* is upstream (the U in the diagram) and the *Customer* is downstream (the D in the diagram). The *Supplier* holds sway in this relationship because it must provide what the *Customer* needs. It's up to the *Customer* to plan with the *Supplier* to meet various expectations, but in the end the *Supplier* determines what the *Customer* will get and when. This is a very typical and practical relationship between teams, even within the same organization, as long as corporate culture does not allow the *Supplier* to be completely autonomous and unresponsive to the real needs of *Customers*.



Conformist

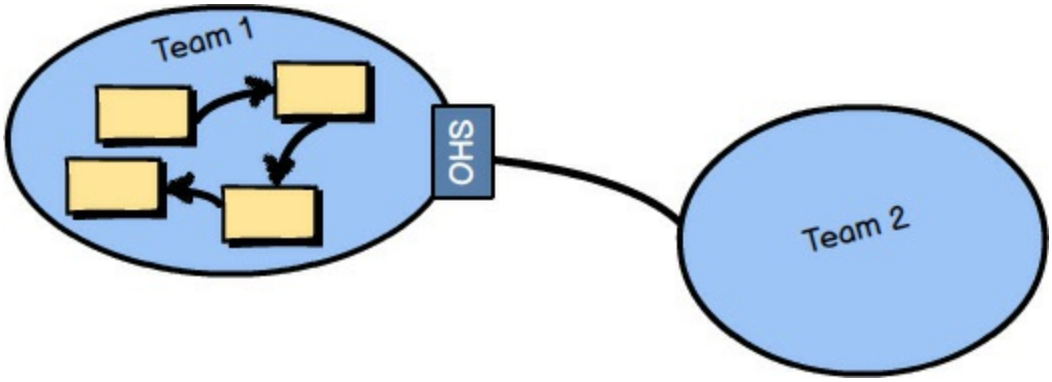
A *Conformist* relationship exists when there are upstream and downstream teams, and the upstream team has no motivation to support the specific needs of the downstream team. For various reasons the downstream team cannot sustain an effort to translate the *Ubiquitous Language* of the upstream model to fit its specific needs, so the team conforms to the upstream model as is. A team will often become a *Conformist* , for example, when integrating with a very large and complex model that is well established. Example: Consider the need to conform to the Amazon.com model when integrating as one of Amazon's affiliate sellers.



Anticorruption Layer

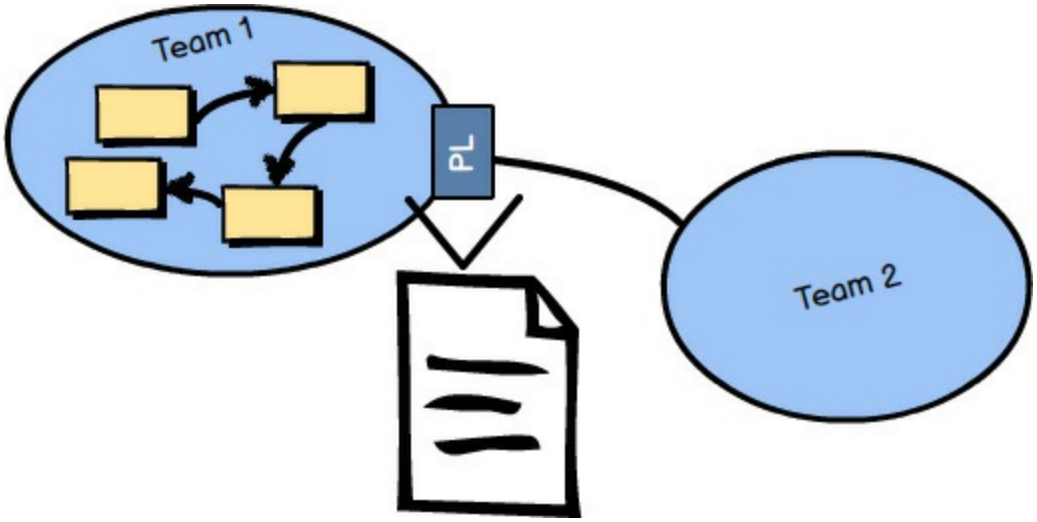
An *Anticorruption Layer* is the most defensive *Context Mapping* relationship, where the downstream team creates a translation layer between its *Ubiquitous Language* (model) and the *Ubiquitous Language* (model) that is upstream to it. The layer isolates the downstream model from the upstream model and translates between the two. Thus, this is also an approach to integration.

Whenever possible, you should try to create an *Anticorruption Layer* between your downstream model and an upstream integration model, so that you can produce model concepts on your side of the integration that specifically fit your business needs and that keep you completely isolated from foreign concepts. Yet, just like hiring a translator to act between two teams speaking different languages, the cost could be too high in various ways for some cases.



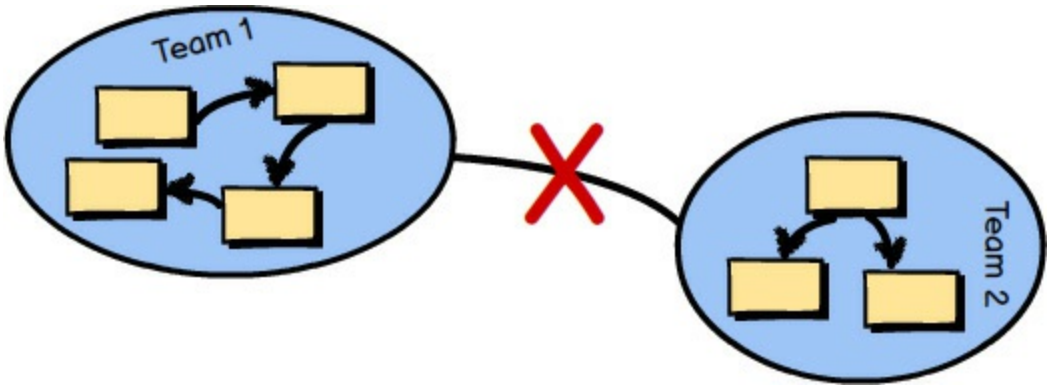
Open Host Service

An *Open Host Service* defines a protocol or interface that gives access to your *Bounded Context* as a set of services. The protocol is "open" so that all who need to integrate with your *Bounded Context* can use it with relative ease. The services offered by the application programming interface (API) are well documented and a pleasure to use. Even if you were Team 2 in this diagram and could not take time to create an isolating *Anticorruption Layer* for your side of the integration, it would be much more tolerable to be a *Conformist* to this model than many legacy systems you may encounter. We might say that the language of the *Open Host Service* is much easier to consume than that of other types of systems.



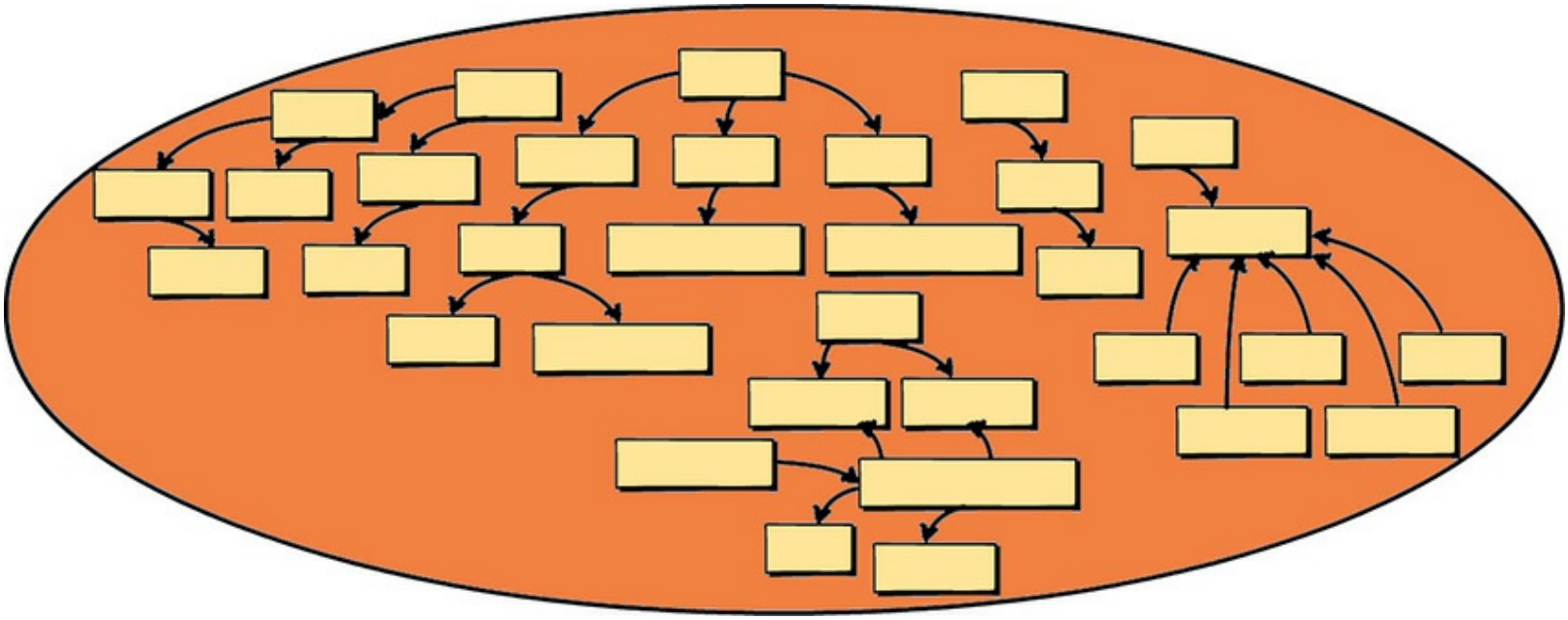
Published Language

A *Published Language* , illustrated in the image at the bottom of page 57 , is a well-documented information exchange language enabling simple consumption and translation by any number of consuming *Bounded Contexts*. Consumers who both read and write can translate from and into the shared language with confidence that their integrations are correct. Such a *Published Language* can be defined with XML Schema, JSON Schema, or a more optimal wire format, such as Protobuf or Avro. Often an *Open Host Service* serves and consumes a *Published Language* , which provides the best integration experience for third parties. This combination makes the translations between two *Ubiquitous Languages* very convenient.



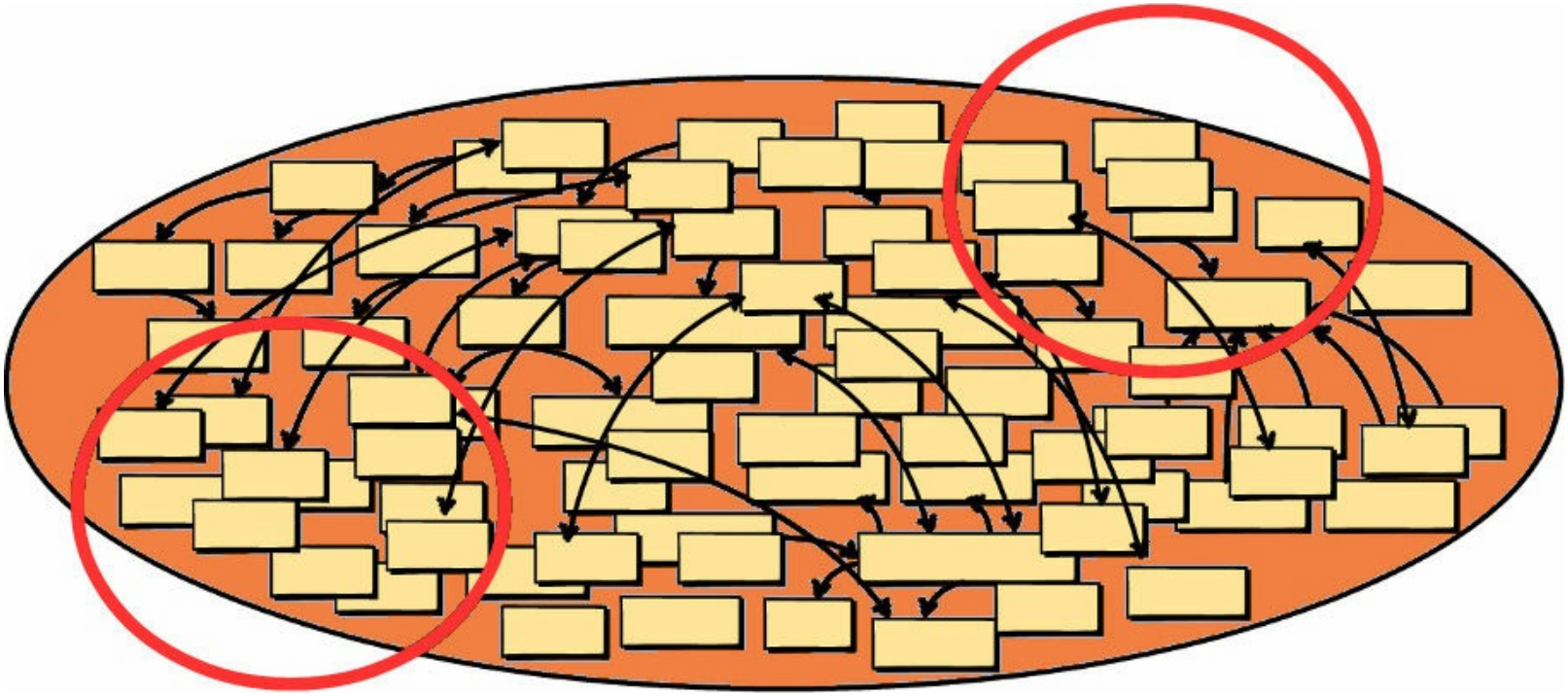
Separate Ways

Separate Ways describes a situation where integration with one or more *Bounded Contexts* will not produce significant payoff through the consumption of various *Ubiquitous Languages*. Perhaps the functionality that you seek is not fully provided by any one *Ubiquitous Language*. In this case produce your own specialized solution in your *Bounded Context* and forget integrating for this special case.

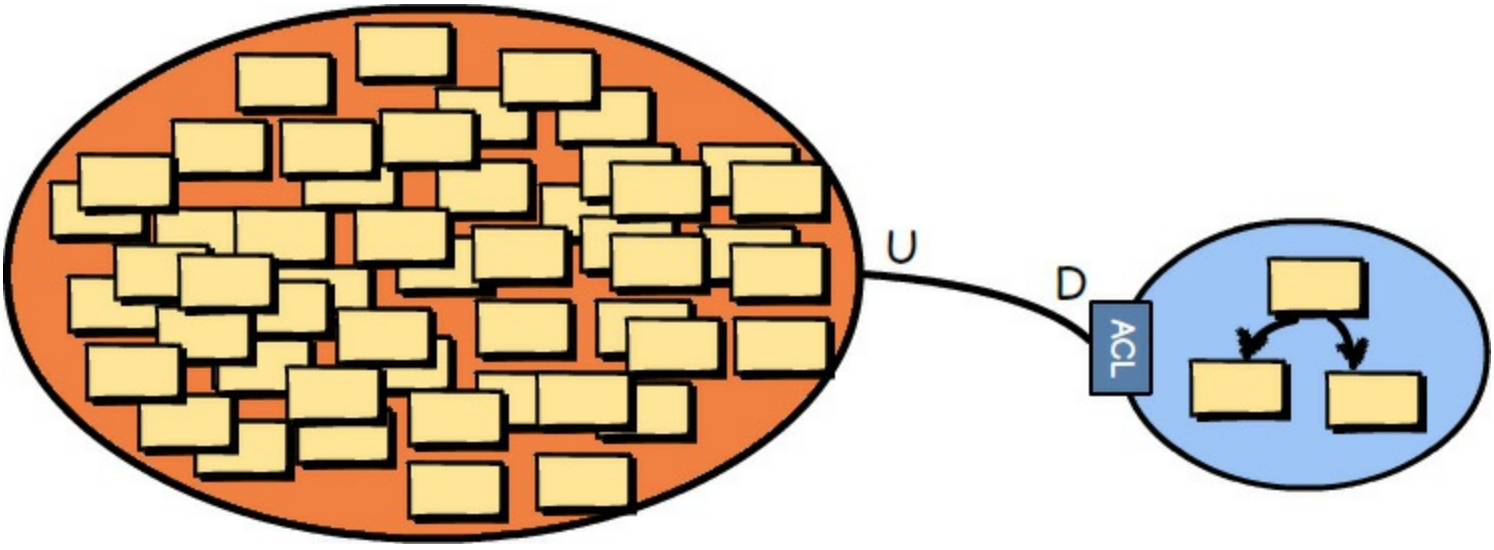


Big Ball of Mud

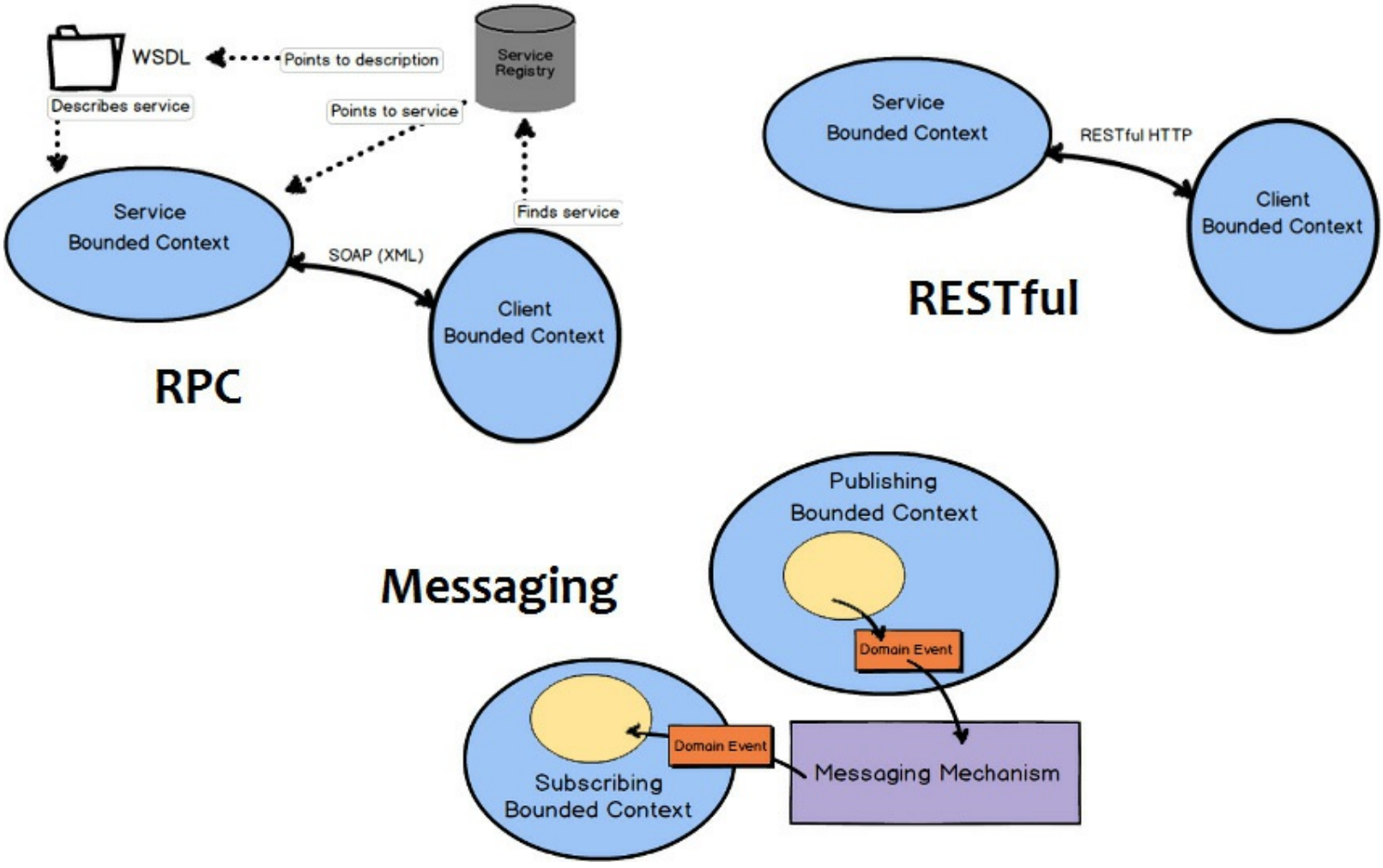
You already learned plenty about *Big Ball of Mud* in the previous chapters, but I am going to reinforce the serious problems you will experience when you must work in or integrate with one. Creating your own *Big Ball of Mud* should be avoided like the plague.



Just in case that's not enough warning, here's what happens over time when you are responsible for creating a *Big Ball of Mud:* (1) A growing number of *Aggregates* cross-contaminate because of unwarranted connections and dependencies. (2) Maintaining one part of the *Big Ball of Mud* causes ripples across the model, which leads to "whack-a-mole" issues. (3) Only tribal knowledge and heroics—speaking all languages at once—save the system from complete collapse.



The problem is that there are already many *Big Balls of Mud* out there in the wide world of software systems, and the number will no doubt grow every month. Even if you are able to avoid creating a *Big Ball of Mud* by employing DDD techniques, you may still need to integrate with one or more. If you must integrate with one or more, try to create an *Anticorruption Layer* against each legacy system in order to protect your own model from the cruft that would otherwise pollute your model with the incomprehensible morass. Whatever you do, *don't speak that language!*

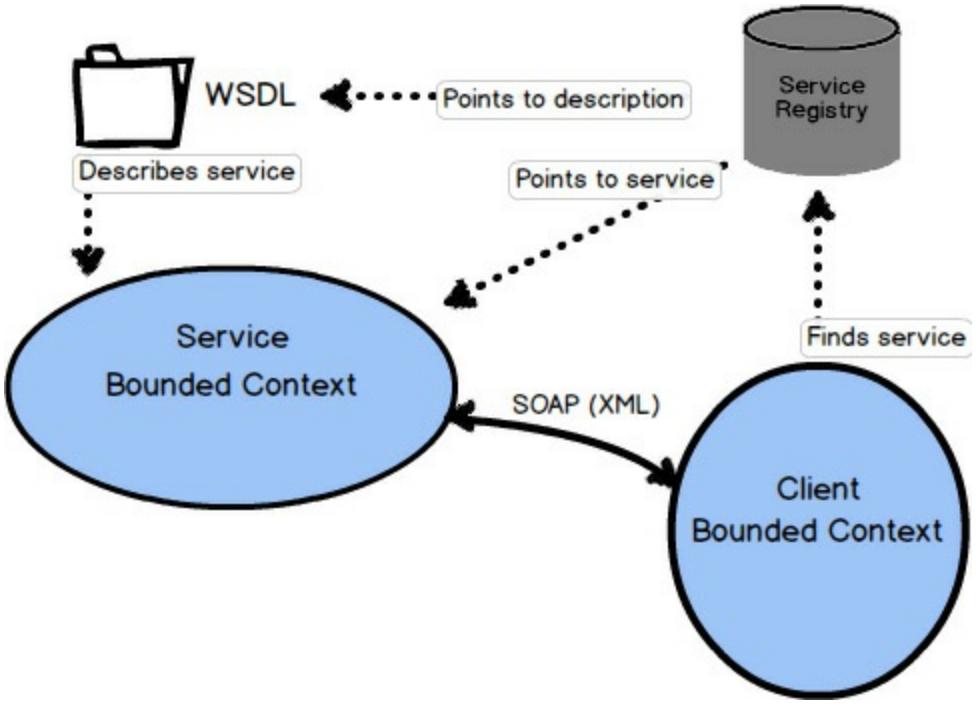


Making Good Use of Context Mapping

You may be wondering what specific kind of interface would be supplied to allow you to integrate with a given *Bounded Context*. That depends on what the team that owns the *Bounded Context* provides. It could be RPC via SOAP, or RESTful interfaces with resources, or it could be a

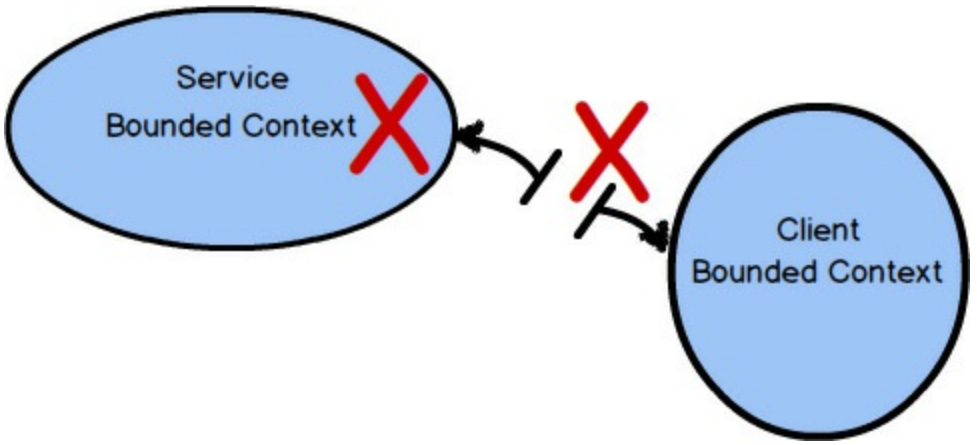
messaging interface using queues or Publish-Subscribe. In the least favorable of situations you may be forced to use database or file system integration, but let's hope that doesn't happen. Database integration really should be avoided, and if you are forced to integrate that way, you really should be sure to isolate your consuming model by means of an *Anticorruption Layer*.

Let's take a look at three of the more trustworthy integration types. We will move from the least robust to the most robust integration approaches. First we will look at RPC, followed by RESTful HTTP, and then messaging.



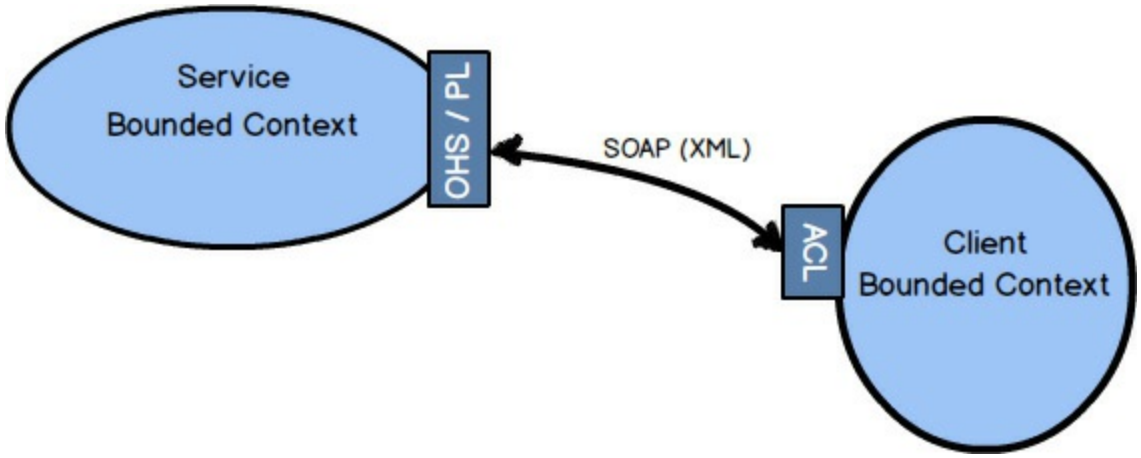
RPC with SOAP

Remote Procedure Calls, or RPC, can work in a number of ways. One popular way to use RPC is via the Simple Object Access Protocol, or SOAP. The idea behind RPC with SOAP is to make using services from another system look like a simple, local procedure or method invocation. Still, the SOAP request must travel over the network, reach the remote system, perform successfully, and return results over the network. This carries the potential for complete network failure, or at least latency that is unanticipated when first implementing the integration. Additionally, RPC over SOAP also implies strong coupling between a client *Bounded Context* and the *Bounded Context* providing the service.

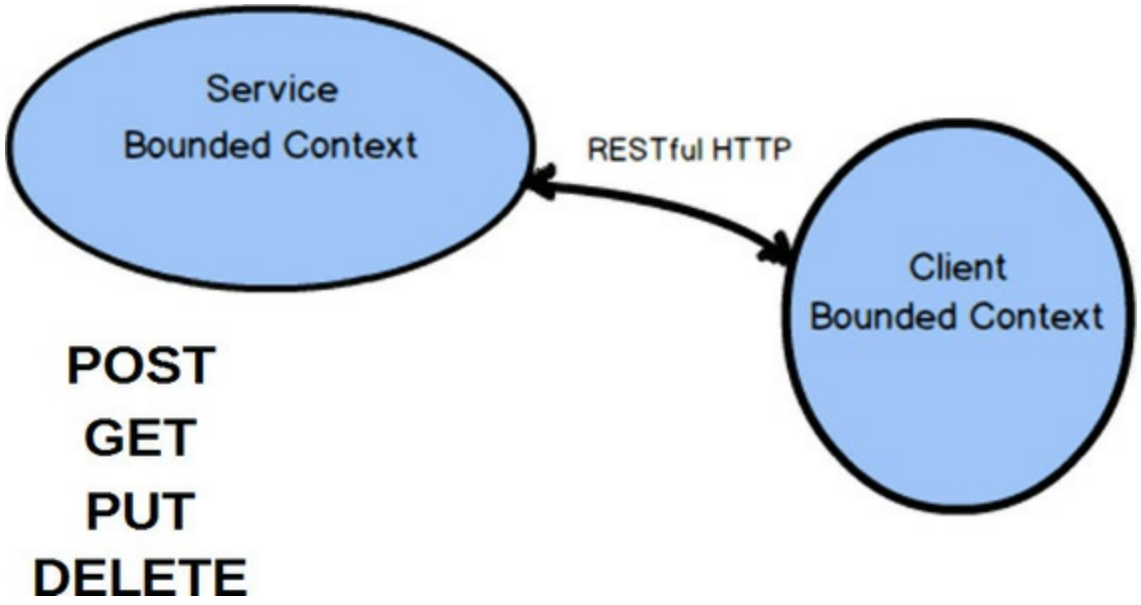


The main problem with RPC, using SOAP or another approach, is that it can lack robustness. If there is a problem with the network or a problem with the system hosting the SOAP API, your

seemingly simple procedure call will fail entirely, giving only error results. Don't be fooled by the seeming ease of use.

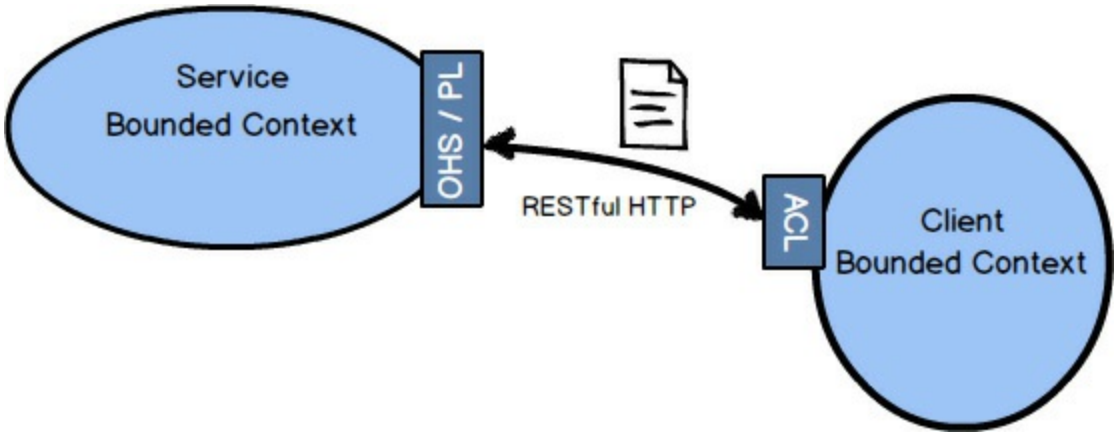


When RPC works—and it mostly works—it can be a very useful way to integrate. If you can influence the design of the service *Bounded Context* , it would be in your best interests if there is a well-designed API that provides an *Open Host Service* with a *Published Language*. Either way, your client *Bounded Context* can be designed with an *Anticorruption Layer* to isolate your model from unwanted outside influences.



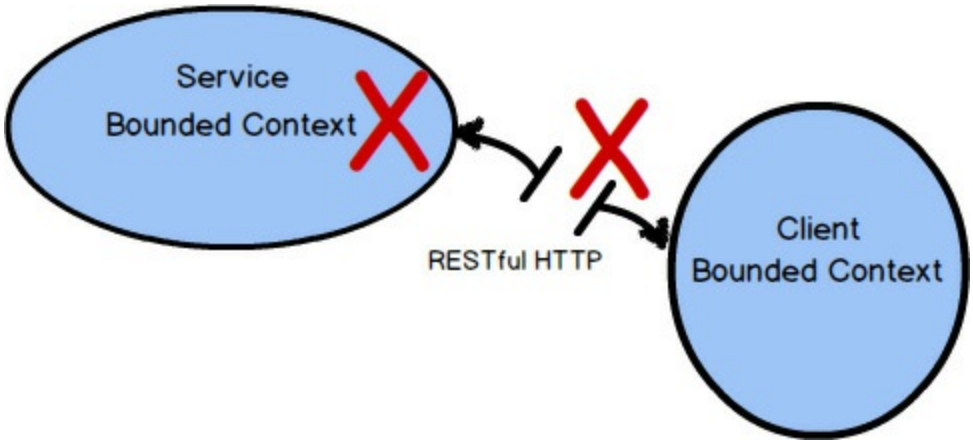
RESTful HTTP

Integration using RESTful HTTP focuses attention on the resources that are exchanged between *Bounded Contexts* , as well as the four primary operations: POST, GET, PUT, and DELETE. Many find that the REST approach to integration works well because it helps them define good APIs for distributed computing. It's difficult to argue against this claim given the success of the Internet and Web.

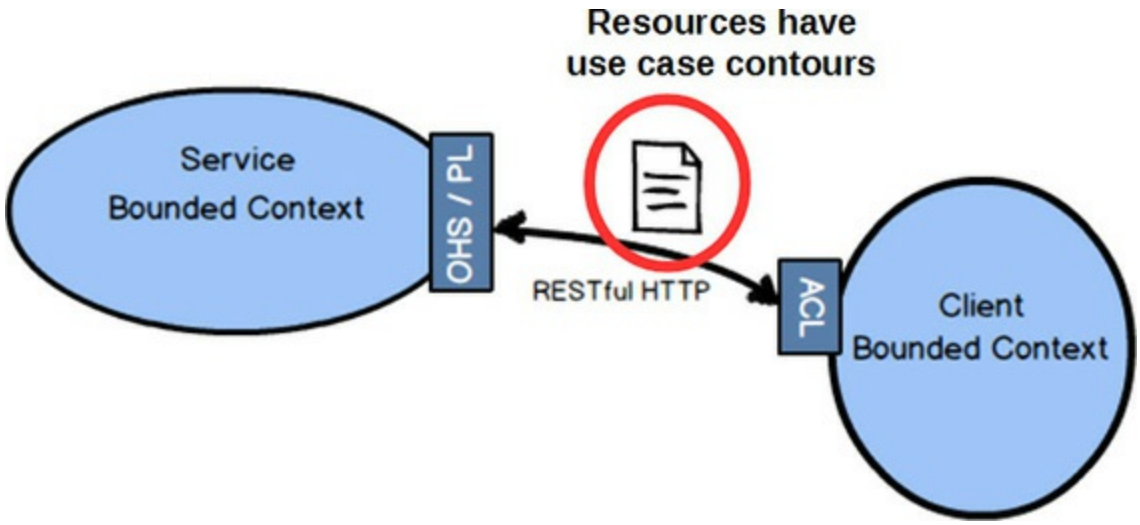


There is a very certain way of thinking when you use RESTful HTTP. I won't get into the details in this book, but you should look into it before trying to employ REST. The book *REST in Practice* [RiP] is a good place to start.

A service *Bounded Context* that sports a REST interface should provide an *Open Host Service* and a *Published Language*. Resources deserve to be defined as a *Published Language* , and combined with your REST URIs they will form a natural *Open Host Service*.

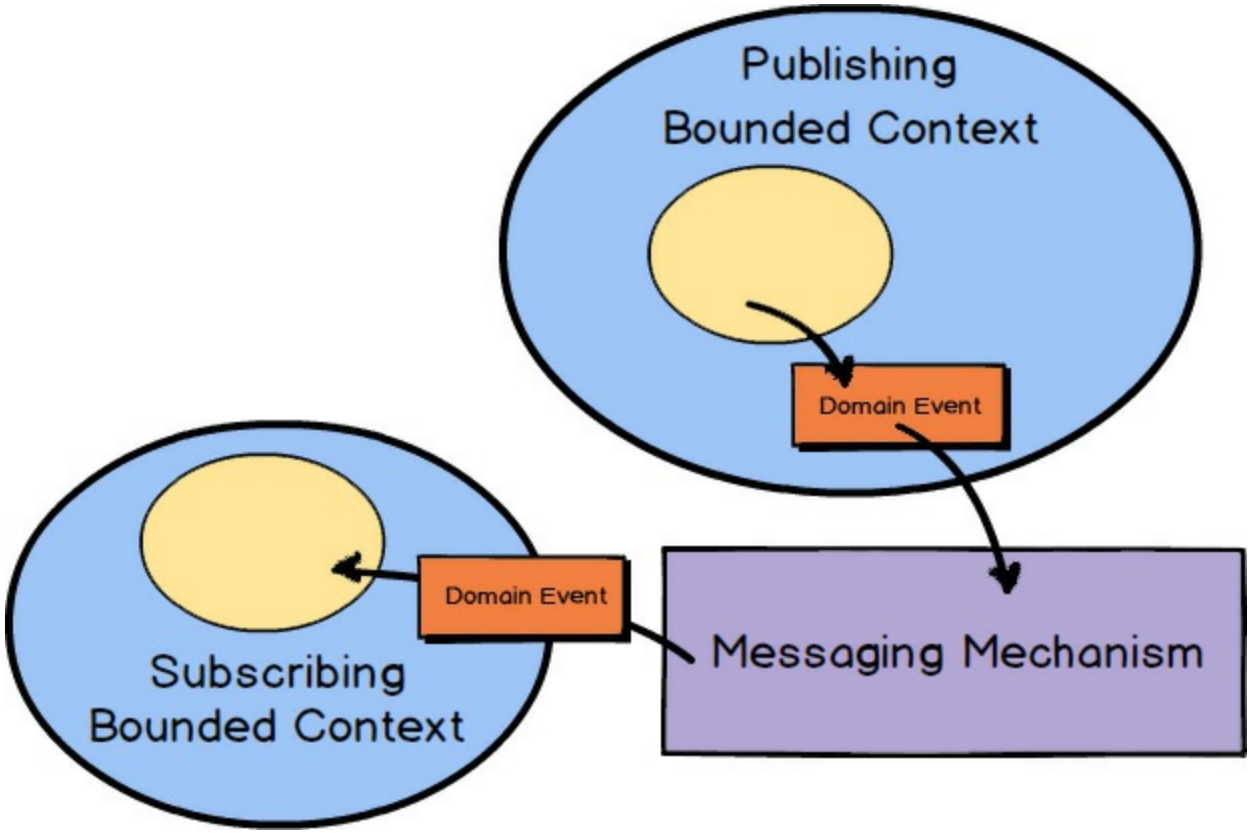


RESTful HTTP will tend to fail for many of the same reasons that RPC does—network and service provider failures, or unanticipated latency. However, RESTful HTTP is based on the premise of the Internet, and who can find fault with the track record of the Web when it comes to reliability, scalability, and overall success?



A common mistake made when using REST is to design resources that directly reflect the *Aggregates* in the domain model. Doing this forces every client into a *Conformist* relationship, where if the model changes shape the resources will also. So you don't want to do that. Instead, resources should be designed synthetically to follow client-driven use cases. By "synthetic" I mean that to the

client the resources provided must have the shape and composition of what they need, not what the actual domain model looks like. Sometimes the model will look just like what the client needs. But what the client needs is what drives the design of the resources, and not the model's current composition.



Messaging

When using asynchronous messaging to integrate, much can be accomplished by a client *Bounded Context* subscribing to the *Domain Events* published by your own or another *Bounded Context*. Using messaging is one of the most robust forms of integration because you remove much of the temporal coupling associated with blocking forms such as RPC and REST. Since you already anticipate the latency of message exchange, you tend to build more robust systems because you never expect immediate results.

Going Asynchronous with REST

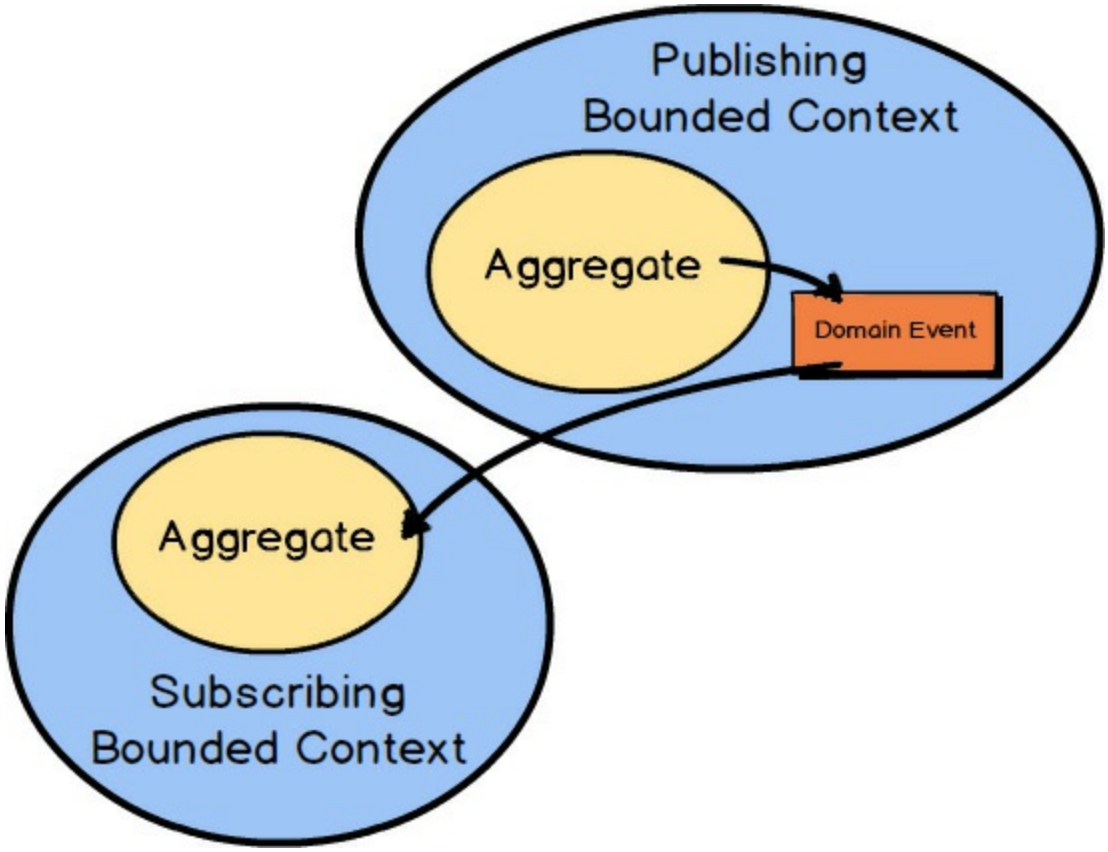
It's possible to accomplish asynchronous messaging using REST-based polling of a sequentially growing set of resources. Using background processing, a client would continuously poll for a service Atom feed resource that provides an ever-increasing set of *Domain Events*. This is a safe approach to maintaining asynchronous operations between a service and clients, while supplying up-to-date events that continue to occur in the service. If the service becomes unavailable for some reason, clients will simply retry on normal intervals, or back off with retry, until the feed resource is available again.

This approach is discussed in detail in *Implementing Domain-Driven Design* [IDDD]

.

Avoid Integration Train Wrecks

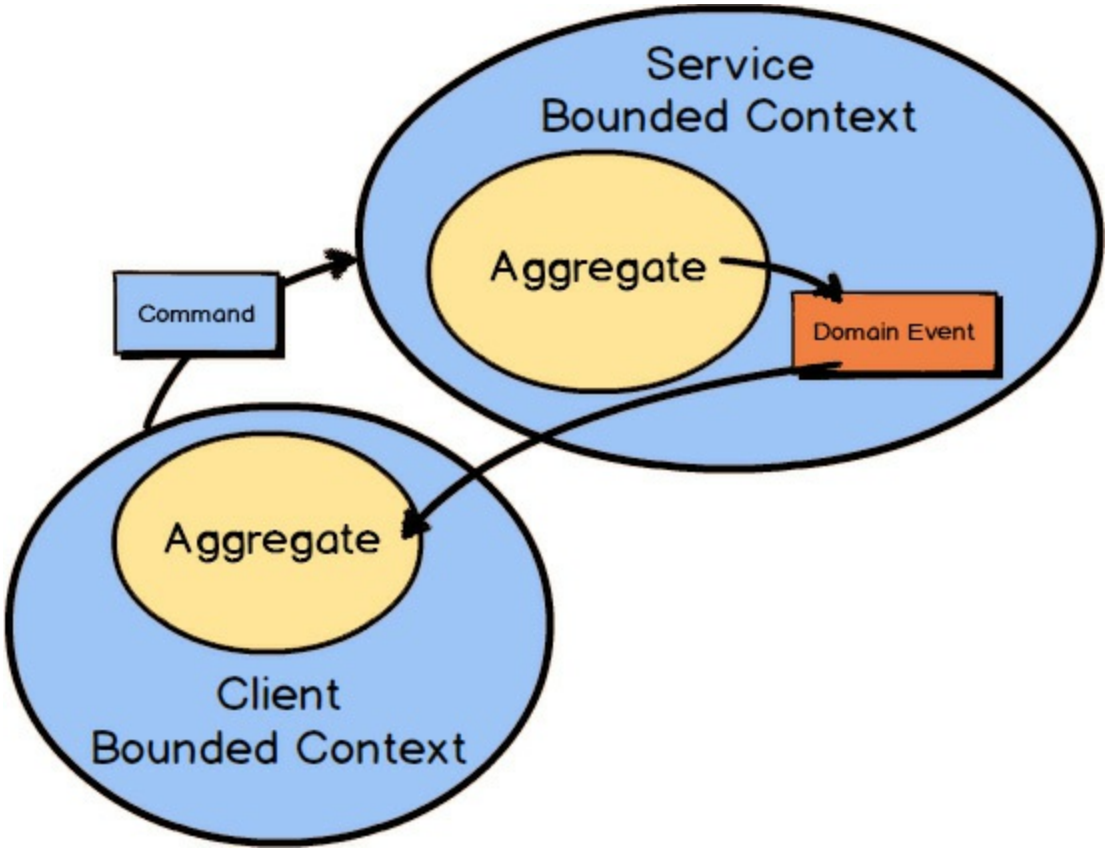
When a client *Bounded Context* (C1) integrates with a service *Bounded Context* (S1), C1 should usually not be making a synchronous, blocking request to S1 as a direct result of handling a request made to it. That is, while some other client (C0) makes a blocking request to C1, don't allow C1 to make a blocking request to S1. Doing so has a very high potential for causing an integration train wreck between C0, C1, and S1. This can be avoided by using asynchronous messaging.



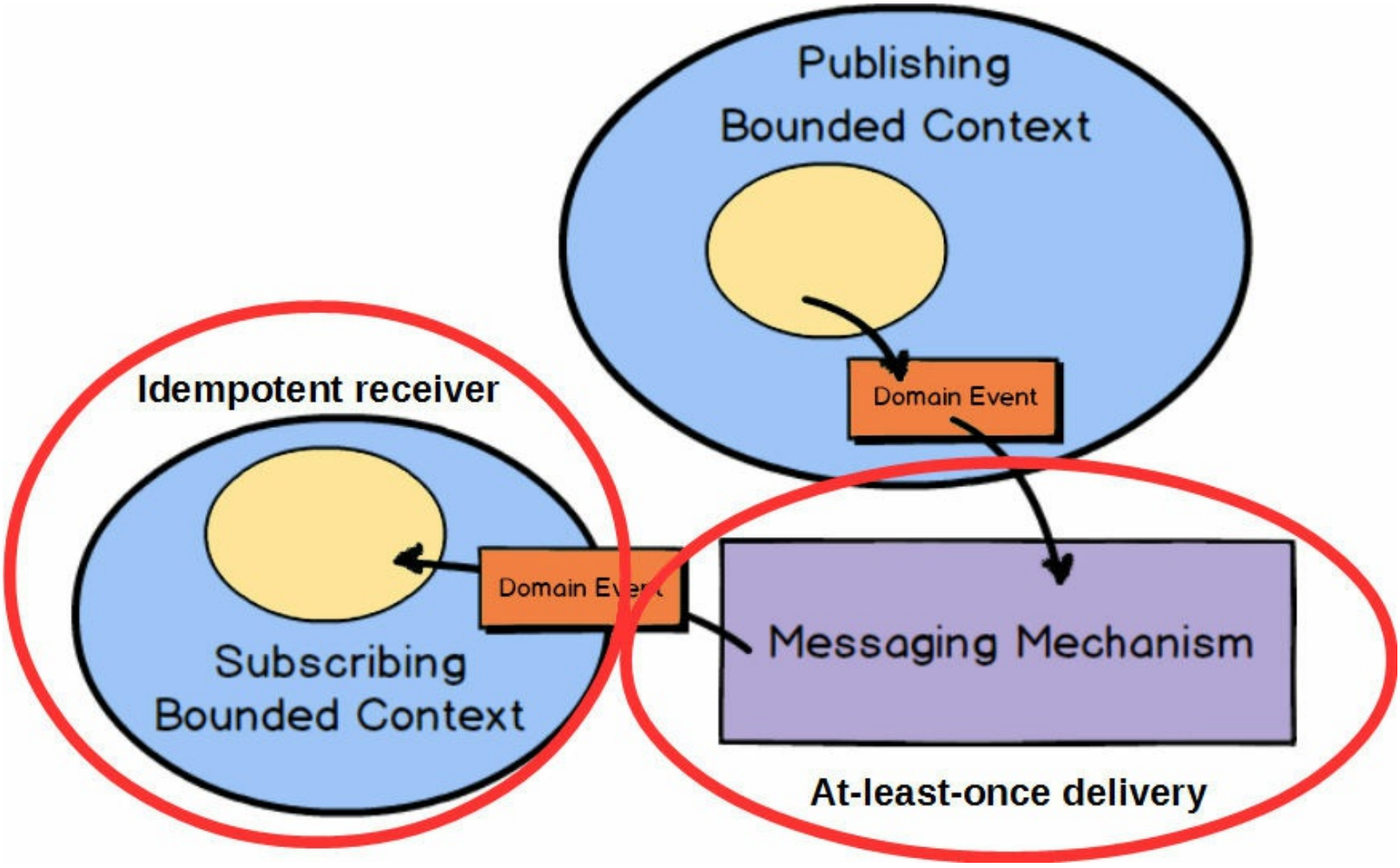
Typically an *Aggregate* in one *Bounded Context* publishes a *Domain Event* , which could be consumed by any number of interested parties. When a subscribing *Bounded Context* receives the *Domain Event* , some action will be taken based on its type and value. Normally it will cause a new *Aggregate* to be created or an existing *Aggregate* to be modified in the consuming *Bounded Context*.

Are Domain Event Consumers Conformists?

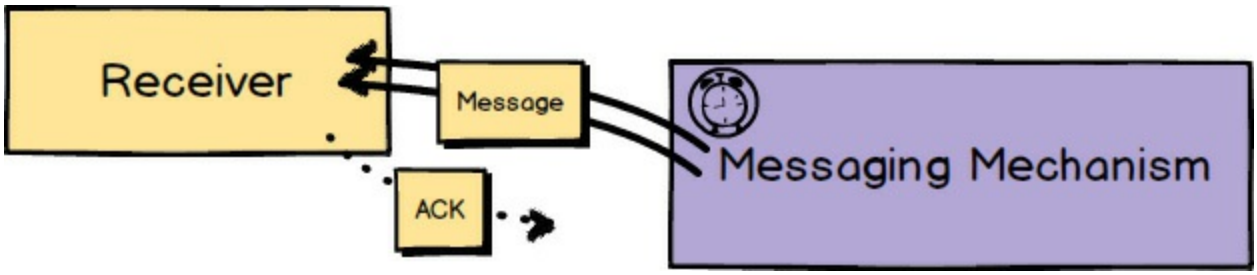
You may be wondering how *Domain Events* can be consumed by another *Bounded Context* and not force that consuming *Bounded Context* into a *Conformist* relationship. As recommended in *Implementing Domain-Driven Design* [IDDD] , and specifically in Chapter 13, "Integrating Bounded Contexts," consumers should not use the event types (e.g., classes) of an event publisher. Rather, they should depend only on the schema of the events, that is, their *Published Language*. This generally means that if the events are published as JSON, or perhaps a more economical object format, the consumer should consume the events by parsing them to obtain their data attributes.



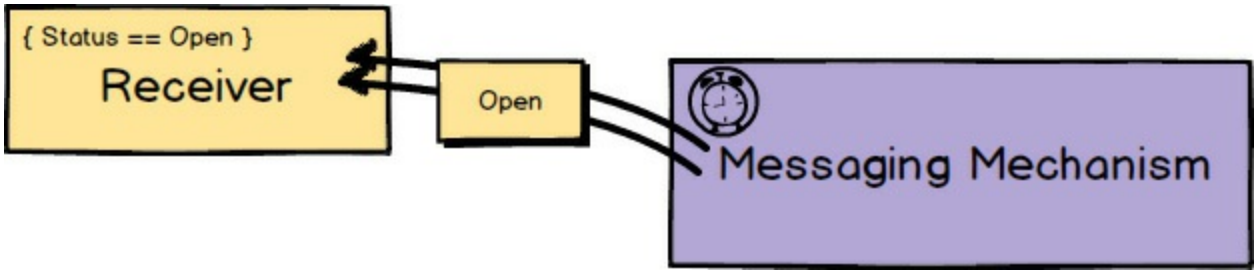
Of course, the foregoing assumes that a subscribing *Bounded Context* can always benefit from unsolicited happenings in the publishing *Bounded Context*. Sometimes, however, a client *Bounded Context* will need to proactively send a *Command Message* to a service *Bounded Context* to force some action. In such cases the client *Bounded Context* will still receive any outcome as a published *Domain Event*.



In all cases of using messaging for integration, the quality of the overall solution will depend heavily on the quality of the chosen messaging mechanism. The messaging mechanism should support *At-Least-Once Delivery* [Reactive] to ensure that all messages will be received eventually. This also means that the subscribing *Bounded Context* must be implemented as an *Idempotent Receiver* [Reactive] .

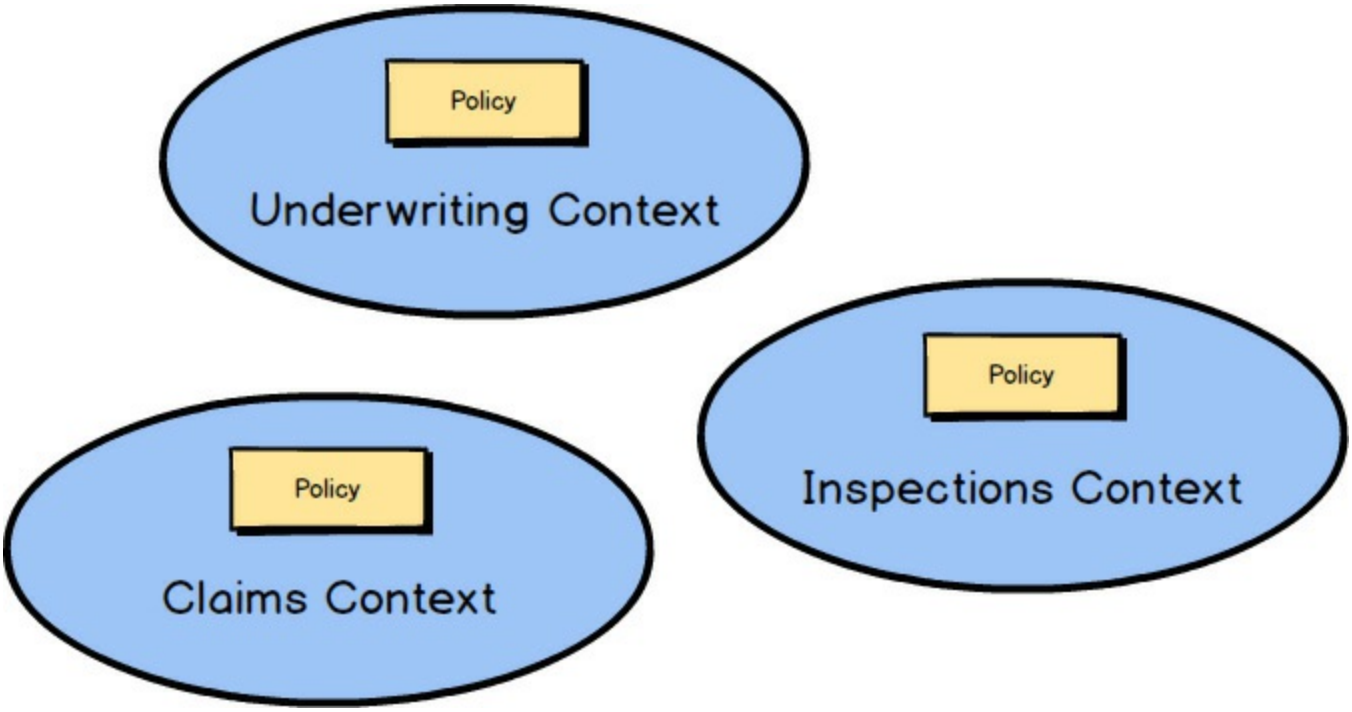


At-Least-Once Delivery [Reactive] is a messaging pattern where the messaging mechanism will periodically redeliver a given message. This will be done in cases of message loss, slow-reacting or downed receivers, and receivers failing to acknowledge receipt. Because of this messaging mechanism design, it is possible for the message to be delivered more than once even though the sender sends it only once. Still, that needn't be a problem when the receiver is designed to deal with this.



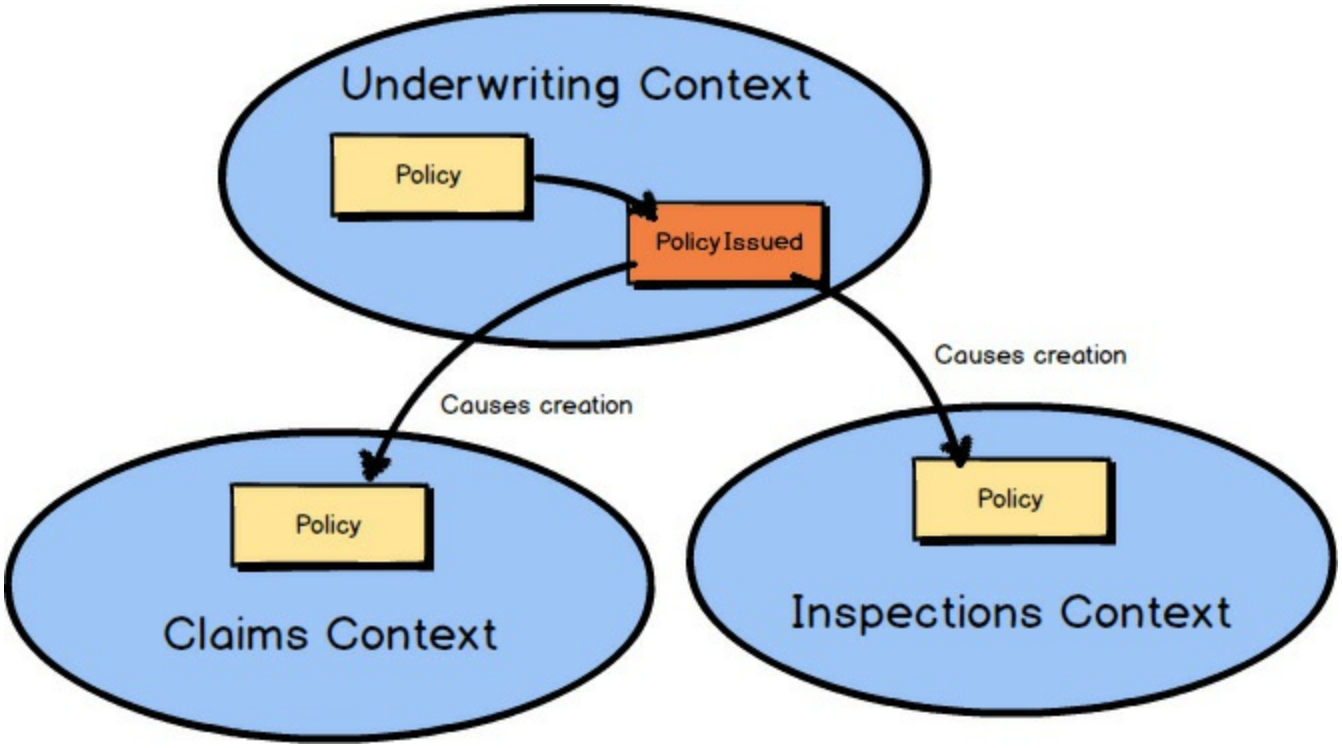
Whenever a message could be delivered more than once, the receiver should be designed to deal correctly with this situation. *Idempotent Receiver* [Reactive] describes how the receiver of a request performs an operation in such a way that it produces the same result even if it is performed multiple times. Thus, if the same message is received multiple times, the receiver will deal with it in a safe manner. This can mean that the receiver uses de-duplication and ignores the repeated message, or safely reapplies the operation with the exact same results that the previous delivery caused.

Due to the fact that messaging mechanisms always introduce asynchronous *Request-Response* [Reactive] communications, some amount of latency is both common and expected. Requests for service should (almost) never block until the service is fulfilled. Thus, designing with messaging in mind means that you will always plan for at least some latency, which will make your overall solution much more robust from the outset.

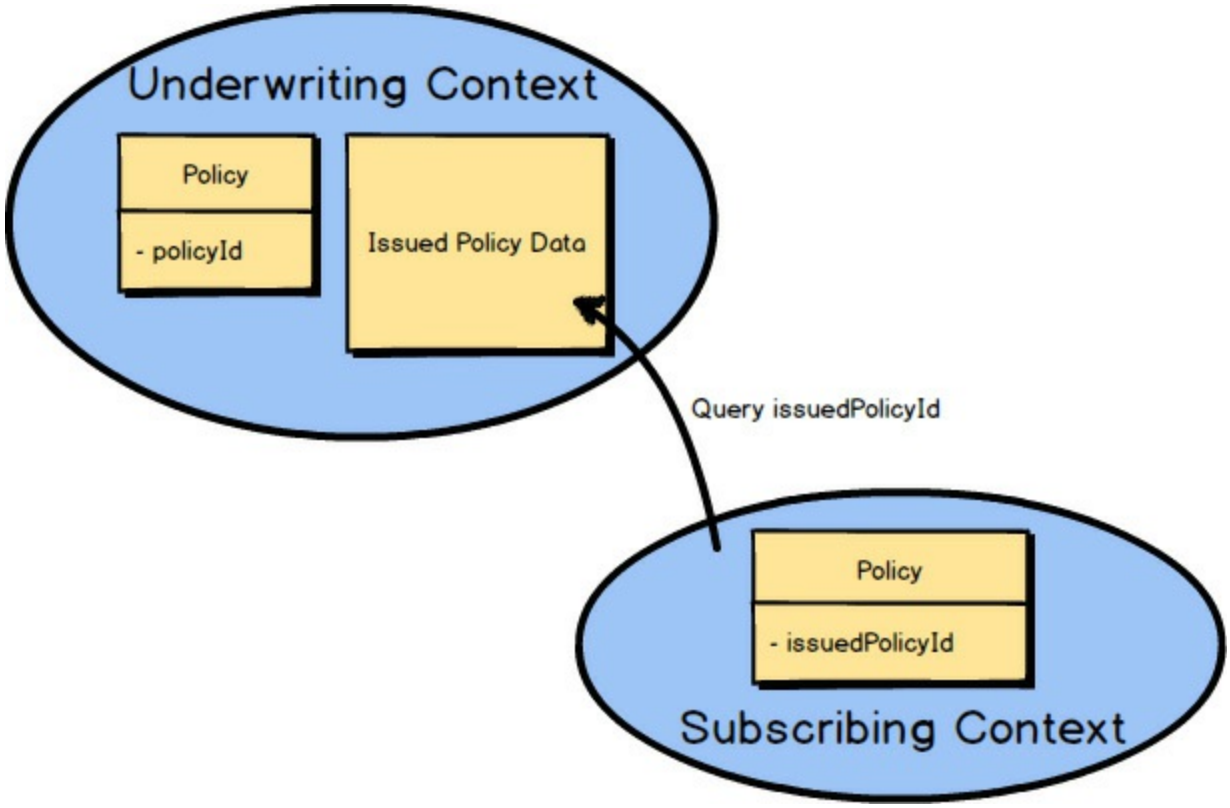


An Example in Context Mapping

Returning to an example discussed in Chapter 2 , "Strategic Design with Bounded Contexts and the Ubiquitous Language ," a question arises about the location of the official `Policy` type. Remember that there are three different `Policy` types in three different *Bounded Contexts*. So, where does the "policy of record" live in the insurance enterprise? It's possible that it belongs to the underwriting division since that is where it originates. For the sake of this example, let's say it does belong to the underwriting division. So how do the other *Bounded Contexts* learn about its existence?



When a component of type `Policy` is issued in the *Underwriting Context* , it could publish a *Domain Event* named `PolicyIssued` . Provided through a messaging subscription, any other *Bounded Context* may react to that *Domain Event* , which could include creating a corresponding `Policy` component in the subscribing *Bounded Context*.



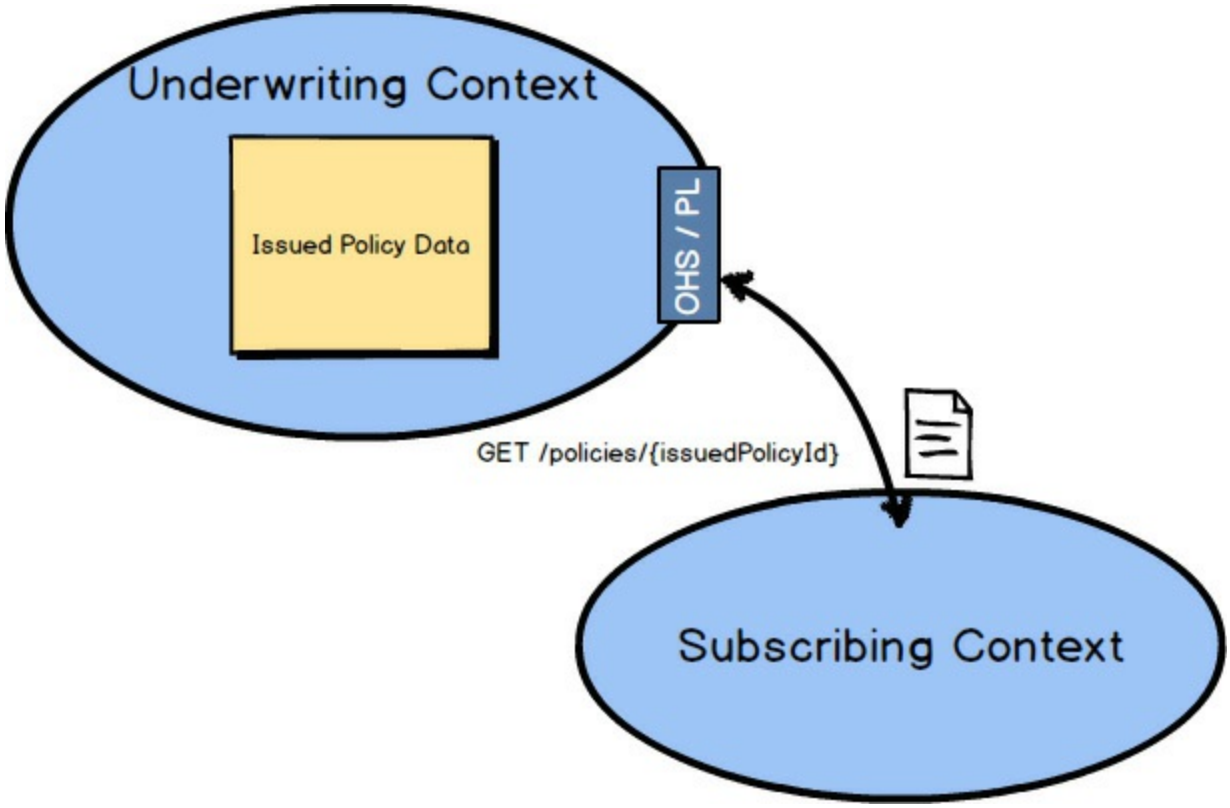
The `PolicyIssued` *Domain Event* would contain the identity of the official `Policy` . Here it's `policyId` . Any components created in a subscribing *Bounded Context* would retain that identity for traceability back to the originating *Underwriting Context.* In this example the identity is saved as `issuedPolicyId` . If there is a need for more `Policy` data than the `PolicyIssued` *Domain Event* provided, the subscribing *Bounded Context* can always query back on the *Underwriting Context* for more information. Here the subscribing *Bounded Context* uses the `issuedPolicyId` to perform a query on the *Underwriting Context.*

Enrichment versus Query-Back Trade-offs

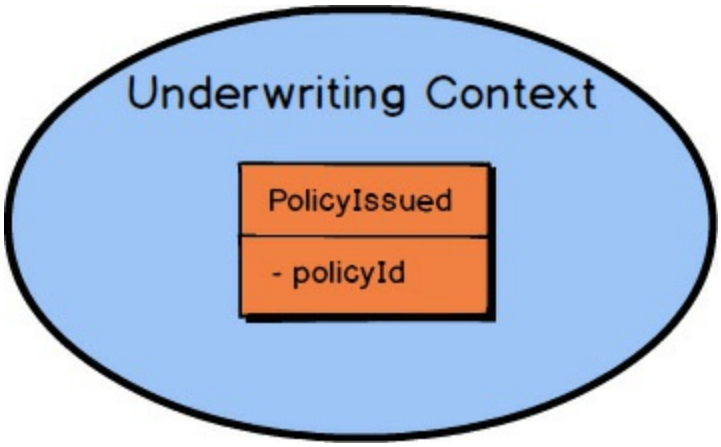
Sometimes there is an advantage to enriching *Domain Events* with enough data to satisfy the needs of all consumers. Sometimes there is an advantage to keeping *Domain Events* thin and allowing for querying back when consumers need more data. The first choice, enrichment, allows for greater autonomy of dependent consumers. If autonomy is your driving requirement, consider enrichment.

On the other hand, it is difficult to predict every piece of data that all consumers will ever need in *Domain Events* , and there may be too much enrichment if you provide it all. For example, it may be a poor security choice to greatly enrich *Domain Events.* If that is the case, designing thin *Domain Events* and a rich query model with security for consumers to request from may be the choice you need.

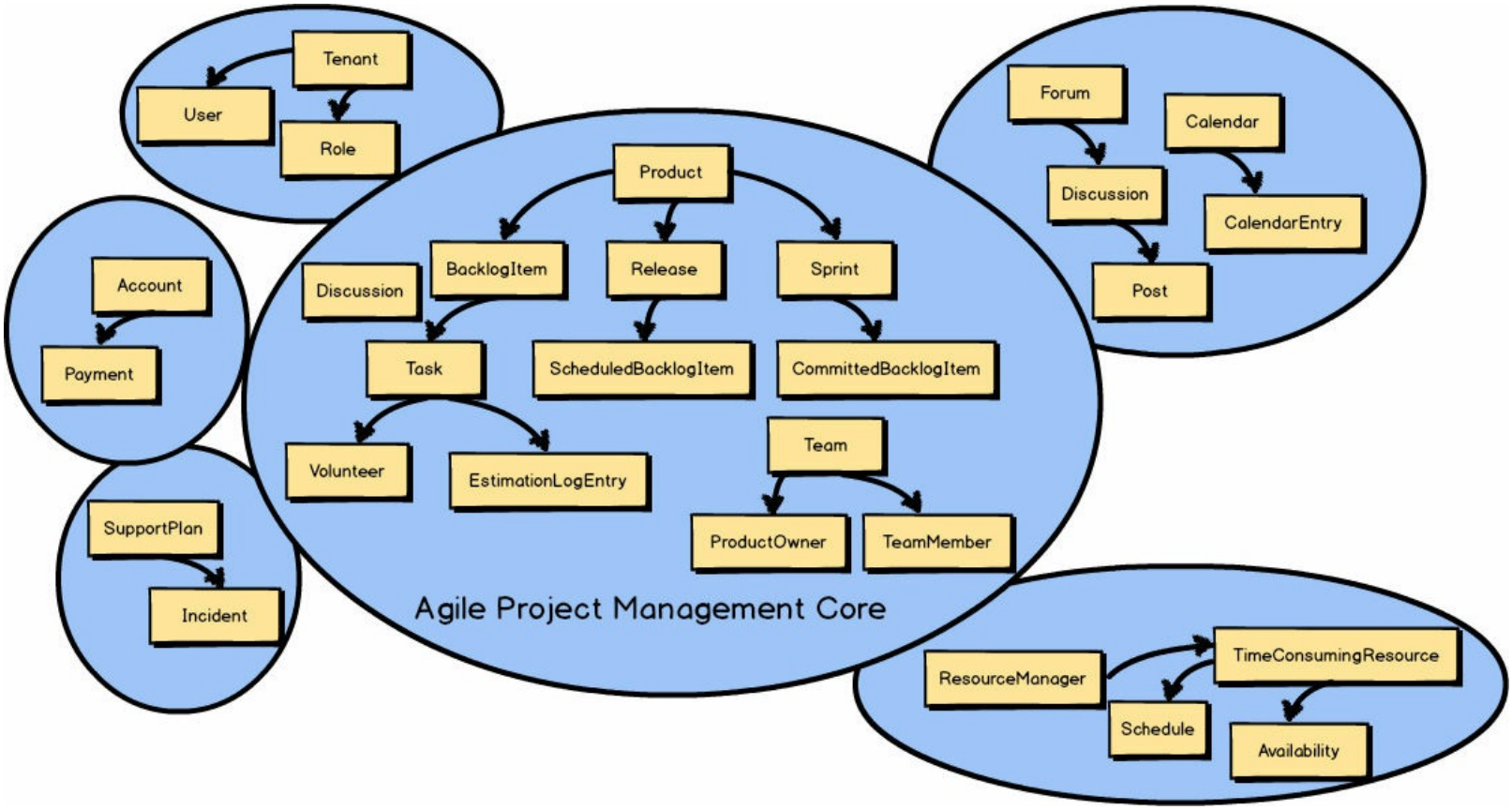
Sometimes circumstances will call for a balanced blend of both approaches.



And how might the query back on the *Underwriting Context* work? You could design a RESTful *Open Host Service* and *Published Language* on the *Underwriting Context*. A simple HTTP GET with the `issuedPolicyId` would retrieve the `IssuedPolicyData` .



You're probably wondering about the data details of the `Policy-Issued` *Domain Event*. I will provide *Domain Event* design details in Chapter 6 , "Tactical Design with Domain Events ."



Are you curious about what happened to the *Agile Project Management Context* example? Straying from that to the insurance business domain allowed you to examine DDD with multiple examples. That should have helped you grasp DDD even better. Don't worry, we will return to the *Agile Project Management Context* in the next chapter.

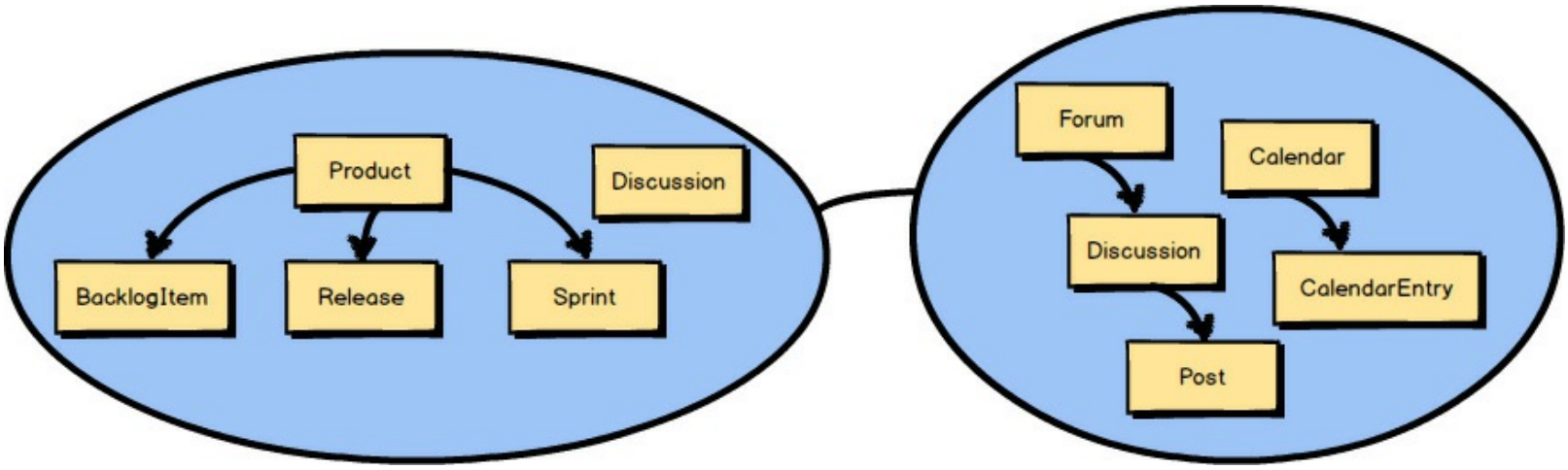
Summary

In summary, you have learned:

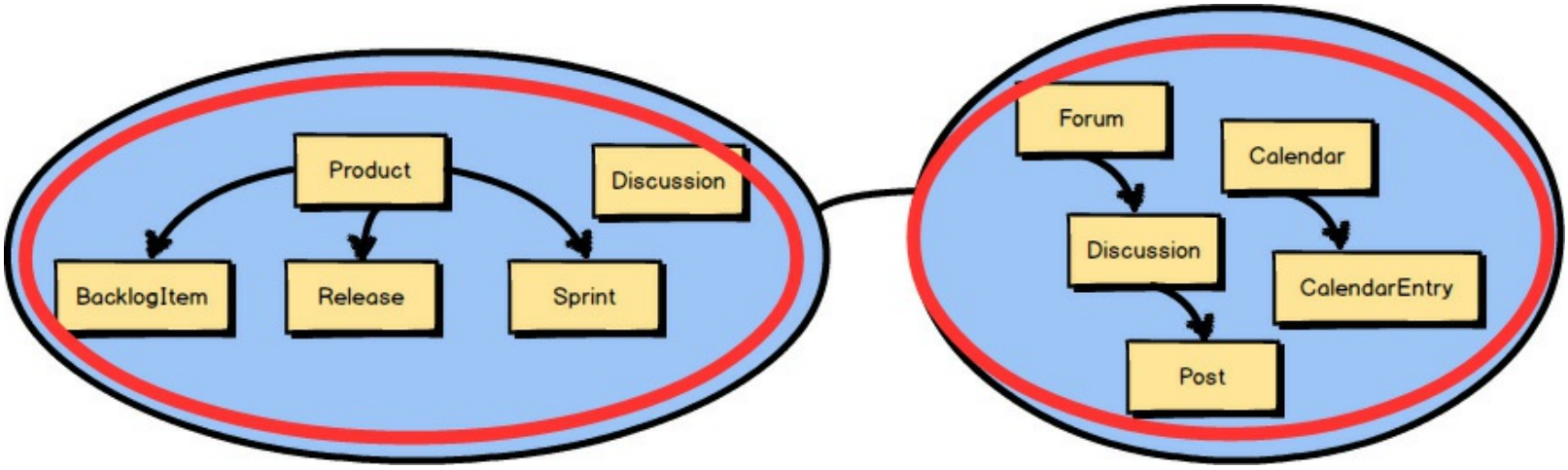
- About the various kinds of *Context Mapping* relationships, such as *Partnership* , *Customer-Supplier* , and *Anticorruption Layer*
- How to use *Context Mapping* integration with RPC, with RESTful HTTP, and with messaging
- How *Domain Events* work with messaging
- A foundation on which you can build your *Context Mapping* experience

For thorough coverage of *Context Maps* , see Chapter 3 of *Implementing Domain-Driven Design* [IDDD] .

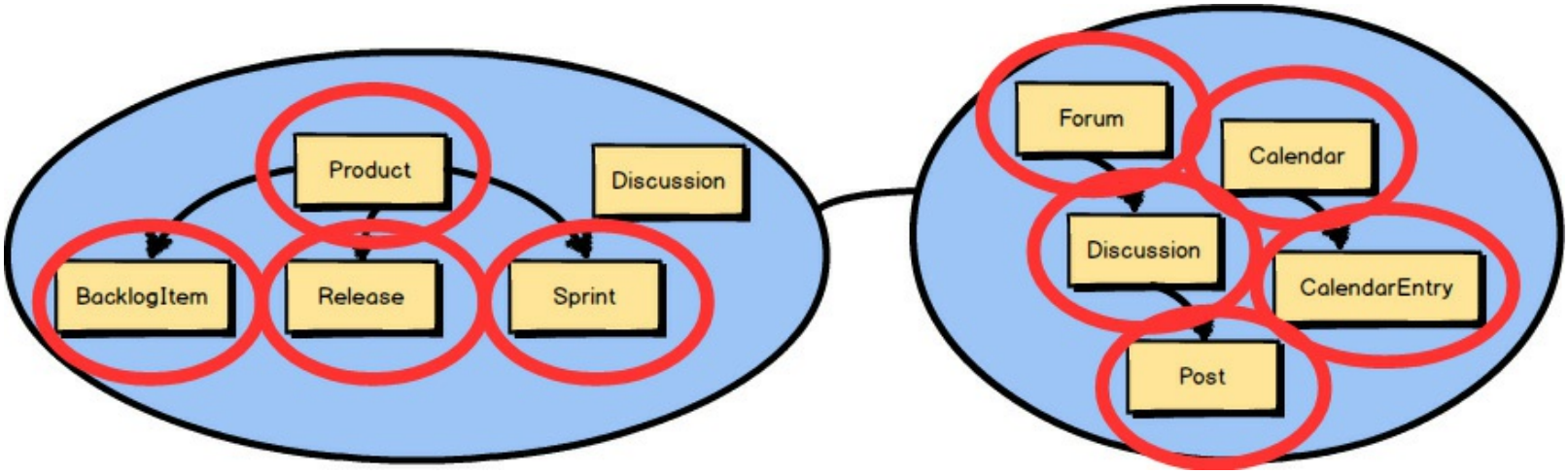
Chapter 5. Tactical Design with Aggregates



So far I have discussed strategic design with *Bounded Contexts* , *Subdomains* , and *Context Maps*. Here you see two *Bounded Contexts* , the *Core Domain* named *Agile Project Management Context* and a *Supporting Subdomain* that provides collaboration tools through *Context Mapping* integration.



But what about the concepts that live inside a *Bounded Context* ? I've touched on these, but I will next cover them in more detail. They are likely the *Aggregates* in your model.



Why Used

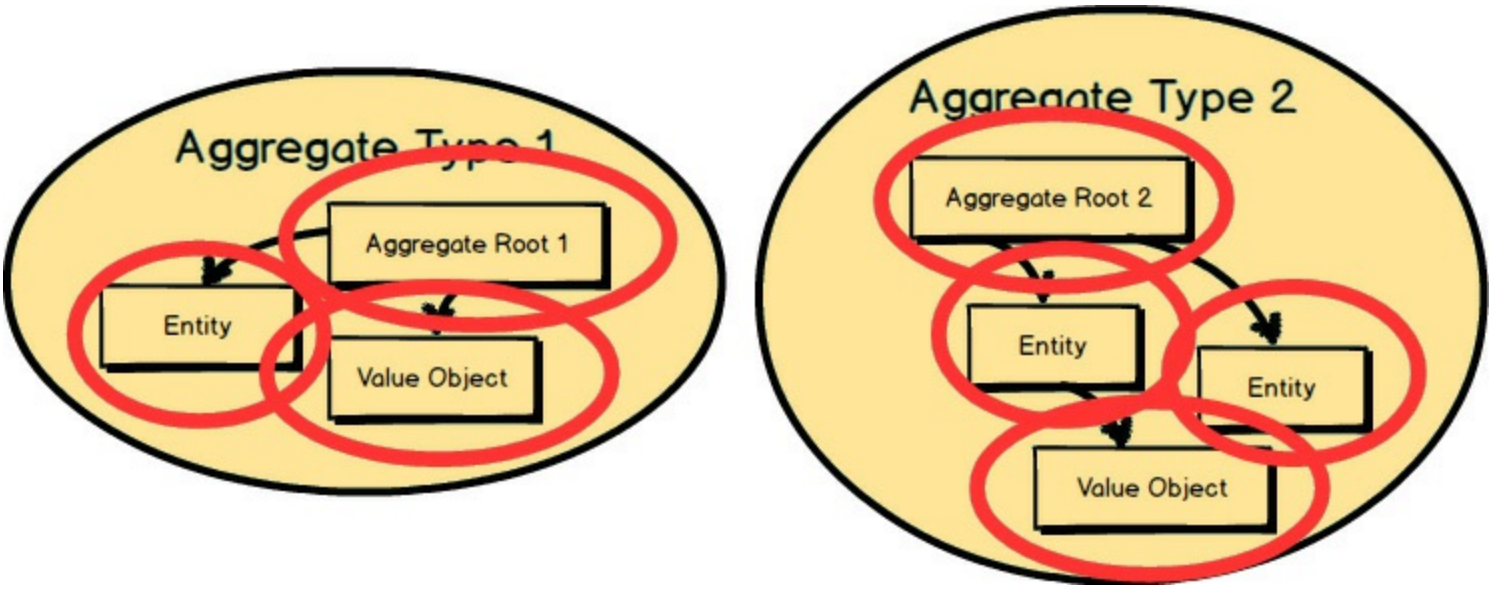
Each of the circled concepts that you see inside these two *Bounded Contexts* is an *Aggregate*. The one concept not circled—`Discussion` —is modeled as a *Value Object*. Even so, we are focused on *Aggregates* in this chapter, and we will take a closer look at how to model `Product` ,

`BacklogItem`, `Release`, and `Sprint`.

What Is an Entity?

An *Entity* models an individual thing. Each *Entity* has a unique identity in that you can distinguish its individuality from among all other *Entities* of the same or a different type. Many times, perhaps even most times, an *Entity* will be mutable; that is, its state will change over time. Still, an *Entity* is not of necessity mutable and may be immutable. The main thing that separates an *Entity* from other modeling tools is its uniqueness—its individuality.

See *Implementing Domain-Driven Design* [IDDD] for an exhaustive treatment of *Entities*.

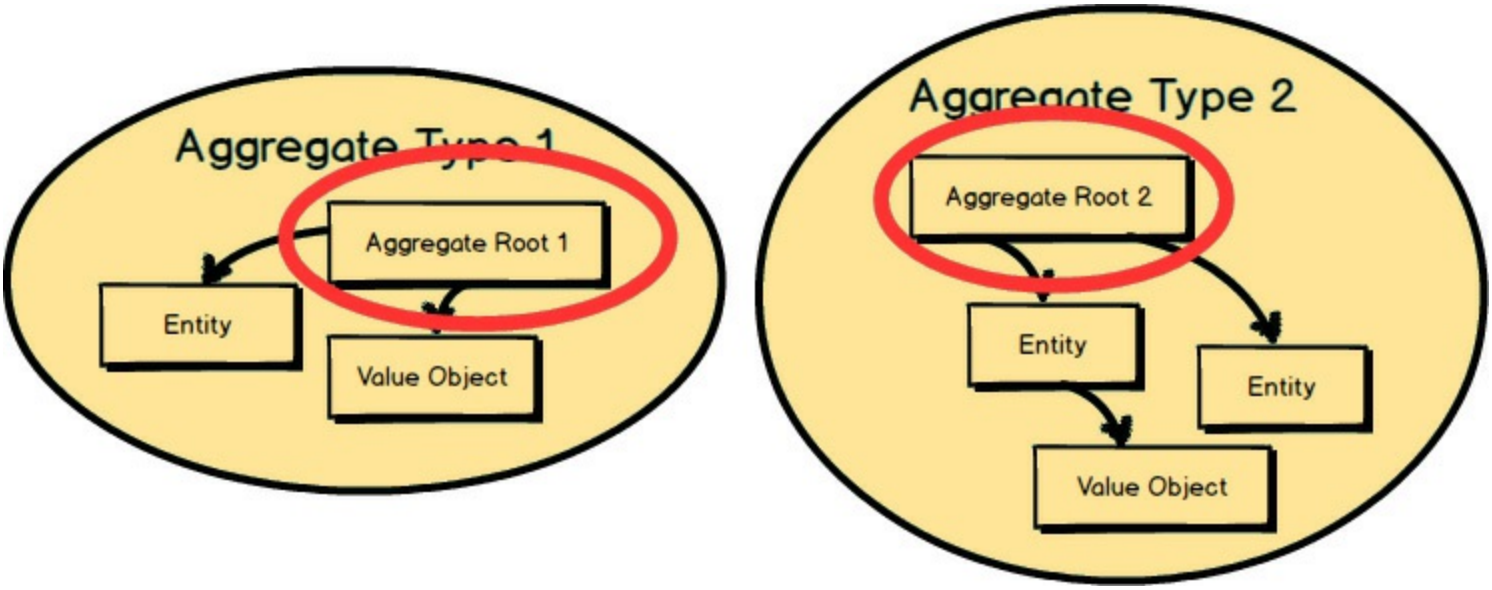


What is an *Aggregate* ? Two are represented here. Each *Aggregate* is composed of one or more *Entities* , where one *Entity* is called the *Aggregate Root*. *Aggregates* may also have *Value Objects* composed on them. As you see here, *Value Objects* are used inside both *Aggregates* .

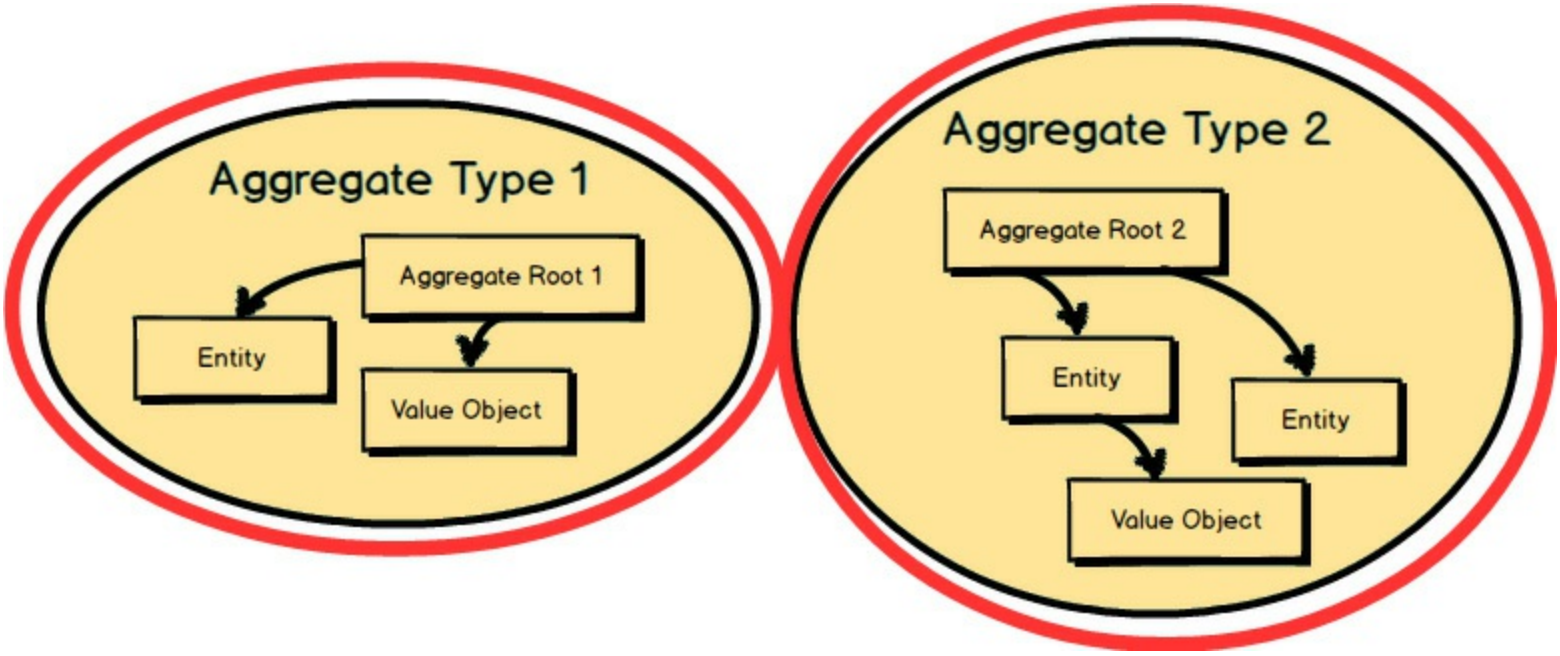
What Is a Value Object?

A *Value Object* , or simply a *Value* , models an immutable conceptual whole. Within the model the *Value* is just that, a value. Unlike an *Entity* , it does not have a unique identity, and equivalence is determined by comparing the attributes encapsulated by the *Value* type. Furthermore, a *Value Object* is not a thing but is often used to describe, quantify, or measure an *Entity*.

See *Implementing Domain-Driven Design* [IDDD] for detailed coverage of *Value Objects*.



The *Root Entity* of each *Aggregate* owns all the other elements clustered inside it. The name of the *Root Entity* is the *Aggregate* 's conceptual name. You should choose a name that properly describes the conceptual whole that the *Aggregate* models.



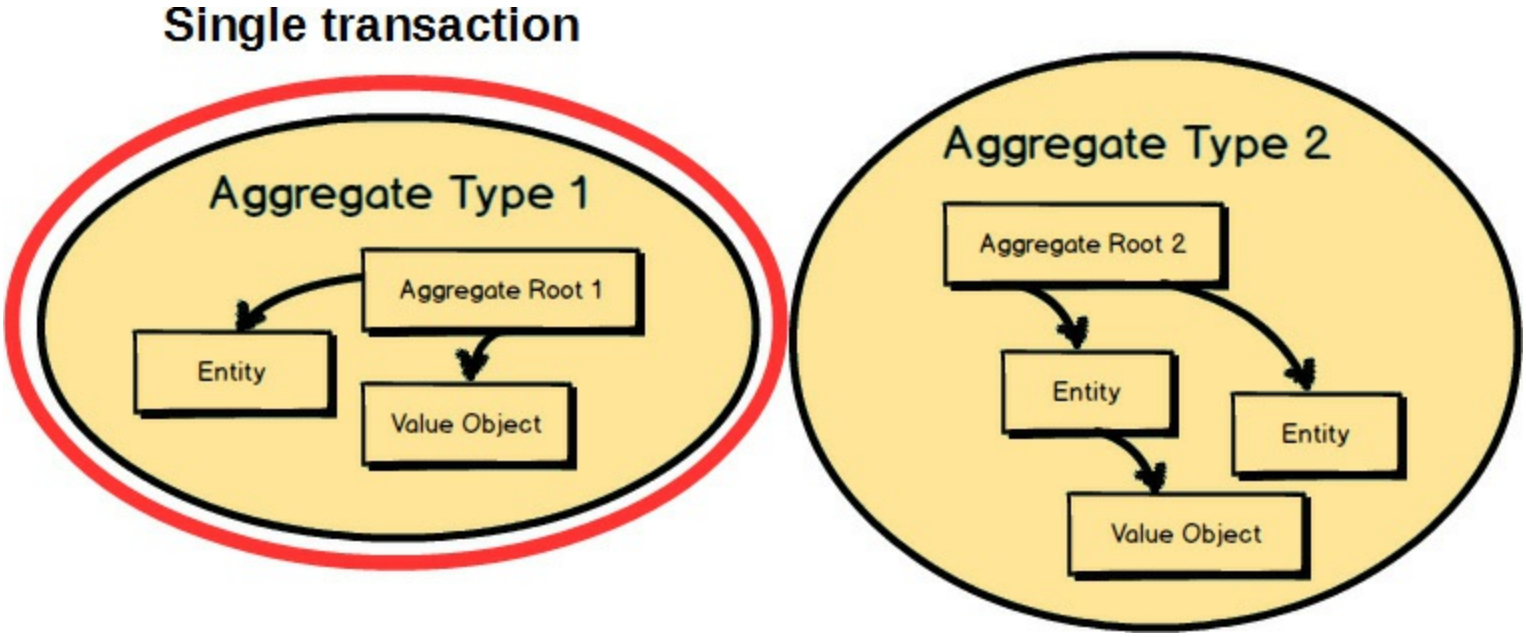
Each *Aggregate* forms a transactional consistency boundary. This means that within a single *Aggregate* , all composed parts must be consistent, according to business rules, when the controlling transaction is committed to the database. This doesn't necessarily mean that you are not supposed to compose other elements within an *Aggregate* that don't need to be consistent after a transaction. After all, an *Aggregate* also models a conceptual whole. But you should be first and foremost concerned with transactional consistency. The outer boundary drawn around `Aggregate Type 1` and `Aggregate Type 2` represents a separate transaction that will be in control of atomically persisting each object cluster.

Broader Meaning of Transaction

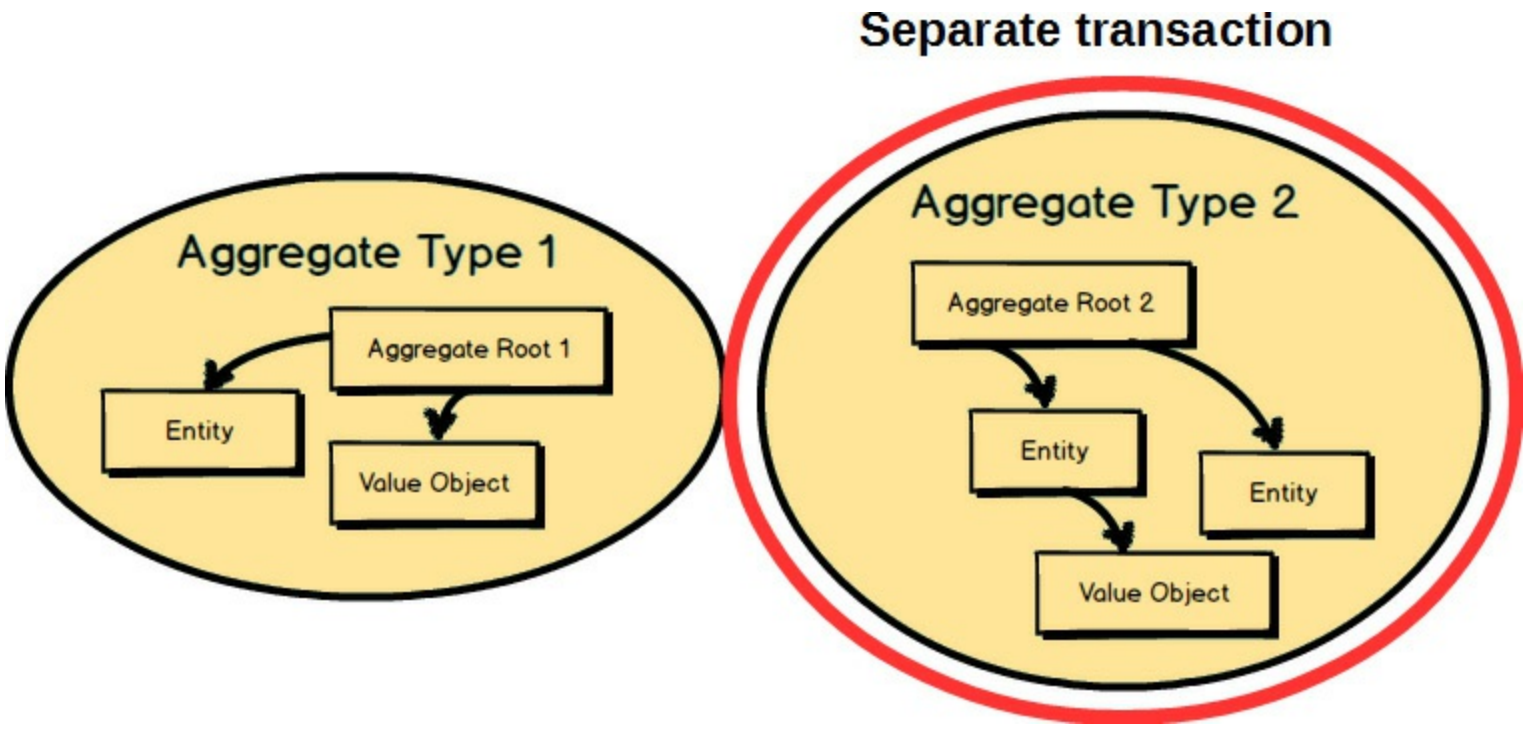
To some degree, the use of transactions in your application is an implementation detail. For example, a typical use would have an Application Service [IDDD] controlling the atomic database transaction on behalf of the domain model. Under a different architecture, such as the Actor model [Reactive] where each *Aggregate* is implemented

as an actor, transactions could be handled using *Event Sourcing* (see the next chapter) with a database that doesn't support atomic transactions. Either way, what I mean by "transaction" is how modifications to an *Aggregate* are isolated and how business invariants—the rules to which the software must always adhere—are guaranteed to be consistent following each business operation. Whether this requirement is controlled by an atomic database transaction or by some other means, the *Aggregate* 's state, or its representation by means of *Event Sourcing* , must be safely and correctly transitioned and maintained at all times.

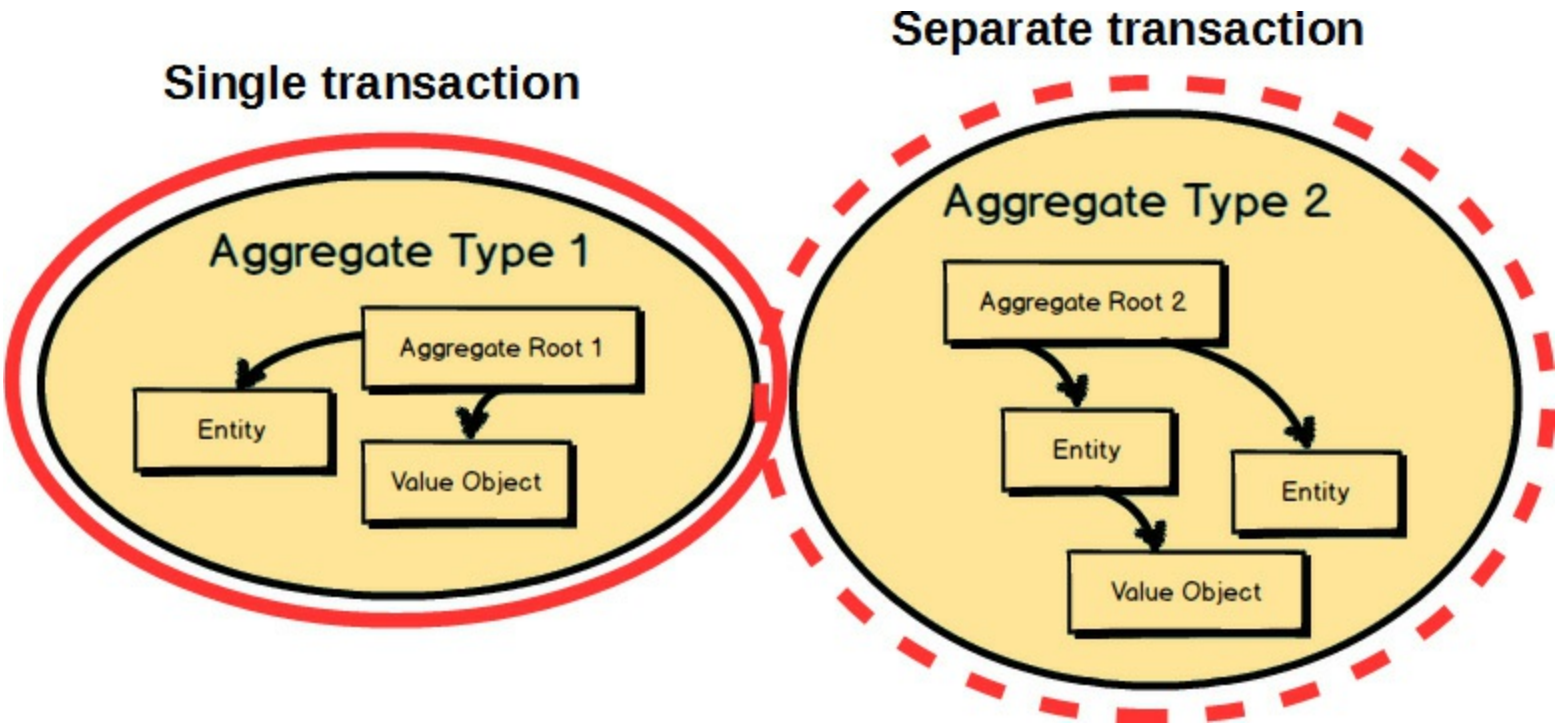
The reasons for the transactional boundary are business motivated, because it is the business that determines what a valid state of the cluster should be at any given time. In other words, if the *Aggregate* was not stored in a whole and valid state, the business operation that was performed would be considered incorrect according to business rules.



To think about this in a different way, consider this. Although two *Aggregates* are represented here, only one of the two should be committed in a single transaction. That's a general rule of *Aggregate* design: modify and commit only one *Aggregate* instance in one transaction. That's why you see only the instance of `Aggregate Type 1` within a transaction. We will look at the other rules of *Aggregate* design soon.



Any other *Aggregate* will be modified and committed in a separate transaction. That's why an *Aggregate* is said to be a transactional consistency boundary. So, you design your *Aggregate* compositions in a way that allows for transactional consistency and success. As seen here, an instance of `Aggregate Type 2` is controlled under a separate transaction from the instance of `Aggregate Type 1`.



Since instances of these two *Aggregates* are designed to be modified in separate transactions, how do we get the instance of `Aggregate Type 2` updated based on changes made to the instance of `Aggregate Type 1`, to which our domain model must react? That's a good question; we will consider the answer to it a bit later in this chapter.

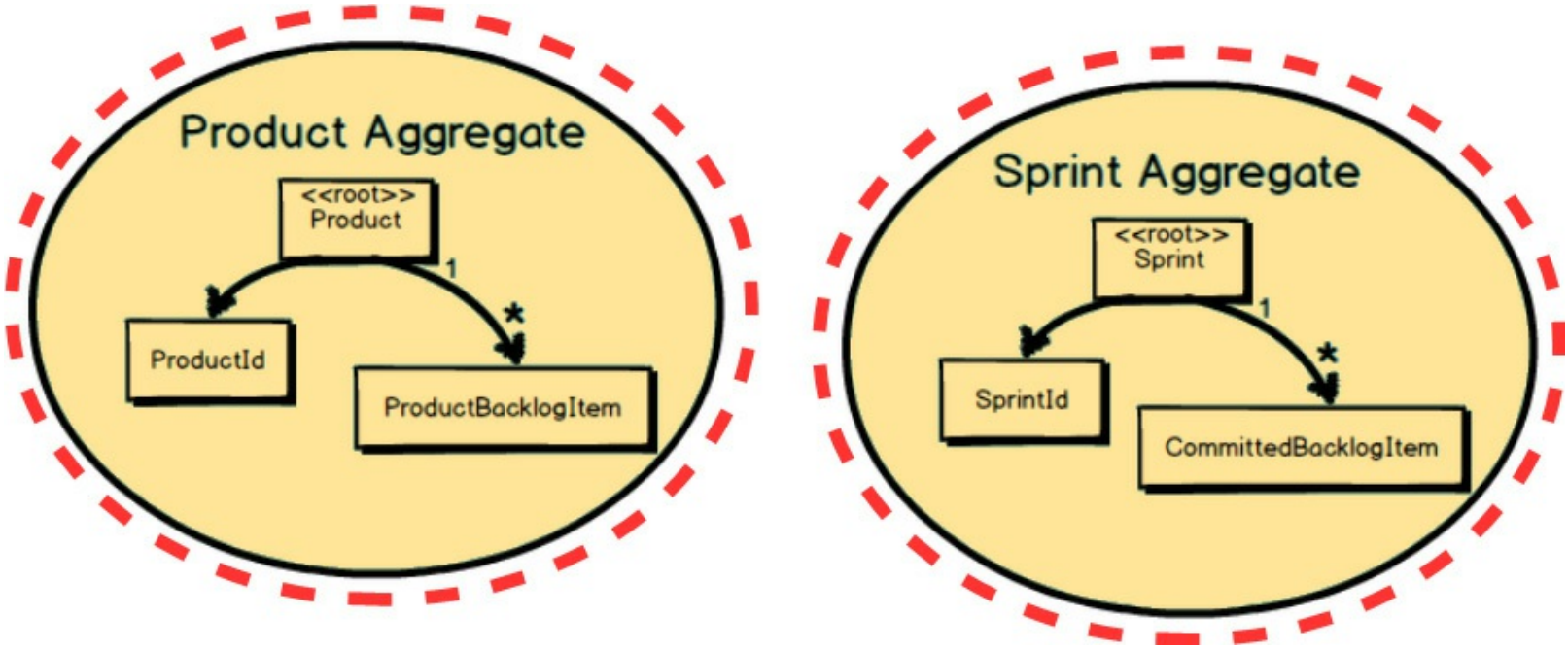
The main point to remember from this section is that business rules are the drivers for determining what must be whole, complete, and consistent at the end of a single transaction.

Aggregate Rules of Thumb

Let's next consider the four basic rules of *Aggregate* design:

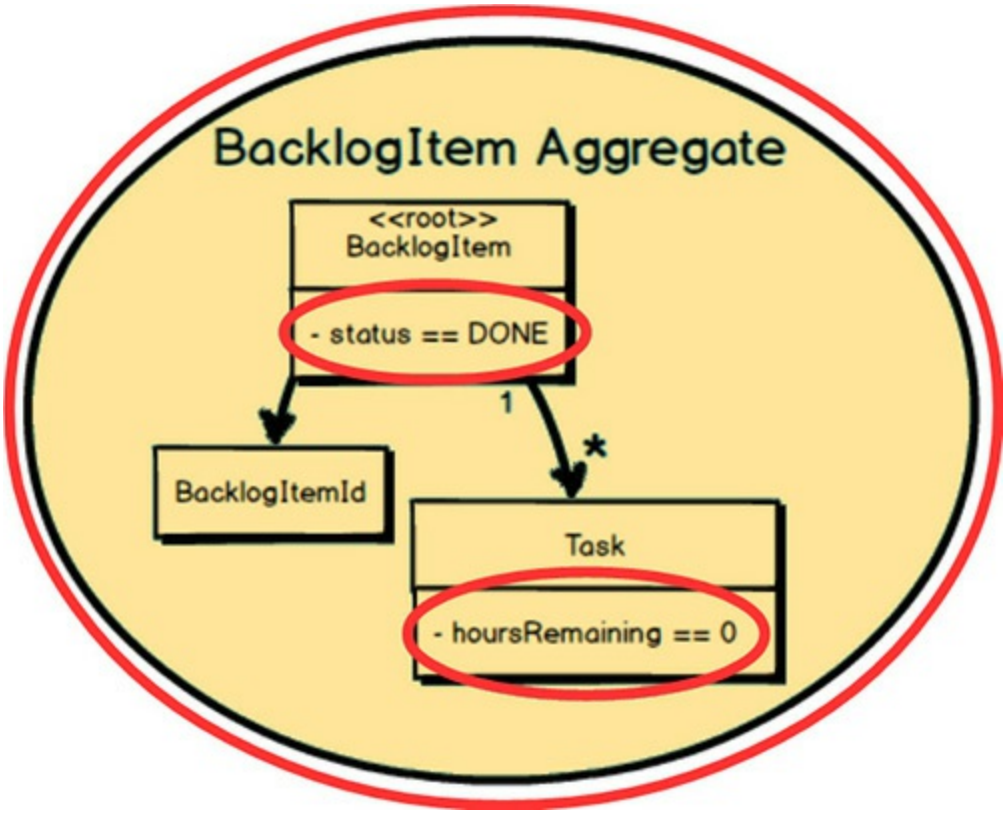
1. Protect business invariants inside *Aggregate* boundaries.
2. Design small *Aggregates*.
3. Reference other *Aggregates* by identity only.
4. Update other *Aggregates* using eventual consistency.

Of course, these rules are not necessarily strictly enforced by any "DDD police." They are meant as sound guidance such that when thoughtfully applied, they will help you design *Aggregates* that work effectively. That being the case, we will now dig into each of these rules to see how they should be applied wherever possible.

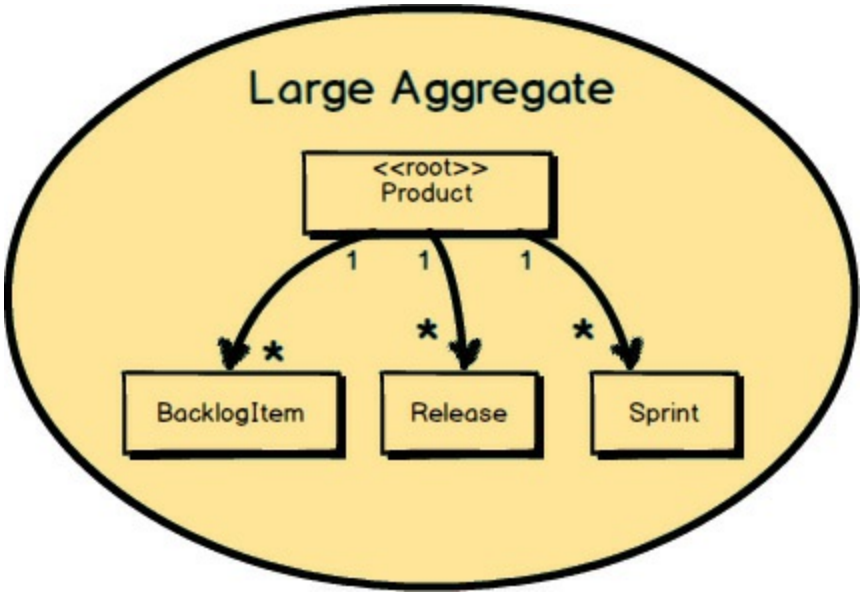


Rule 1: Protect Business Invariants inside Aggregate Boundaries

Rule 1 means that the business should ultimately determine *Aggregate* compositions based on what must be consistent when a transaction is committed. In the example on page 81 , `Product` is designed such that at the end of a transaction all composed `ProductBacklogItem` instances must be accounted for and consistent with the `Product` root. Also, `Sprint` is designed such that at the end of a transaction all composed `CommittedBacklogItem` instances must be accounted for and consistent with the `Sprint` root.

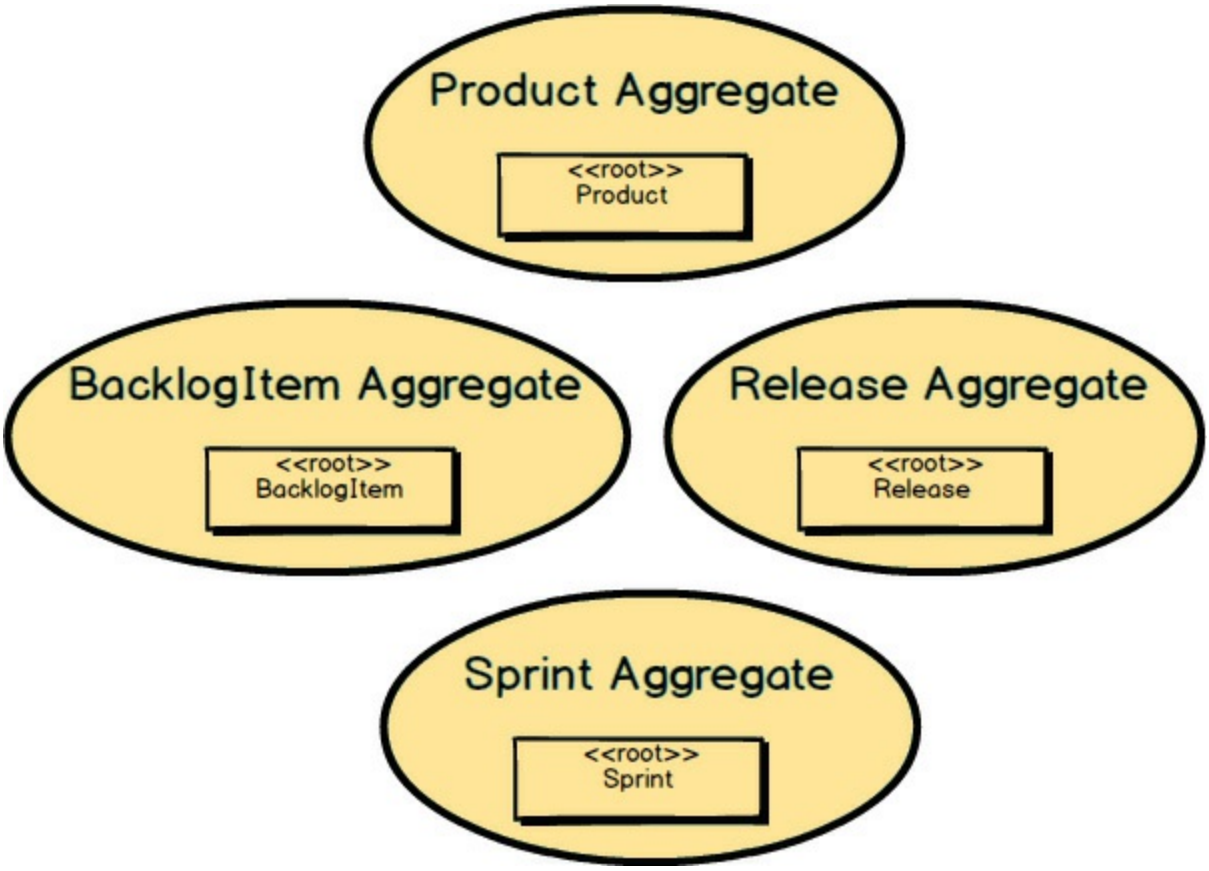


Rule 1 becomes clearer with another example. Here's the `BacklogItem` *Aggregate*. There is a business rule that states, "When all `Task` instances have `hoursRemaining` of zero, the `BacklogItem` status must be set to `DONE` ." Thus, at the end of a transaction this very specific business invariant must be met. The business requires it.



Rule 2: Design Small Aggregates

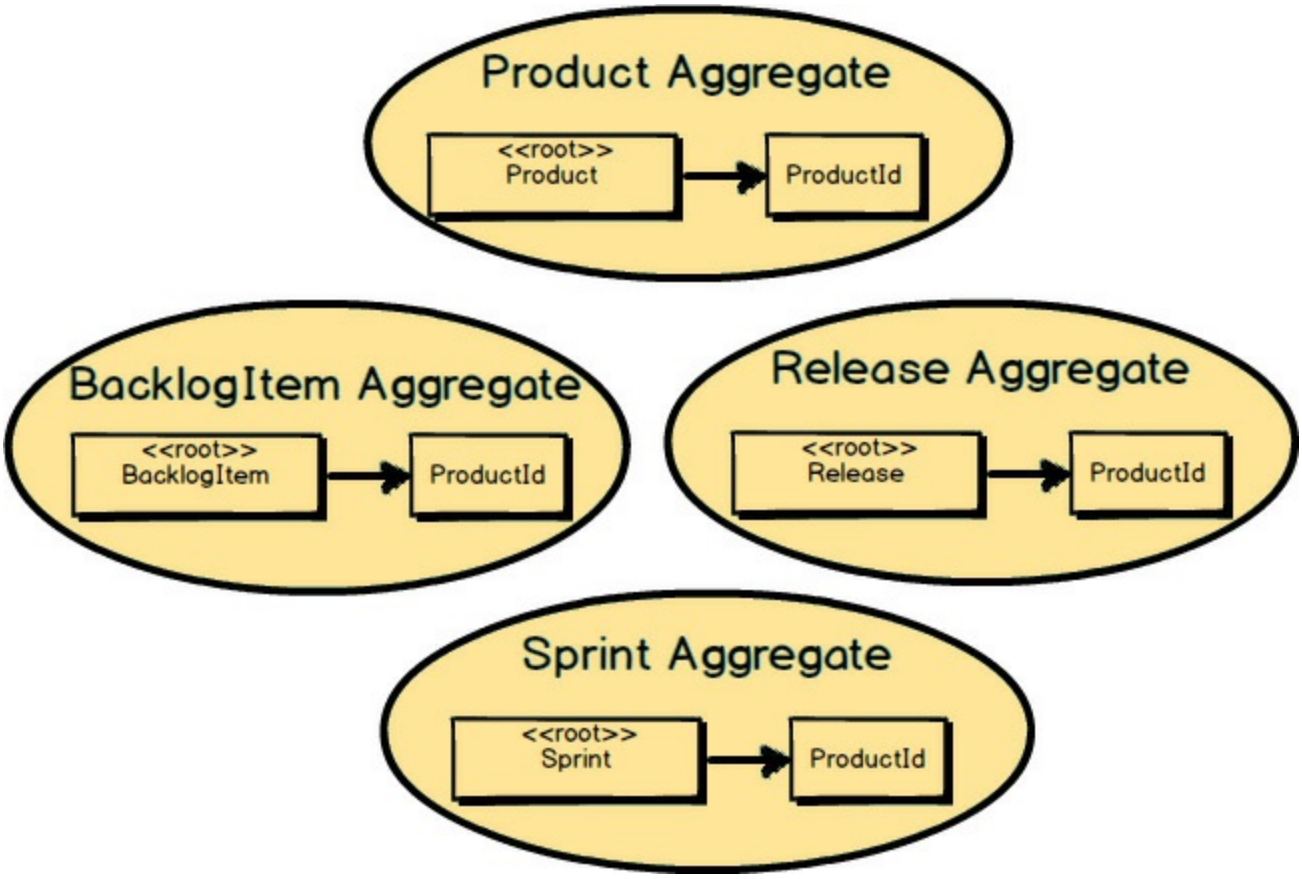
This rule highlights that the memory footprint and transactional scope of each *Aggregate* should be relatively small. In the preceding diagram the *Aggregate* that is represented is not small. Here, `Product` literally contains a potentially very large collection of `BacklogItem` instances, a large collection of `Release` instances, and a large collection of `Sprint` instances. Over time, these collections could grow to be quite large, with thousands of `BacklogItem` instances and probably hundreds of `Release` and `Sprint` instances. This design approach is generally a very poor choice.



However, if we break up the `Product` *Aggregate* to form four separate *Aggregates* , this is what we get: a small `Product` *Aggregate* , a small `BacklogItem` *Aggregate* , a small `Release` *Aggregate* , and a small `Sprint` *Aggregate*. These load quickly, take less memory, and are faster to garbage collect. Perhaps most importantly, these *Aggregates* will have transactional success much more frequently than the previous large-cluster `Product` *Aggregate*.

Following this rule has the added benefit that each *Aggregate* will be easier to work on, because each associated task can be managed by a single developer. This also means that the *Aggregate* will be easier to test.

Another thing to keep in mind when designing *Aggregates* is the *Single Responsibility Principle* (SRP). If your *Aggregate* is trying to do too many things, it is not following SRP, and this will likely be telling in its size. Ask yourself, for example, whether your `Product` is a very focused implementation of a Scrum product, or if it is also trying to be other things. What is the reason to change `Product` : to make it a better Scrum product, or to manage backlog items, releases, and sprints? You should change `Product` only in order to make it a better Scrum product.

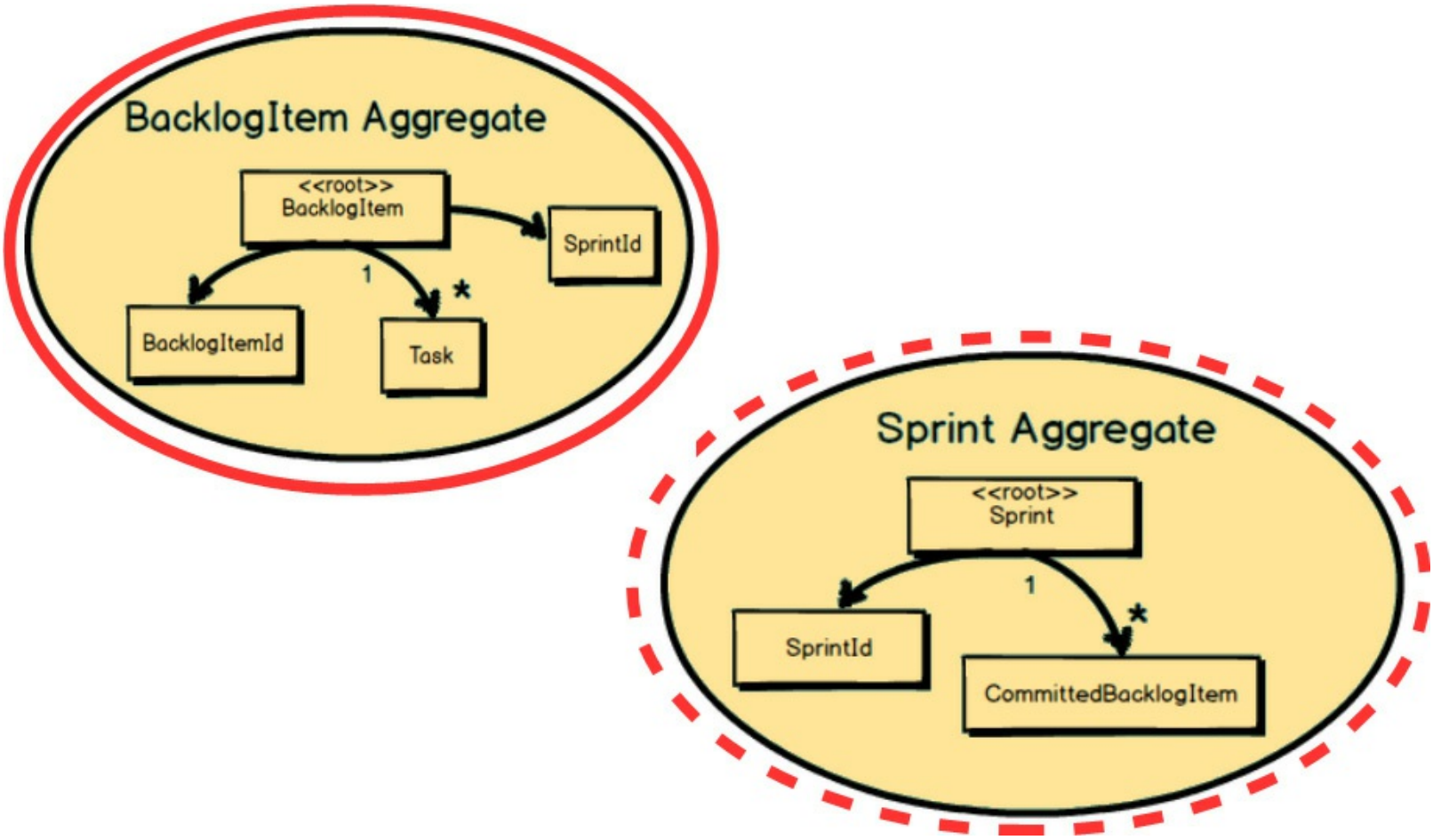


Rule 3: Reference Other Aggregates by Identity Only

Now that we've broken up the large-cluster `Product` into four smaller *Aggregates* , how should each reference the others where needed? Here we follow Rule 3, "Reference other *Aggregates* by identity only." In this example we see that `BacklogItem` , `Release` , and `Sprint` all reference `Product` by holding a `ProductId` . This helps keep *Aggregates* small and prevents reaching out to modify multiple *Aggregates* in the same transaction.

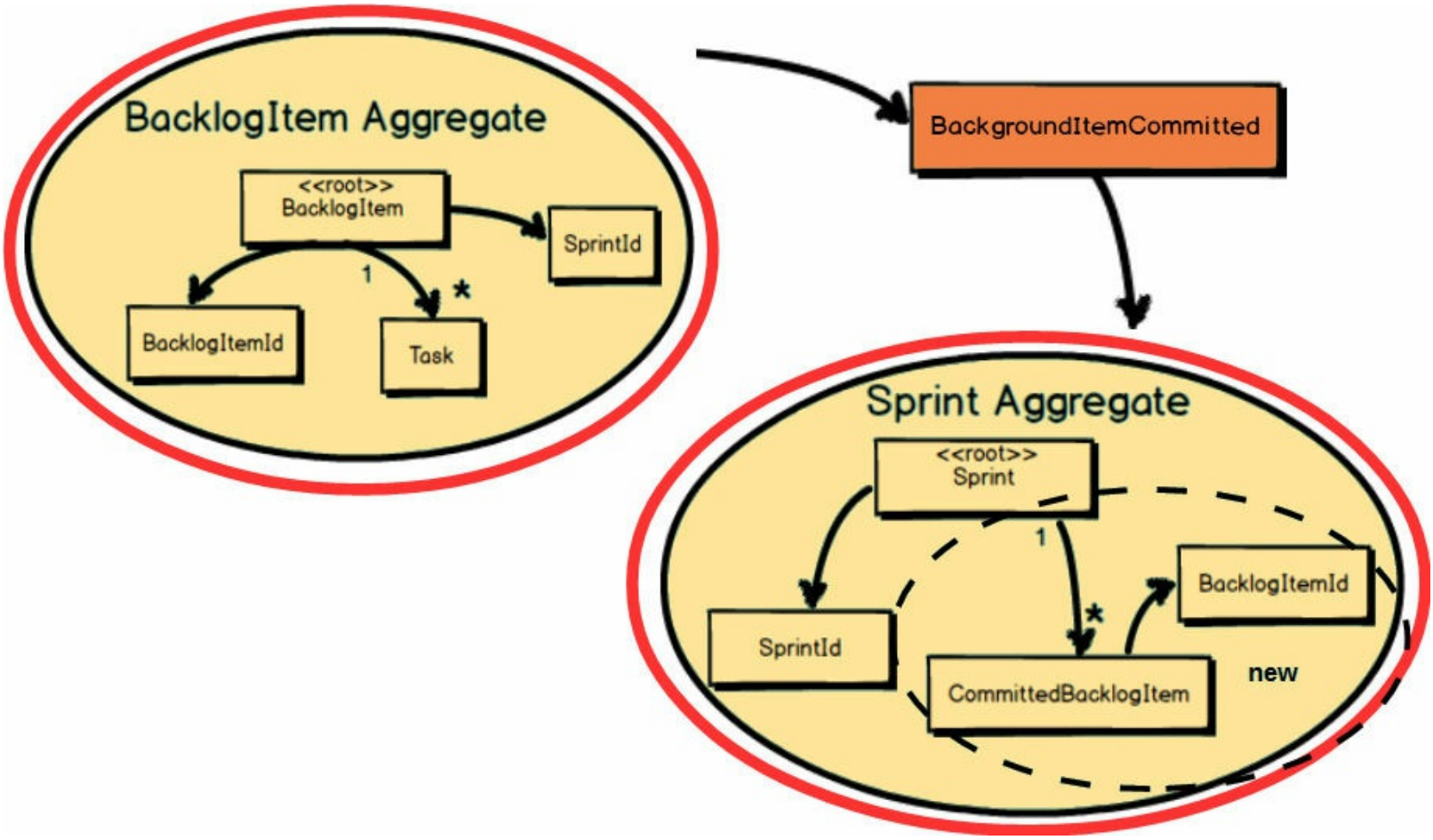
This further helps keep the *Aggregate* design small and efficient, making for lower memory requirements and quicker loading from a persistence store. It also helps enforce the rule not to modify other *Aggregate* instances within the same transaction. With only identities of other *Aggregates* , there is no easy way to obtain a direct object reference to them.

Another benefit to using reference by identity only is that your *Aggregates* can be easily stored in just about any kind of persistence mechanism, such as relational database, document database, key-value store, and data grids/fabrics. This means that you have options to use a MySQL relational table, a JSON-based store such as PostgreSQL or MongoDB, GemFire/Geode, Coherence, and GigaSpaces.

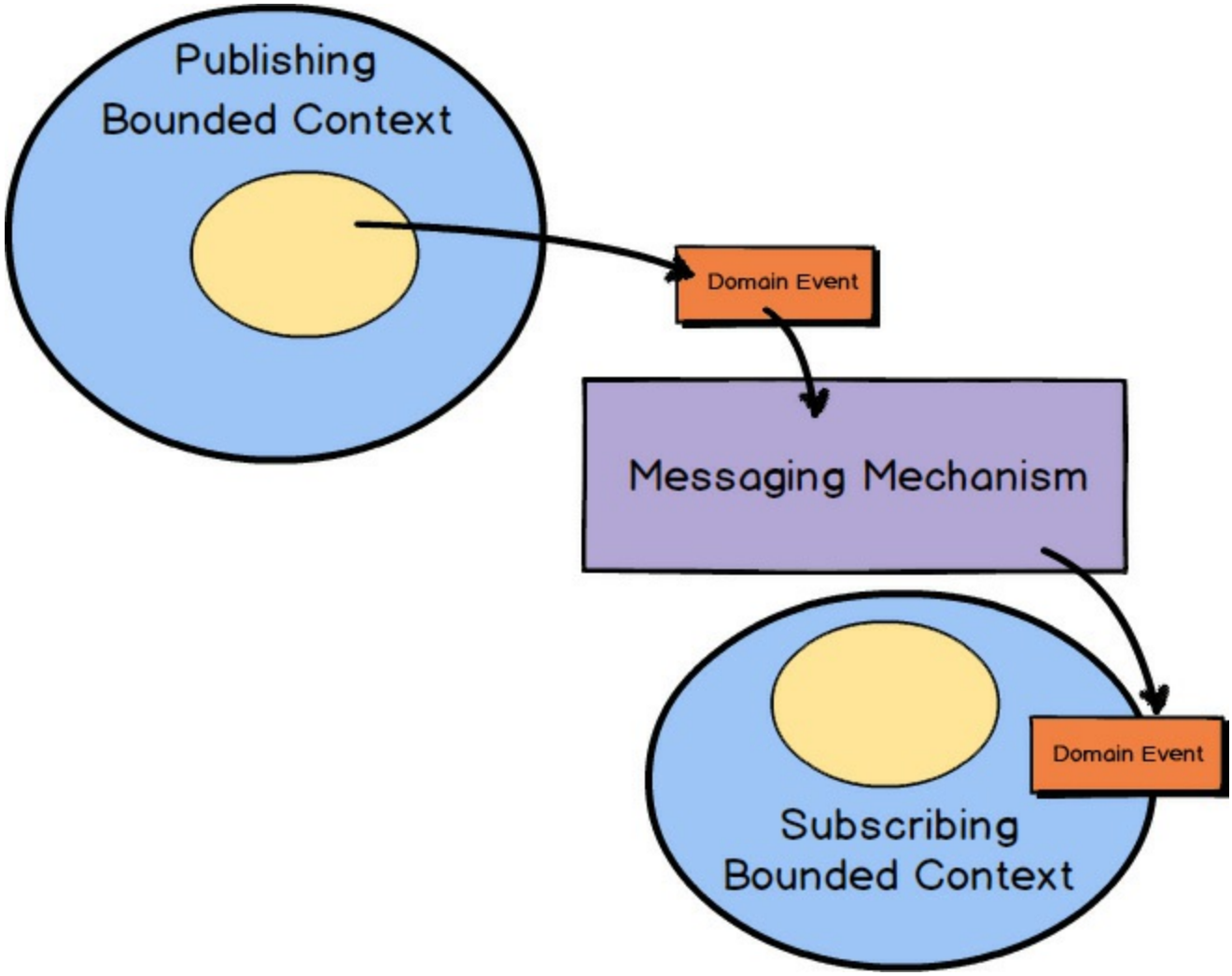


Rule 4: Update Other Aggregates Using Eventual Consistency

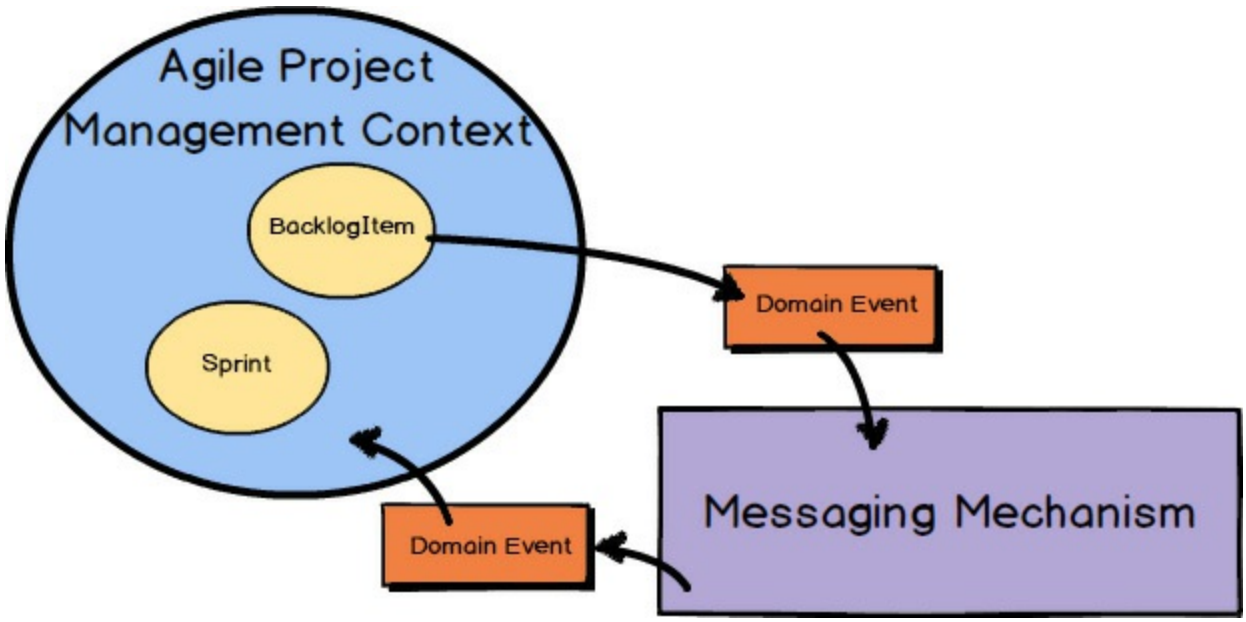
Here a `BacklogItem` is committed to a `Sprint` . Both the `BacklogItem` and the `Sprint` must react to this. It is first the `BacklogItem` that knows it has been committed to a `Sprint` . This is managed in one transaction, when the state of the `BacklogItem` is modified to contain the `SprintId` of the `Sprint` to which it is committed. So, how do we ensure that the `Sprint` is also updated with the `BacklogItemId` of the newly committed `BacklogItem` ?



As part of the `BacklogItem` *Aggregate* 's transaction, it publishes a *Domain Event* named `BacklogItemCommitted` . The `BacklogItem` transaction completes and its state is persisted along with the `Backlog-ItemCommitted` *Domain Event.* When the `BacklogItemCommitted` makes its way to a local subscriber, a transaction is started and the state of the `Sprint` is modified to hold the `BacklogItemId` of the committed `BacklogItem` . The `Sprint` holds the `BacklogItemId` inside a new `CommittedBacklogItem` *Entity.*



Recall now what you learned in Chapter 4 , "Strategic Design with Context Mapping ." *Domain Events* are published by an *Aggregate* and subscribed to by an interested *Bounded Context*. The messaging mechanism delivers the *Domain Events* to interested parties by means of subscriptions. The interested *Bounded Context* can be the same one from which the *Domain Event* was published, or it could be different *Bounded Contexts*.



It's just that in the case of the `BacklogItem` *Aggregate* and the `Sprint` *Aggregate* , the publisher and subscriber are in the same *Bounded Context*. You don't absolutely need to use a full-blown messaging middleware product for this case, but it's easy to do so since you already use it for

publishing to other *Bounded Contexts*.

If Eventual Consistency Seems Scary

There is nothing incredibly difficult about using eventual consistency. Still, until you can gain some experience, you may be concerned about using it. If so, you should still partition your model into *Aggregates* according to business-defined transactional boundaries. However, there is nothing preventing you from committing modifications to two or more *Aggregates* in a single atomic database transaction. You might choose to use this approach in cases that you know will succeed but use eventual consistency for all others. This will allow you to get used to the techniques without taking too big an initial step. Just understand that this is not the primary way that *Aggregates* are meant to be used, and you may experience transactional failures as a result.

Modeling Aggregates

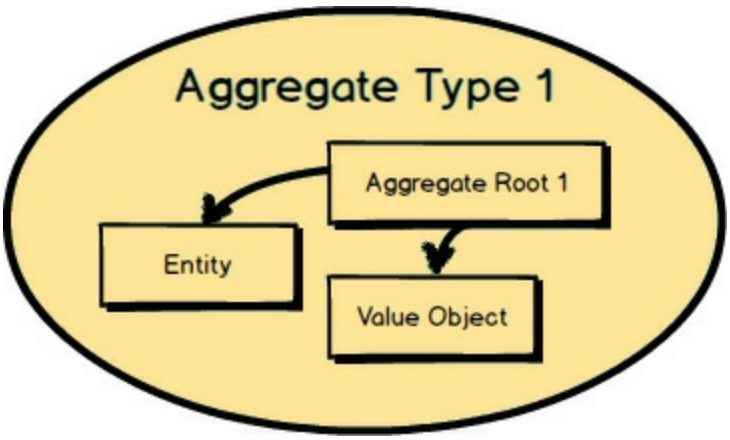
There are a few hooks waiting for you as you work on your domain model, implementing your *Aggregates*. One big, nasty hook is the *Anemic Domain Model* [IDDD] . This is where you are using an object-oriented domain model, and all of your *Aggregates* have only public accessors (getters and setters) but no real business behavior. This tends to happen when there is a technical rather than business focus during modeling. Designing an *Anemic Domain Model* requires you to take on all the overhead of a domain model without realizing any of its benefits. Don't take the bait!

Also watch out for leaking business logic into the Application Services above your domain model. It can happen undetected, just like physical anemia. Delegating business logic from services to helper/utility classes isn't going to work out well either. Service utilities always exhibit an identity crisis and can never keep their stories straight. Place your business logic in your domain model, or suffer bugs sponsored by an *Anemic Domain Model*.

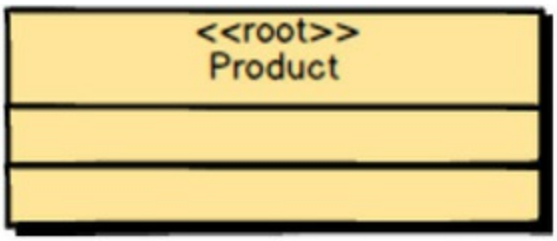
What about Functional Programming?

When using functional programming, the rules change considerably. While an *Anemic Domain Model* is a bad idea when using object-oriented programming, it is somewhat the norm when applying functional programming. That's because functional programming promotes the separation of data and behavior. Your data is designed as immutable data structures or record types, and your behavior is implemented as pure functions that operate on the immutable records of specific types. Rather than modifying the data that functions receive as arguments, the functions return new values. These new values may be the new state of an *Aggregate* or a *Domain Event* that represents a transition in an *Aggregate* 's state.

I have largely addressed the object-oriented approach in this chapter because it is still the most widely used and well understood. Yet if you are employing a functional language and approach to DDD, be aware that some of this guidance is not applicable or is at least subject to overriding rules.



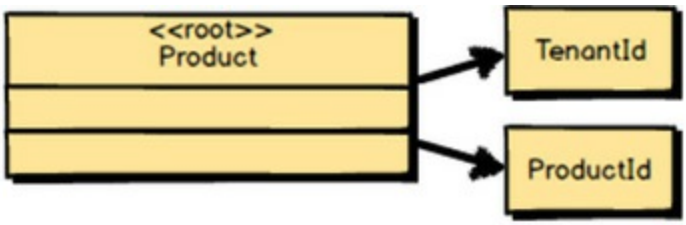
Next I'm going to show you some of the technical components you will need to implement a basic *Aggregate* design. I assume you are using Scala, C#, Java, or another object-oriented programming language. The following examples are in C# but are very understandable by Scala, F#, Java, Ruby, Python, and other programmers alike.




```
public class Product : Entity
{
   ...
}
```

The first thing you must do is create a class for your *Aggregate Root Entity*. Here is a UML (Unified Modeling Language) representation of the `Product` *Root Entity*. Included is also the `Product` class in C#, which extends a base class named `Entity` . This base class just takes care of standard *Entity* kinds of things. See *Implementing Domain-Driven Design* [IDDD] for exhaustive

discussions on both *Entity* and *Aggregate* design and implementation.

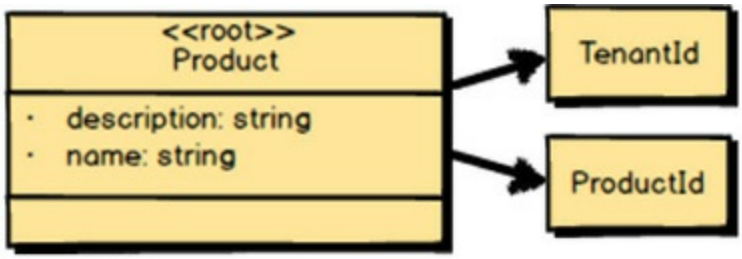



```
public class Product : Entity
{
  private ProductId productId;
  private TenantId tenantId;
}
```

Every *Aggregate Root Entity* must have a globally unique identity. A `Product` in the *Agile Project Management Context* actually has two forms of globally unique identity. The `TenantId` scopes the *Root Entity* inside a given subscriber organization. (Every organization that subscribes to the offered services is known as a tenant and thus has a unique identity for that.) The second identity, which is also globally unique, is the `ProductId` . This second identity sets the `Product` apart from all others within the same tenant. Also included is the C# code that declares the two identities inside `Product` .

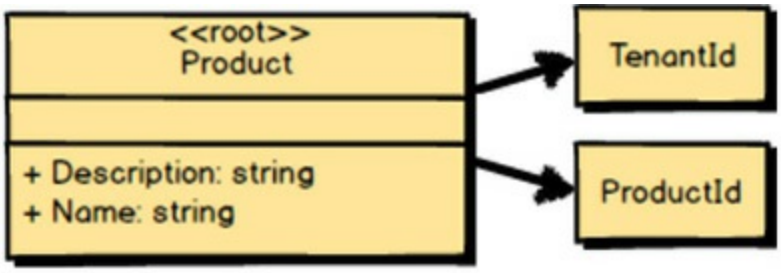
Use of Value Objects

Here, both `TenantId` and `ProductId` are modeled as immutable *Value Objects*.




```
public class Product : Entity
{
  private string description;
  private string name;
  private ProductId productId;
  private TenantId tenantId;
}
```

Next you capture any intrinsic attributes or fields that are necessary for finding the *Aggregate*. In the case of `Product` , there are both `description` and `name` . Users can search one or both of these to find each `Product` . I also provide the C# code that declares these two intrinsic attributes.



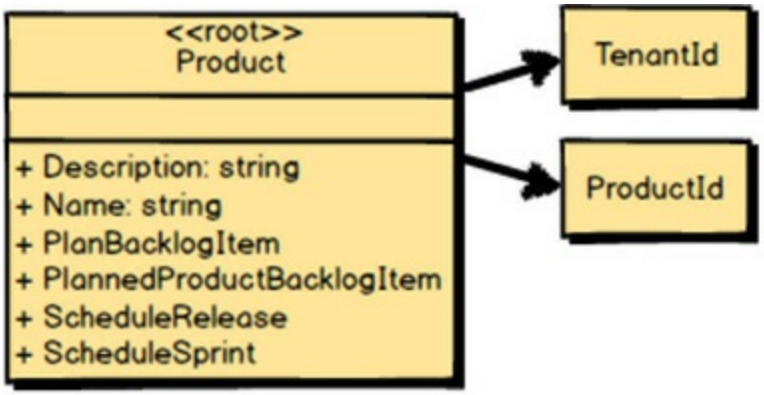

```
public class Product : Entity
{
  ...
  public string Description
    { get; private set; }

  public string Name
    { get; private set; }
}
```

Of course, you can add simple behavior such as read accessors (getters) for intrinsic attributes. In C# this would probably be done using public property getters. However, you may not want to expose setters as public. Without public setters, how do property/attribute values change? When using an object-oriented approach (C#, Scala, and Java), you change internal state using behavioral methods. If using a functional approach (F#, Scala, and Clojure), the functions will return new values that are different from the values passed as arguments.

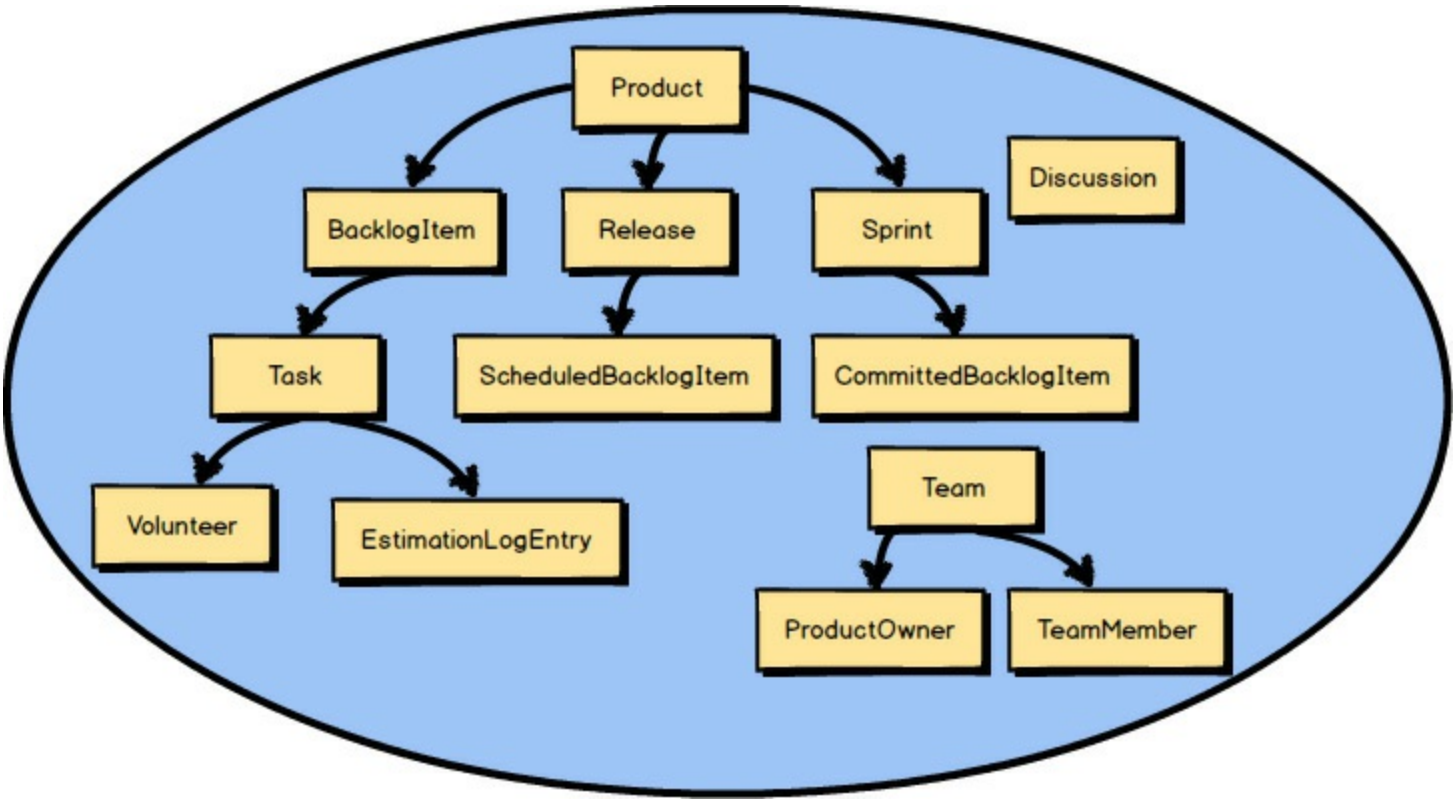
```
public class Product : Entity
{
  ...
  public string Name
    { get; private set; }
}
```

You should be on a mission to fight the *Anemic Domain Model* [IDDD] . If you expose public setter methods, it could quickly lead to anemia, because the logic for setting values on `Product` would be implemented outside the model. Think hard before doing this, and keep this warning in mind.




```
public class Product : Entity
{
   ...
   public void PlannedProductBacklogItem(...)
   {
      ...
   }
}
```

Finally, you add in any complex behavior. Here we've got four new methods: `PlanBacklogItem()`, `PlannedProductBacklogItem()`, `ScheduleRelease()`, and `ScheduleSprint()`. The C# code for each of these methods should be added to the class.



Remember, when using DDD, we are always modeling a *Ubiquitous Language* inside a *Bounded Context*. Thus, all parts of the `Product` *Aggregate* are modeled per the *Ubiquitous Language*. You don't just make up these composed parts. Everything shows harmony between *Domain Experts* and developers of your close-knit team.

Choose Your Abstractions Carefully

An effective software model is always based on a set of abstractions that address the business's way of doing things. There is, however, the need to choose the appropriate level of abstraction for each concept being modeled.

If you follow the direction of your *Ubiquitous Language* , you will generally create the proper abstractions. It's much easier to model the abstractions correctly because it is the *Domain Experts* who convey at least the genesis of your modeling language. Still, sometimes software developers who are overzealous for solving the wrong problems will try to force in abstractions that are, well, too abstract.

For example, in the *Agile Project Management Context* we are dealing with Scrum. It makes sense to model `Product` , `BacklogItem` , `Release` , and `Sprint` concepts that we've been discussing. Even so, what if the software developers were less concerned about modeling the *Ubiquitous Language* of Scrum, and more interested in modeling a solution to all current and future Scrum concepts?

If this angle were pursued, the developers would probably come up with abstractions such as `ScrumElement` and `ScrumElementContainer` . A `ScrumElement` could fill the current need for `Product` and `Backlog-Item` , and `ScrumElementContainer` could represent the obviously more explicit concepts of `Release` and `Sprint` . The `ScrumElement` would have a `typeName` property, and it would be set to `"Product"` or `"BacklogItem"` in appropriate cases. We could design the same kind of `typeName` property for `ScrumElementContainer` and allow the values `"Release"` or `"Sprint"` to be set on it.

Do you see the problems with this approach? There are more than a few, but consider the following:

- The language of the software model does not match the mental model of the *Domain Experts*.
- The level of abstraction is too high, and you will get into deep trouble when you start to model the details of each of the individual types.
- This will lead to creating special cases in each of the classes and likely result in a complex class hierarchy with general approaches to explicit problems.
- You will have much more code than you need, because you are trying to solve an unsolvable problem that should not matter in the first place.
- Often the language of the wrong abstractions will find its way even into the user interface, which will cause confusion for users.
- You will waste considerable time and money.
- You will never be able to address all future needs up front, which means if new Scrum concepts are ever added in the future, your existing model will prove to be a failure in foreseeing those needs.

Following such a path may seem strange to some, but this incorrect level of abstractions is used often in technically inspired implementations.

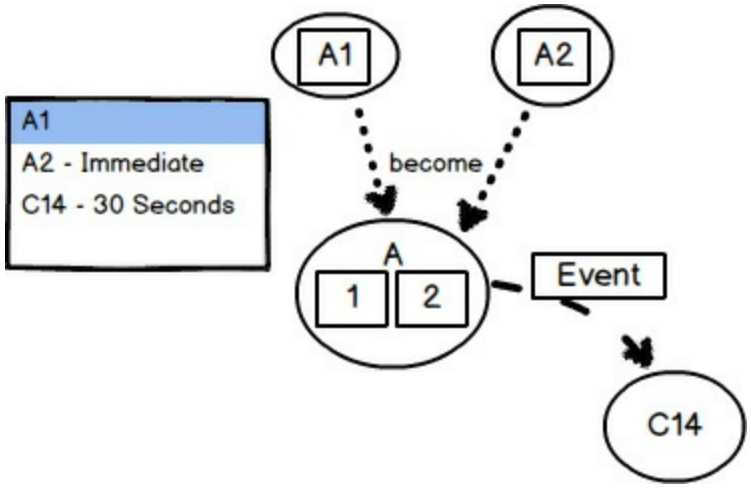
Don't get taken in by this alluring, highly abstract implementation trap. Model the *Ubiquitous Language* explicitly according to the mental model of the *Domain Experts* that is refined by your team. By modeling what the business needs today, you will save a considerable amount of time, budget, code, and embarrassment. More still, you will do the business a great service by modeling an accurate and useful *Bounded Context* that reflects an effective design.

Right-Sizing Aggregates

You may be wondering how you can determine the boundaries of *Aggregates* and prevent the design of large clusters, while still maintaining consistency boundaries that will protect true business invariants. Here I have provided a worthy design approach. If you have already created large-cluster *Aggregates* , you can use this approach to refactor into smaller ones, but I am not going to start from that perspective.

Consider these design steps that will help you reach consistency boundary goals:

1. Put your first focus on the second rule of *Aggregate* design, "Design small *Aggregates*. " Start by creating every *Aggregate* with just one *Entity* , which will serve as the *Aggregate Root*. Don't even dare to place two *Entities* in a single boundary. That opportunity will come soon enough. Populate each of the *Entities* with the fields/attributes/properties that you believe are most closely associated with the single *Root Entity*. One big hint here is to define every field/attribute/property that is required to identify and find the *Aggregate* , as well as any additional intrinsic fields/attributes/properties that are required for the *Aggregate* to be constructed and left in a valid initial state.
2. Now place your focus on the first rule of *Aggregate* design, "Protect business invariants inside *Aggregate* boundaries." You have already asserted by the previous step that at a minimum all the intrinsic fields/attributes must be up-to-date when the single-*Entity Aggregate* is persisted. But now you need to look at each of your *Aggregates* one at a time. As you do so for *Aggregate* A1, ask the *Domain Experts* if any other *Aggregates* you have defined must be updated in reaction to changes made to *Aggregate* A1. Make a list of each of the *Aggregates* and their consistency rules, which will indicate the time frames for all reaction-based updates. In other words, "*Aggregate* A1" would be the heading of one list, and other *Aggregate* types would be listed under A1 if they will be updated in reaction to A1 updates.
3. Now ask the *Domain Experts* how much time may elapse until each of the reaction-based updates may take place. This will lead to two kinds of specifications: (a) immediately, and (b) within *N* seconds/minutes/hours/days. One possible way to find the correct business threshold is by presenting an exaggerated time frame (such as weeks or months) that is obviously unacceptable. This will likely cause business experts to respond with an acceptable time frame.
4. For each of the immediate time frames (3a), you should strongly consider composing those two *Entities* within the same *Aggregate* boundary. That means, for example, that *Aggregate* A1 and *Aggregate* A2 will actually be composed into a new *Aggregate* A[1,2]. Now *Aggregates* A1 and A2 as they were previously defined will no longer exist. There is only *Aggregate* A[1,2].
5. For each of the reacting *Aggregates* that can be updated following a given elapsed time (3b), you will update these using the fourth rule of *Aggregate* design, "Update other *Aggregates* using eventual consistency."



In this figure our focus is on modeling *Aggregate* A1. Note from the A1 list of consistency rules that A2 has an immediate time frame, while C14 has an eventual (30 seconds) time frame. As a result, A1 and A2 are modeled into a single *Aggregate* A[1,2]. During runtime *Aggregate* A[1,2] publishes a *Domain Event* that causes *Aggregate* C14 to be updated eventually.

Be careful that the business doesn't insist that every *Aggregate* fall within the 3a specification (immediate consistency). It can be an especially strong tendency when many in the design session are influenced by database design and data modeling. Those stakeholders will have a very transaction-centered point of view. However, it is very unlikely that the business really needs immediate consistency in every case. To change this thinking you will probably have to spend time proving how transactions will fail due to concurrent updates by multiple users across different composed parts of the (now) large-cluster *Aggregates*. Furthermore, you can point out how much memory overhead there is with such large-cluster designs. Obviously these kinds of problems are what we are trying to avoid in the first place.

This exercise indicates that eventual consistency is business driven, not technically driven. Of course, you will have to find a way to technically cause eventual updates between multiple *Aggregates* , as discussed in the previous chapter on *Context Mapping*. Even so, it is only the business that can determine the acceptable time frame for updates to occur between various *Entities*. Some are immediate, or transactional, which means they must be managed by the same *Aggregate*. Some are eventual, which means they may be managed through *Domain Events* and messaging, for example. Considering what the business would have to do if it ran its operations only by means of paper systems can provide some worthwhile insights into how various domain-driven operations should work within a software model of the business operations.

Testable Units

You should also design your *Aggregates* to be a sound encapsulation for unit testing. Complex *Aggregates* are hard to test. Following the previous design guidance will help you model testable *Aggregates*.

Unit testing is different from validating business specifications (acceptance tests) as discussed in Chapter 2 , "Strategic Design with Bounded Contexts and the Ubiquitous Language ," and Chapter 7 , "Acceleration and Management Tools ." Development of the unit tests will follow the creation of scenario specification acceptance tests. What we are concerned with here is testing that the *Aggregate* correctly does what it is supposed to do. You want to push on all the operations to ensure the correctness, quality, and stability of your *Aggregates*. You can use a unit testing framework for

this, and there is much literature available on how to effectively unit test. These unit tests will be directly associated with your *Bounded Context* and kept with its source code repository.

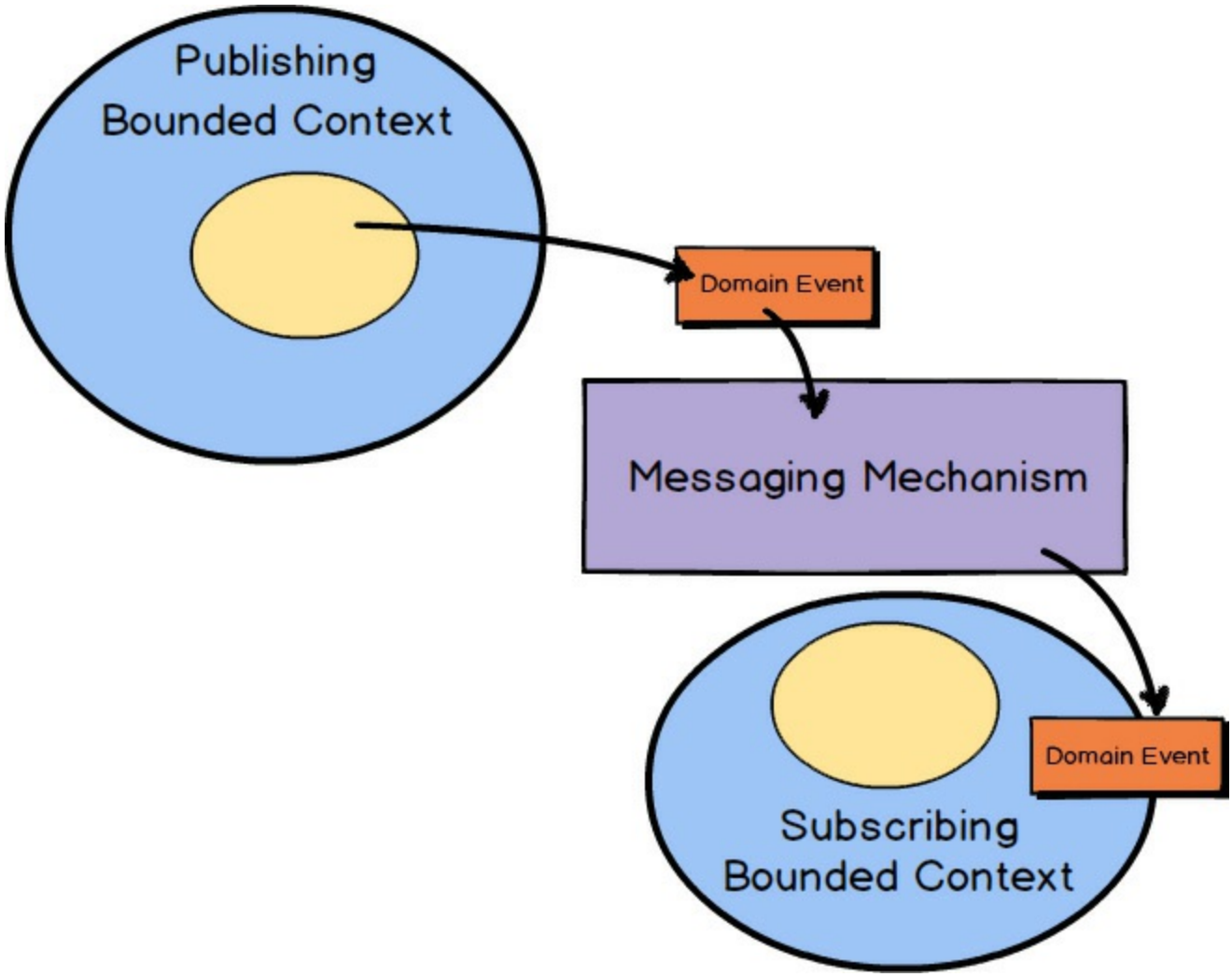
Summary

In this chapter you learned:

- What the *Aggregate* pattern is and why you should use it
- The importance of designing with a consistency boundary in mind
- About the various parts of an *Aggregate*
- The four rules of thumb of effective *Aggregate* design
- How you can model an *Aggregate* 's unique identity
- The importance of *Aggregate* attributes and how to prevent creating an *Anemic Domain Model*
- How to model behavior on an *Aggregate*
- To always adhere to the *Ubiquitous Language* within a *Bounded Context*
- The importance of selecting the proper level of abstraction for your designs
- A technique for right-sizing your *Aggregate* compositions, and how that includes designing for testability

For a more in-depth treatment of *Entities* , *Value Objects* , and *Aggregates* , see Chapters 5 , 6 , and 10 of *Implementing Domain-Driven Design* [IDDD] .

Chapter 6. Tactical Design with Domain Events



You've already seen a bit in previous chapters about how *Domain Events* are used. A *Domain Event* is a record of some business-significant occurrence in a *Bounded Context*. By now you know that *Domain Events* are a very important tool for strategic design. Still, often during tactical design *Domain Events* are conceptualized and become a part of your *Core Domain*.

To see the full power that results from using *Domain Events* , consider the concept of causal consistency. A business domain provides causal consistency if its operations that are causally related —one operation causes another—are seen by every dependent node of a distributed system in the same order [Causal] . This means that causally related operations must occur in a specific order, and thus one thing cannot happen unless another thing happens before it. Perhaps this means that one *Aggregate* cannot be created or modified until it is clear that a specific operation occurred to another *Aggregate* :

1. Sue posts a message saying, "I lost my wallet!"
2. Gary says in reply, "That's terrible!"
3. Sue posts a message saying, "Don't worry, I found my wallet!"
4. Gary replies, "That's great!"

If these messages were replicated on distributed nodes, but not in a causal order, it could appear that Gary said, "That's great!" to the message "I lost my wallet!" The message "That's great!" is not directly or causally related to "I lost my wallet!" and that's definitely not what Gary wants Sue or

anyone else to read. Thus, if causality is not achieved in the proper way, the overall domain would be wrong or at least misleading. This sort of causal, linearized system architecture can be readily achieved through the creation and publication of correctly ordered *Domain Events*.

From tactical design efforts *Domain Events* become a reality in your domain model and can as a result be published and consumed in your own *Bounded Context* and by others. It's a very powerful way to inform interested listeners of important occurrences that have taken place. Now you will learn how to model *Domain Events* and use them in your *Bounded Contexts*.

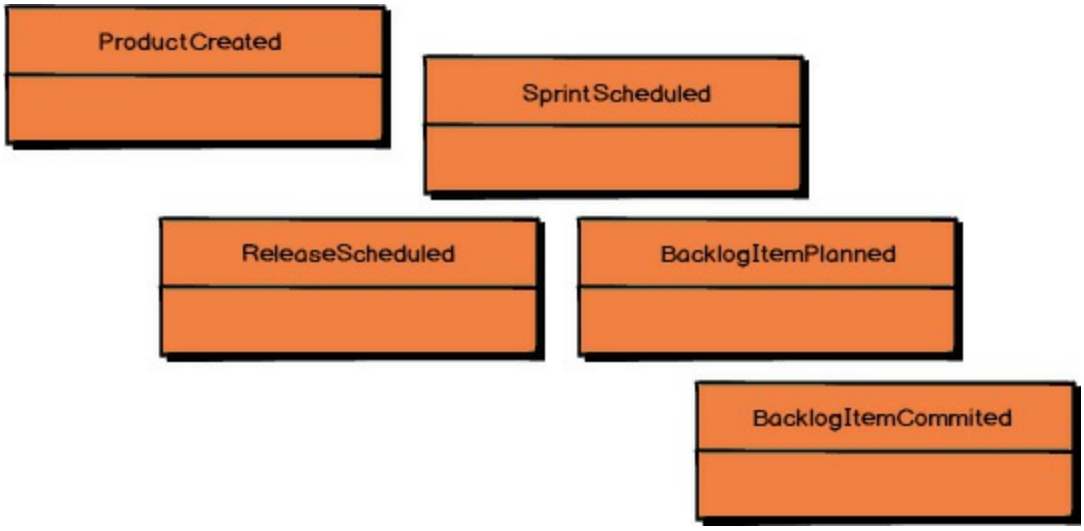
Designing, Implementing, and Using Domain Events

The following guides you through the steps needed to effectively design and implement *Domain Events* in your *Bounded Context*. Following this, you will also see examples of how *Domain Events* are used.

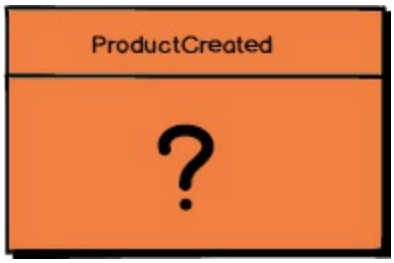

```
public interface DomainEvent
{
  public Date OccurredOn
  {
    get;
  }
}
```

This C# code might be considered the minimum interface that every *Domain Event* should support. You generally want to convey the date and time when your *Domain Event* occurred, so that's provided by the `OccurredOn` property. This detail is not an absolute necessity, but it is often useful. So your *Domain Event* types would likely implement this interface.

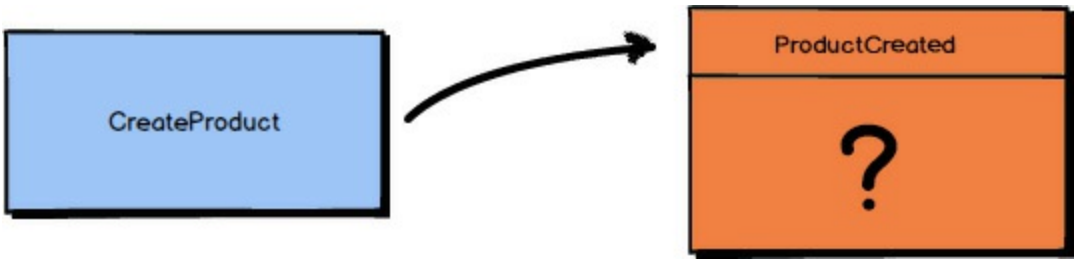
You must show care in how you name your *Domain Event* types. The words you use should reflect your model's *Ubiquitous Language*. These words will form a bridge between the happenings in your model and the outside world. It's vital that you communicate your happenings well.



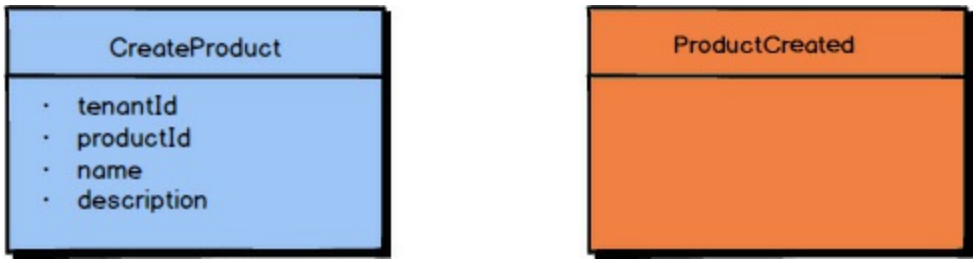
Your *Domain Event* type names should be a statement of a past occurrence, that is, a verb in the past tense. Here are some examples from the *Agile Project Management Context* : `ProductCreated` , for instance, states that a Scrum product was created at some past time. Other *Domain Events* are `ReleaseScheduled` , `SprintScheduled` , `BacklogItemPlanned` , and `BacklogItemCommitted` . Each of the names clearly and concisely states what happened in your *Core Domain.*



It's a combination of the *Domain Event* 's name and its properties that fully conveys the record of what happened in the domain model. But what properties should a *Domain Event* hold?

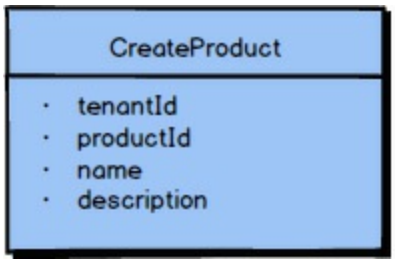


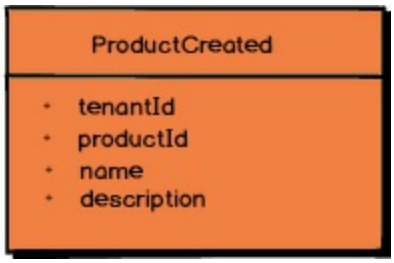
Ask yourself, "What is the application stimulus that causes the *Domain Event* to be published?" In the case of `ProductCreated` there is a command that causes it (a command is just the object form of a method/action request). The command is named `CreateProduct` . So you can say that `ProductCreated` is the result of a `CreateProduct` command.



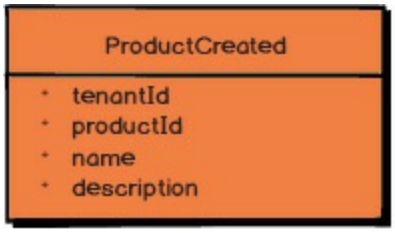
The `CreateProduct` command has a number of properties: (1) the `tenantId` that identifies the subscribing tenant, (2) the `productId` that identifies the unique `Product` being created, the (3) `Product name` , and (4) the `Product description` . Each of these properties is

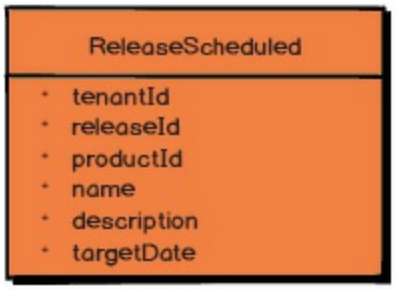
essential to creating a `Product`.





Therefore, the `ProductCreated` *Domain Event* should hold all the properties that were provided with the command that caused it to be created: (1) `tenantId`, (2) `productId`, (3) `name`, and (4) `description`. This will fully and accurately inform all subscribers what happened in the model; that is, a `Product` was created, it was for the tenant identified with the `tenantId`, the `Product` was uniquely identified with `productId`, and the `Product` had the `name` and `description` assigned to it.

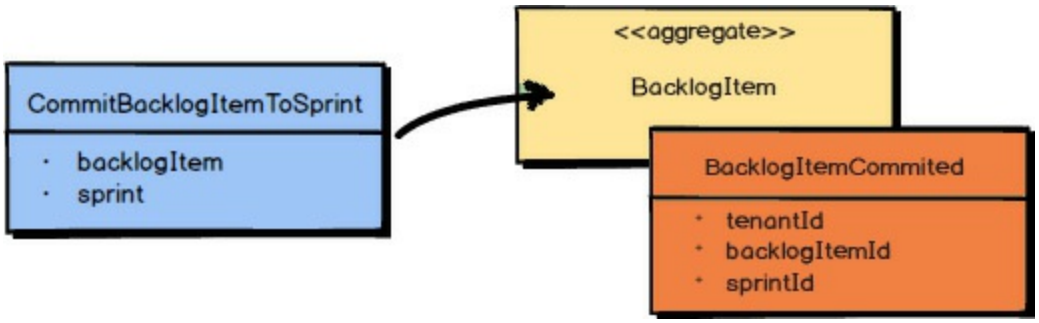




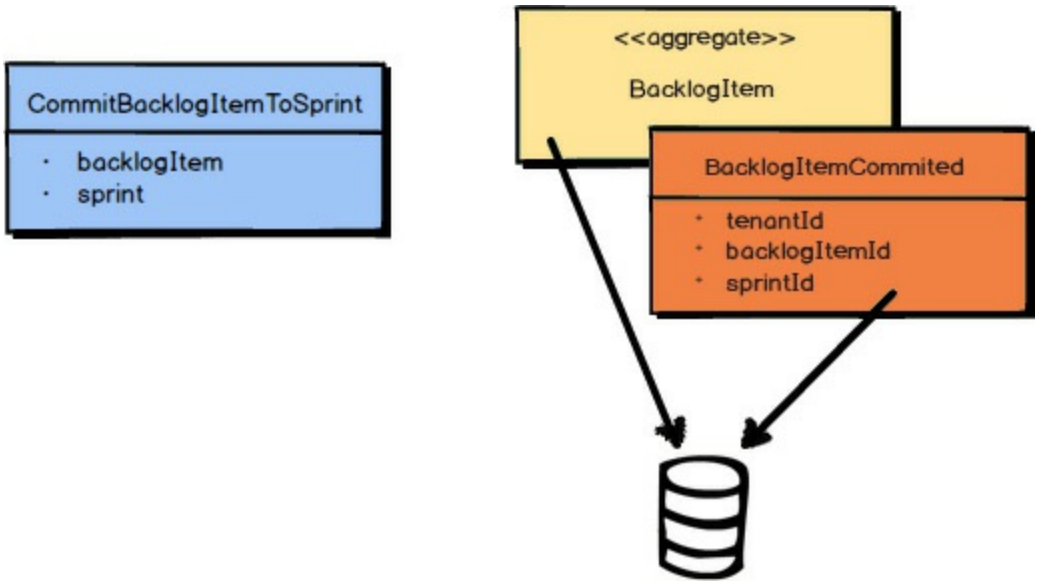
These five examples give you a good idea of the properties that should be included with the various *Domain Events* published by the *Agile Project Management Context*. For instance, when a `BacklogItem` is committed to a `Sprint`, the `BacklogItemCommitted` *Domain Event* is instantiated and published. This *Domain Event* contains the `tenantId`, the `backlogItemId` of the `BacklogItem` that was committed, and the `sprintId` of the `Sprint` to which it was committed.

As described in Chapter 4, "Strategic Design with Context Mapping," there are times when a *Domain Event* can be enriched with additional data. This can be especially helpful to consumers that don't want to query back on your *Bounded Context* to obtain additional data that they need. Even so, you must be careful not to fill up a *Domain Event* with so much data that it loses its meaning. For example, consider the problem with `BacklogItemCommitted` holding the entire state of the `BacklogItem`. According to this *Domain Event*, what actually happened? All the extra data may make it unclear, unless you require the consumer to have a deep understanding of your

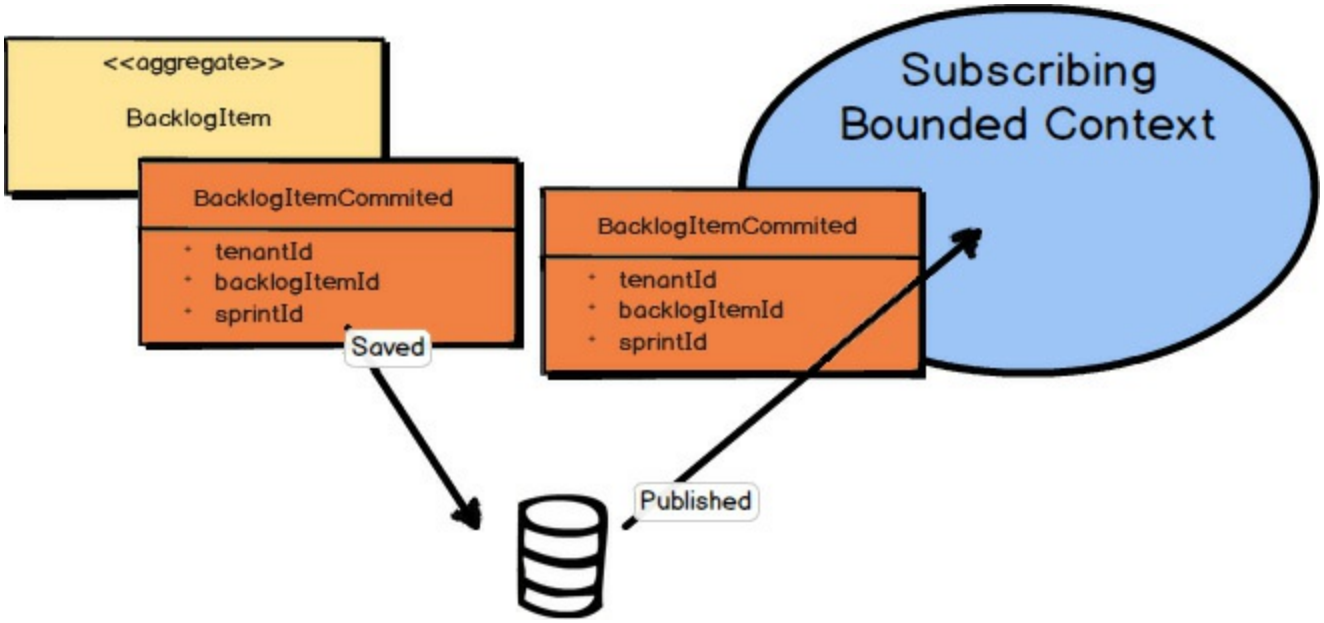
`BacklogItem` element. Also, consider using `BacklogItemUpdated` with the full state of the `BacklogItem` , as opposed to providing `BacklogItemCommitted` . What happened to the `BacklogItem` is very unclear, because the consumer would have to compare the latest `BacklogItemUpdated` to the previous `BacklogItemUpdated` in order to understand what actually occurred to the `BacklogItem` .



To make the proper use of *Domain Events* clearer, let's walk through one scenario. The product owner commits a `BacklogItem` to a `Sprint` . The command itself causes the `BacklogItem` and the `Sprint` to be loaded. Then the command is executed on the `BacklogItem` *Aggregate*. This causes the state of the `BacklogItem` to be modified, and then the `BacklogItemCommitted` *Domain Event* is published as an outcome.

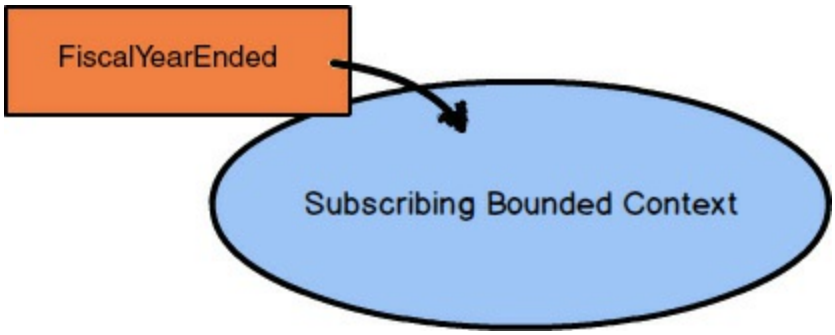


It's important that the modified *Aggregate* and the *Domain Event* be saved together in the same transaction. If you are using an object-relational mapping tool, you would save the *Aggregate* to one table and the *Domain Event* to an event store table, and then commit the transaction. If you are using *Event Sourcing* , the state of the *Aggregate* is fully represented by the *Domain Events* themselves. I discuss *Event Sourcing* in the next section of this chapter. Either way, persisting the *Domain Event* in the event store preserves its causal ordering relative to what has happened across the domain model.



Once your *Domain Event* is saved to the event store, it can be published to any interested parties. This might be within your own *Bounded Context* and to external *Bounded Contexts*. This is your way of telling the world that something noteworthy has occurred in your *Core Domain*.

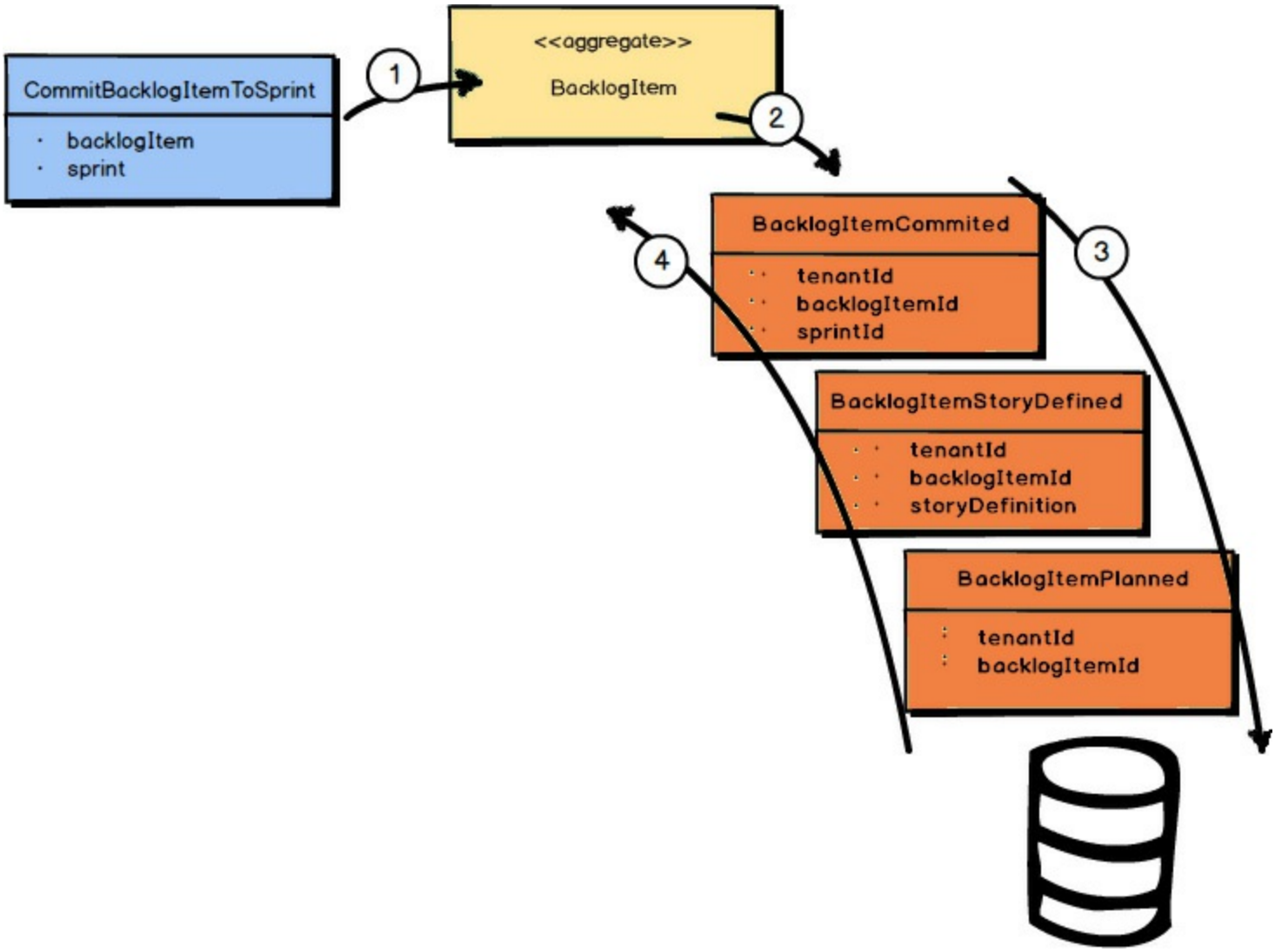
Note that just saving the *Domain Event* in its causal order doesn't guarantee that it will arrive at other distributed nodes in the same order. Thus, it is also the responsibility of the consuming *Bounded Context* to recognize proper causality. It might be the *Domain Event* type itself that can indicate causality, or it may be metadata associated with the *Domain Event* , such as a sequence or causal identifier. The sequence or causal identifier would indicate what caused this *Domain Event* , and if the cause was not yet seen, the consumer must wait to apply the newly arrived event until its cause arrives. In some cases it is possible to ignore latent *Domain Events* that have already been superseded by the actions associated with a later one; in this case causality has a dismissible impact.



One more point about what can cause a *Domain Event* is noteworthy. Although often it is a user-based command emitted by the user interface that causes an event to occur, sometimes *Domain Events* can be caused by a different source. This might be from a timer that expires, such as at the end of the business day or the end of a week, month, or year. In cases like this it won't be a command that causes the event, because the ending of some time period is a matter of fact. You can't reject the fact that some time frame has expired, and if the business cares about this fact, the time expiration is modeled as a *Domain Event* , and not as a command.

What is more, such an expiring time frame will generally have a descriptive name that will become part of the *Ubiquitous Language*. For example, "Fiscal Year Ended" may be an important event that your business needs to react to. Furthermore, 4:00 p.m. (16:00) on Wall Street is known as "Markets Closed" and not just as 4:00 p.m. Therefore, you have a name for that particular time-based *Domain Event*.

A command is different from a *Domain Event* in that a command can be rejected as inappropriate in some cases, such as due to supply and availability of some resources (product, funds, etc.), or another kind of business-level validation. So, a command may be rejected, but a *Domain Event* is a matter of history and cannot logically be denied. Even so, in response to a time-based *Domain Event* it could be that the application will need to generate one or more commands in order to ask the application to carry out some set of actions.



Event Sourcing

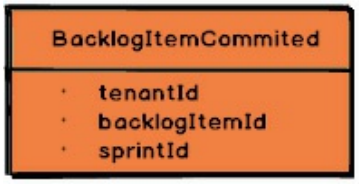
Event Sourcing can be described as persisting all *Domain Events* that have occurred for an *Aggregate* instance as a record of what changed about that *Aggregate* instance. Rather than persisting the *Aggregate* state as a whole, you store all the individual *Domain Events* that have happened to it. Let's step through how this is supported.

All of the *Domain Events* that have occurred for one *Aggregate* instance, ordered as they originally occurred, make up its event stream. The event stream begins with the first *Domain Event* that ever occurred for the *Aggregate* instance and continues until the last *Domain Event* that occurred. As new *Domain Events* occur for a given *Aggregate* instance, they are appended to the end of its event stream. Reapplying the event stream to the *Aggregate* allows its state to be reconstituted from persistence back into memory. In other words, when using *Event Sourcing* , an *Aggregate* that was removed from memory for any reason is reconstituted entirely from its event stream.

In the preceding diagram, the first *Domain Event* to occur was `BacklogItemPlanned` ; the

next was `BacklogItemStoryDefined` ; and the event that just occurred is `BacklogItemCommitted` . The full event stream is currently composed of those three events, and they follow the order described and seen in the diagram.

Each of the *Domain Events* that occurs for a given *Aggregate* instance is caused by a command, just as described previously. In the preceding diagram, it is the `CommitBacklogItemToSprint` command that has just been handled, and this has caused the `BacklogItemCommitted` *Domain Event* to occur.



Stream Id	Stream Version	Event Type	Event Content
backlogItem123	1	BacklogItemPlanned	{ ... }
backlogItem123	2	BacklogItemStoryDefined	{ ... }
backlogItem123	3	BacklogItemCommitted	{ ... }
...	N	...	{ ... }
...	N	...	{ ... }
...	N	...	{ ... }

The event store is just a sequential storage collection or table where all *Domain Events* are appended. Because the event store is append-only, it makes the storage mechanism extremely fast, so you can plan on a *Core Domain* that uses *Event Sourcing* to have very high throughput, low latency, and be capable of high scalability.

Performance Conscious

If one of your primary concerns is performance, you will appreciate knowing about caching and snapshots. First of all, your highest-performing *Aggregates* will be those that are cached in memory, where there is no need to reconstitute them from storage each time they are used. Using the Actor model with actors as *Aggregates* [Reactive] is one of the easier ways to keep your *Aggregates'* state cached.

Another tool at your disposal is snapshots, where the load time of your *Aggregates* that have been evicted from memory can be reconstituted optimally without reloading every *Domain Event* from an event stream. This translates to maintaining a snapshot of some incremental state of your *Aggregate* (object, actor, or record) in the database. Snapshots are discussed in more detail in *Implementing Domain-Driven Design* [IDDD] and in *Reactive Messaging Patterns with the Actor Model* [Reactive] .

One of the greatest advantages of using *Event Sourcing* is that it saves a record of everything that has ever happened in your *Core Domain* , at the individual occurrence level. This can be very helpful to your business for many reasons, ones that you can imagine today, such as compliance and analytics, and ones that you won't realize until later. There are also technical advantages. For example, software developers can use event streams to examine usage trends and to debug their source code.

You can find coverage of *Event Sourcing* techniques in *Implementing Domain-Driven Design* [IDDD] . Also, when you use *Event Sourcing* you are almost certainly obligated to use CQRS. You can also find discussions of this topic in *Implementing Domain-Driven Design* [IDDD] .

Summary

In this chapter you learned:

- How to create and name your *Domain Events*
- The importance of defining and implementing a standard *Domain Event* interface
- That naming your *Domain Events* well is especially important
- How to define the properties of your *Domain Events*
- That some *Domain Events* may be caused by commands, while others may happen due to the detection of some other changing state, such as a date or time
- How to save your *Domain Events* to an event store
- How to publish your *Domain Events* after they are saved
- About *Event Sourcing* and how your *Domain Events* can be stored and used to represent the state of your *Aggregates*

For a thorough treatment of *Domain Events* and integration, see Chapters 8 and 13 of *Implementing Domain-Driven Design* [IDDD] .

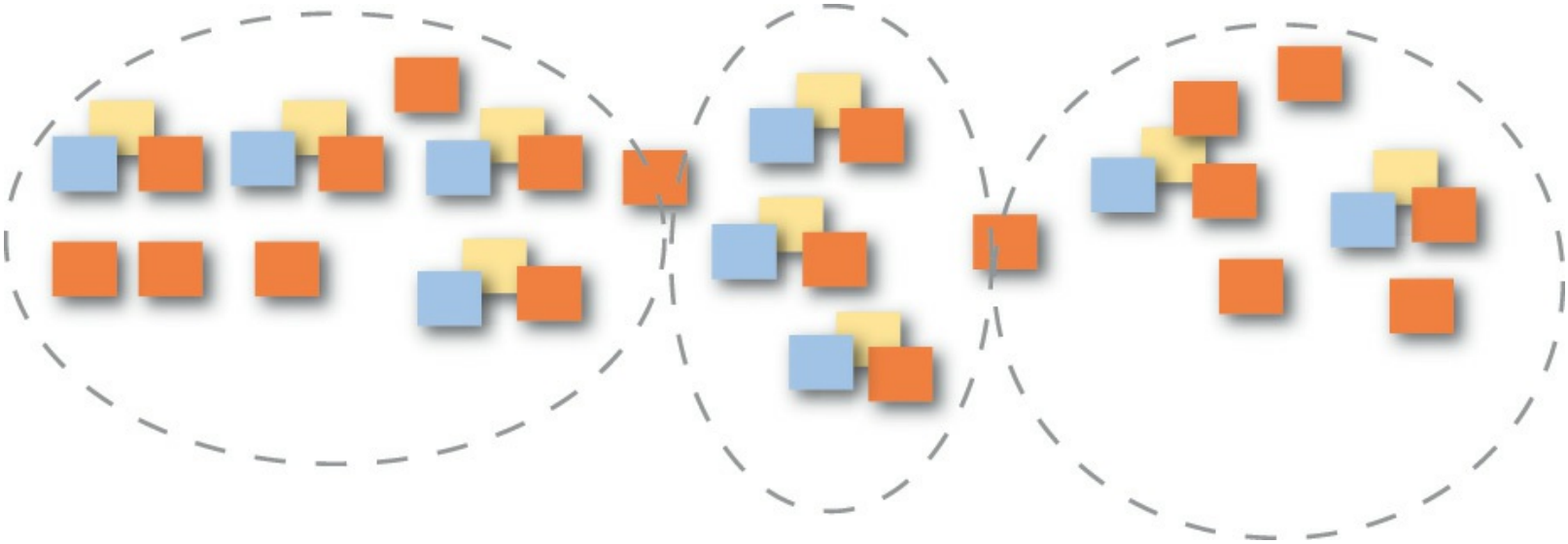
Chapter 7. Acceleration and Management Tools

When using DDD we are on a quest for deep learning about how the business works, and then to model software based on the extent of our learning. It's really a process of learning, experimenting, challenging, learning more, and modeling again. We need to crunch and distill knowledge in great quantities and produce a design that is effective in meeting the strategic needs of an organization. The challenge is that we have to learn quickly. In a fast-paced industry we are usually working against time, because time matters, and time generally drives many of our decisions, possibly even more than it should. If we don't deliver on time and within budget, no matter what we have achieved with the software, we seem to have failed. And everyone is counting on us to succeed in every way.

Some have made efforts to convince management that most project time estimations are valueless and that they cannot be successfully used. I am not sure how those efforts are working out in the large, but every client with whom I work is still being pressured to deliver within very specific time frames, which forces timeboxing into the design/implementation process. At best it's a constant struggle between software development and management.

Unfortunately, one common response to this negative pressure is to try to economize and shorten timelines by eliminating design. Recall from the first chapter that design is inevitable, and that either you will do poorly as a result of bad design, or you will succeed by delivering with an effective design, and possibly even with a good design. So, what you should attempt to do is meet the demands of time squarely and design in an accelerated way, using approaches that will help you deliver the very best design possible within the limits of time that you face.

To that end I provide some very useful design acceleration and project management tools in this chapter. First I discuss *Event Storming* and then conclude with a way to leverage the artifacts produced by that collaboration process to create estimates that are meaningful and, best of all, attainable.



Event Storming

Event Storming is a rapid design technique that is meant to engage both *Domain Experts* and developers in a fast-paced learning process. It is focused on the business and business process rather than on nouns and data.

Prior to learning *Event Storming* I used a technique that I called event-driven modeling. It usually involved conversations, concrete scenarios, and event-centric modeling using very lightweight UML. The UML-specific steps could be achieved on a whiteboard alone and could also be captured in a tool. However, as you probably know, few business people are informed and proficient in even a minimalistic use of UML. So, this left most of the modeling part of the exercise to me or another developer who understood the basics of UML. It was a very useful approach, but there had to be a way to get business experts more directly involved in the process. That probably meant leaving out UML in favor of a more engaging tool.

I first learned about *Event Storming* years ago from Alberto Brandolini [Ziobrando] , who had also experimented with another form of event-driven modeling. On one occasion, being short on time, Alberto decided that he should ditch the UML and use sticky notes instead. This was the birth of an approach to rapid learning and software design that got everybody in the room very directly involved in the process. Here are some of its advantages:

- It is a very tactile approach. Everyone gets a pad of sticky notes and a pen and is responsible for contributing to the learning and design sessions. Both business people and developers stand on equal ground as they learn together. Everyone provides input to the *Ubiquitous Language*.
- It focuses everyone on events and the business process rather than on classes and the database.
- It is a very visual approach, which dismisses code from the experimentation and puts everyone on a level footing with the design process.
- It is very fast and very cheap to perform. You can literally storm out a new *Core Domain* in rough format in a matter of hours rather than weeks. If you write something on a sticky note that you later decide doesn't work, you wad up the sticky note and throw it away. It costs you only a penny or two for that mistake, and no one is going to resist the opportunity to refine due to effort already invested.
- Your team will have breakthroughs in understanding. Period. It happens every time. Some will come to the session thinking that they have a pretty good understanding of the specific core

business model, but no matter, they always leave with a greater understanding and even new insights about the business process.

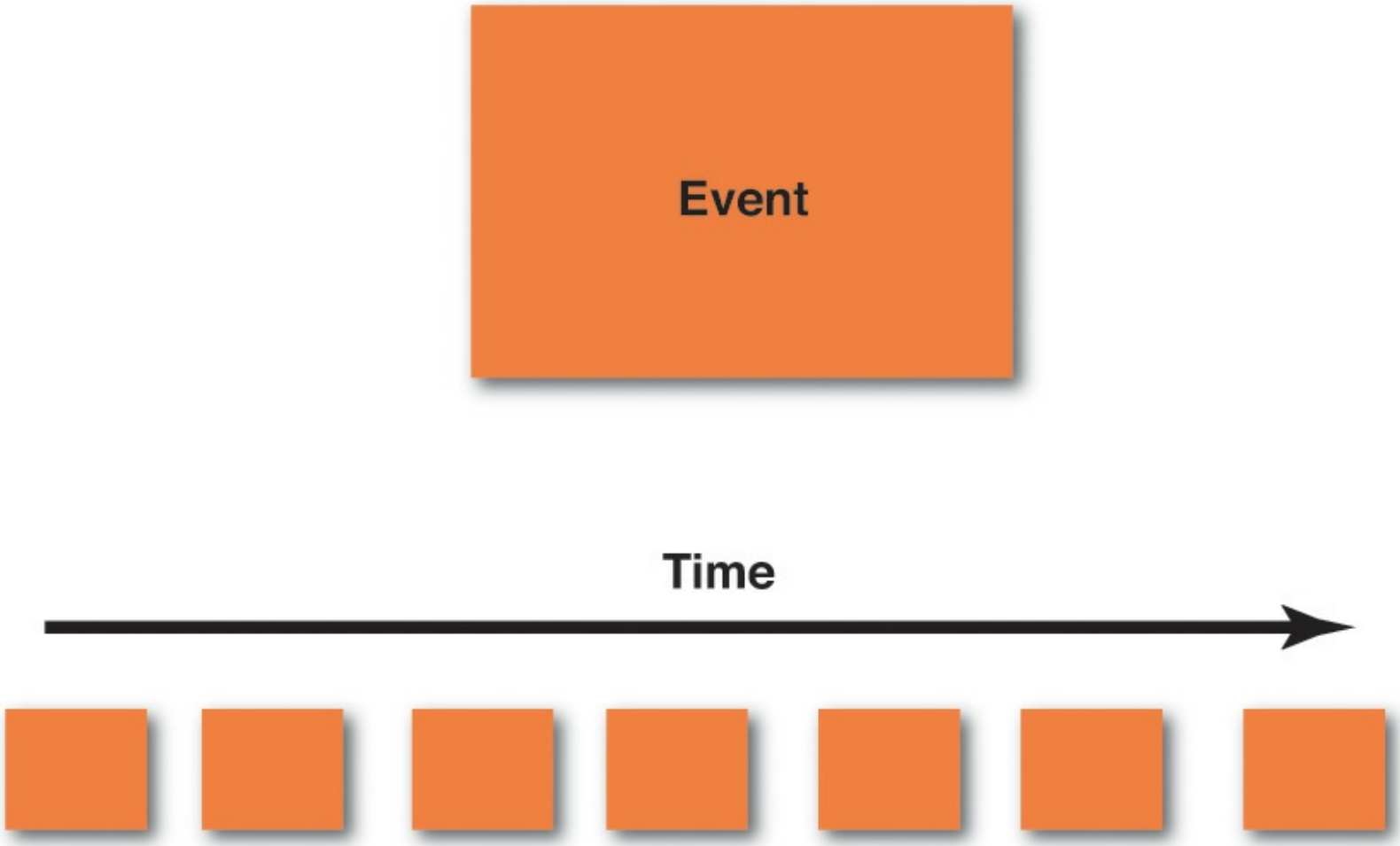
- Everybody learns something. Whether you are a *Domain Expert* or a software developer, you will walk away from the sessions with a crisp, clear understanding of the model at hand. This is different from achieving breakthroughs and is important in its own right. In many projects, at least some project members, and possibly many, do not understand what they are working on until it is too late and the damage is already in the code. Storming out a model helps everyone clear up misunderstandings and move forward with a unified direction and purpose.
- This implies that you are also identifying problems in both the model and in understanding as early and quickly as possible. Iron out misunderstandings and leverage the outcome as new insights. Everybody in the room benefits.
- You can use *Event Storming* for both *big-picture* and *design-level* modeling. Doing big-picture storming will be less precise, while design-level storming will lead you toward certain software artifacts.
- It is unnecessary to limit the storming sessions to one. You can start off with a two-hour storming session, then take a break. Sleep on your accomplishments and return the next day to spend another hour or two to expand and refine. If you do this for two hours a day for three or four days, you will gain a deep understanding of your *Core Domain* and integrations with your surrounding *Subdomains*.

Here is a list of the people, mindset, and supplies you will need to storm out a model:

- Having the right people is essential, which means the *Domain Expert(s)* and developers who are to work on the model. Everyone is going to have some of the questions and some of the answers. In order to support each other, they all need to be in the same room during the modeling sessions.
- Everyone should come with an open mind that is free of strict judgment. The biggest mistake I see during *Event Storming* sessions is people trying to be too correct too soon. You should rather be fully determined to create far too many events. More events is better than fewer events, because that's what will make you learn the most. There is time to refine later, and refinement is fast and cheap.
- Have on hand an assortment of colors of sticky notes, and plenty of them. At a minimum you need these colors: orange, purple/red, light blue, pale yellow, lilac, and pink. You may find that other colors (such as green; see later examples) come in handy. The sticky note dimensions can be square (3 inches by 3 inches, or 7.62 cm by 7.62 cm) rather than the wider variety that are rectangular. You don't need to write much on the sticky note; usually just a few words will do. Consider getting the extra-sticky variety. You don't want your sticky notes falling on the floor.
- Provide one black marker pen for each person, which will enable handwriting to show up bold and clear. Fine-tip markers are best.
- Find a wide wall where you can model. Width is more important than height, but your modeling surface should be approximately one meter/yard high. Width should be practically unlimited, but something in the range of 10 meters/yards should be considered a minimum. In lieu of such a wall being available, you can always use a long conference table or the floor. The problem with a table is that it will ultimately limit your modeling space. The problem with the floor is that it might not be accessible to everyone on the team. A wall is best.

- Obtain a long roll of paper, such as can often be found at art stores, teaching supply stores, and even at Ikea stores. The paper should be to the dimensions described previously, with at least 10 meters/yards in width and 1 meter/yard in height. Hang the paper on the wall using strong tape. Some may decide to forgo the paper and just work on whiteboards. This may work for a while, but the sticky notes tend to lose adhesion over time on whiteboards, especially if they are pulled up and restuck in different locations. Stickies adhere longer when they are applied to paper. If you intend to model for brief periods of time over three or four days, rather than in one long session, longevity of stickiness is important.

Given the basic supplies and having the right people participating in the session, you are ready to begin. Consider each of the steps, one by one.



1. *Storm out the business process by creating a series of Domain Events on sticky notes. The most popular color to use for Domain Events is orange.* Using orange makes the *Domain Events* stand out most prominently on the modeling surface.

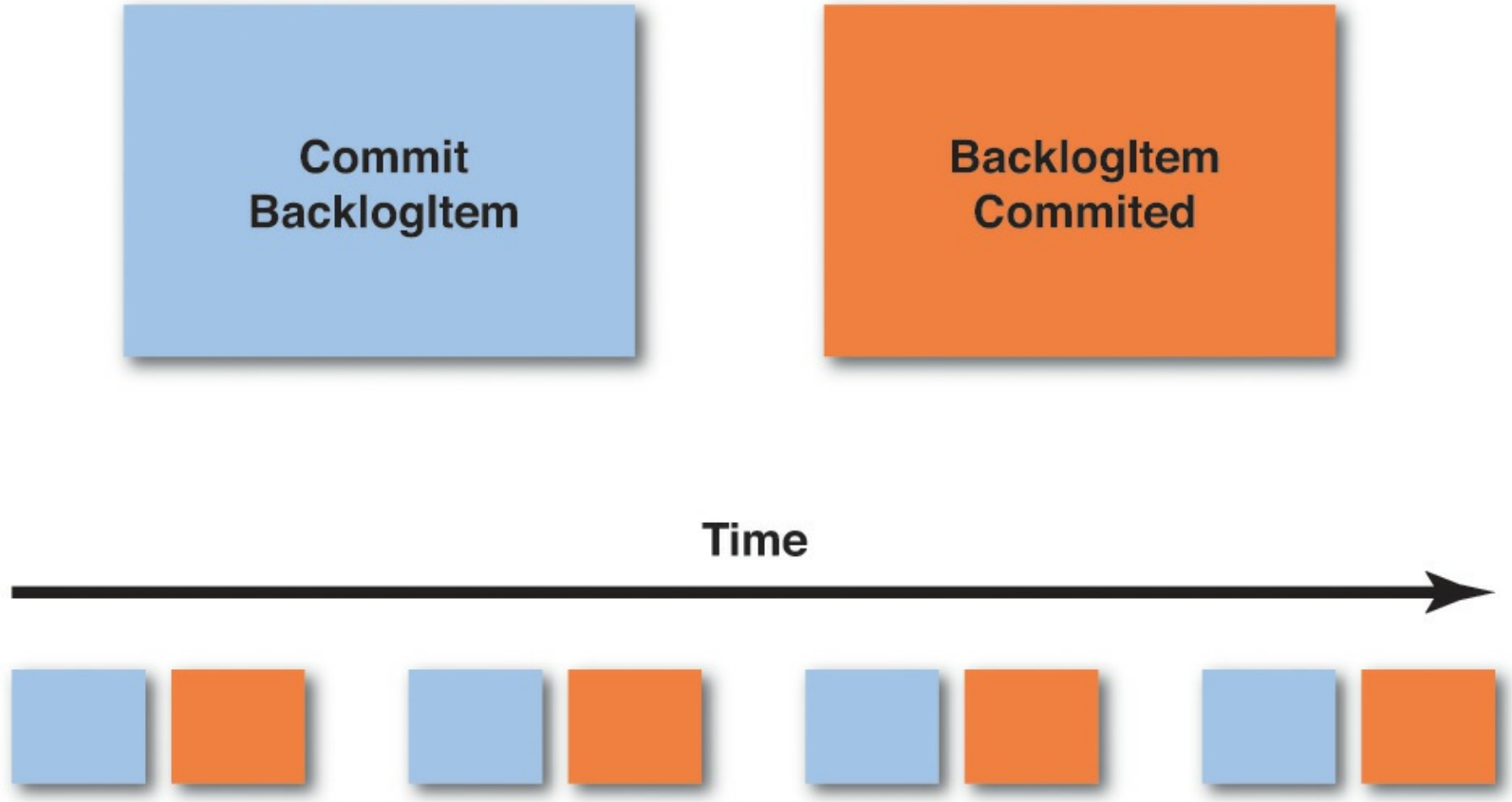
   The following are some basic guidelines that should be employed as you create your *Domain Events:*

   - Creating *Domain Events* first should emphasize that we have our first and primary focus on the business process, not on the data and its structure. It may take your team 10 to 15 minutes to warm up to this, but follow the steps just as I outline them here. Don't be tempted to jump ahead.
   - Write the name of each *Domain Event* on a sticky note. The name, as you learned in the previous chapter, should be a verb stated in the past tense. For example, an event may be named `ProductCreated`, and another may be named `BacklogItemCommitted`.

(You can certainly break these names into multiple lines on the sticky notes.) If you are doing big-picture storming and you think that these names are too precise for the participants, use other names.

- Place the sticky notes on your modeling surface in time order, that is, from left to right in the order in which each event occurs in the domain. You start with the first *Domain Events* to the far left of your modeling surface and then move gradually to the right. Sometimes you won't have a good understanding of the time order, in which case you should just put the corresponding *Domain Events* somewhere in the model. Figure out the "when" part, which will probably become obvious, later.
- A *Domain Event* that happens in parallel with another according to your business process can be located under the *Domain Event* that happens at the same time. So, you use vertical space to represent parallel processing.
- As you go through this part of the storming session, you are going to find trouble spots in your existing or new business process. Clearly mark these with a purple/red sticky note and some text that explains why it's a problem. You need to invest time at such points to learn more.
- Sometimes the outcome of a *Domain Event* is a *Process* that needs to run. This could be a single step or multiple complex steps. Each *Domain Event* that causes a *Proces* s to be executed should be captured and named on a lilac sticky note. Draw a line with an arrowhead from the *Domain Event* to the named *Process* (lilac sticky note). Model a fine-grained *Domain Event* only if it is important to your *Core Domain.* Most likely a user registration process is a necessity but probably not considered a core feature of your application. Model the registration process as one single coarse-grained event, `UserRegistered` , and move on. Put your concentrated efforts into more important events.

If you think you have exhausted all possible important *Domain Events* , it may be time to take a break and come back to the modeling session later. Returning to the modeling surface a day later will no doubt cause you to find missing concepts and to refine or toss out superficial ones that you previously considered important. Even so, at some point you will have identified most of the *Domain Events* that are of greatest importance. At that time you should move on to the next step.

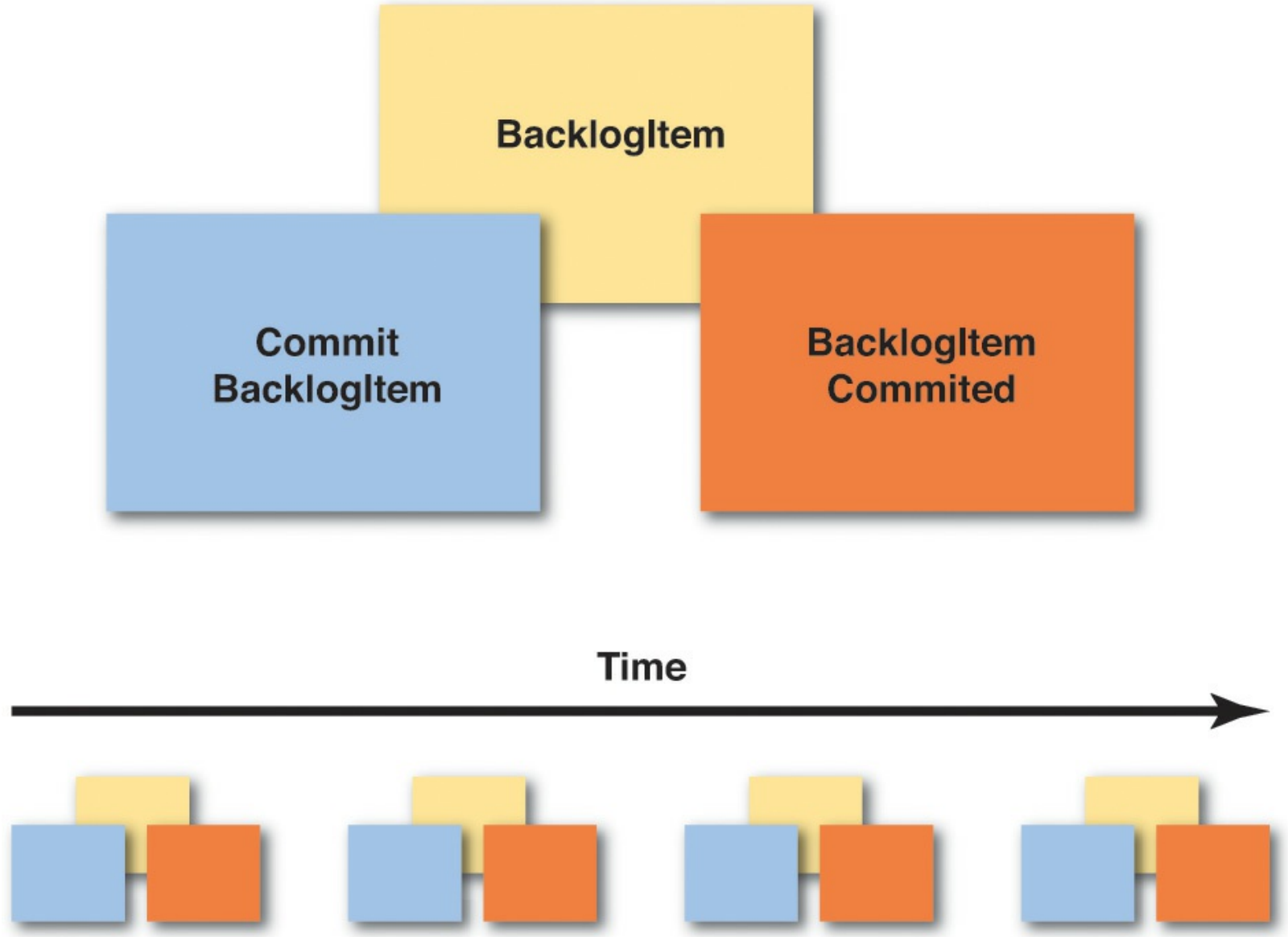


2. *Create the Commands that cause each Domain Event.* Sometimes a *Domain Event* will be the outcome of a happening in another system, and it will flow into your system as a result. Still, often a *Command* will be the outcome of some user gesture, and that *Command* , when carried out, will cause a *Domain Event.* The *Command* should be stated in the imperative, such as `CreateProduct` and `CommitBacklogItem` . These are some basic guidelines:

- On the light blue sticky notes, write the name of the *Command* that causes each corresponding *Domain Event.* For example, if you have a *Domain Event* named `BacklogItemCommitted` , the corresponding *Command* that causes that event is named `CommitBacklogItem` .
- Place the light blue sticky note of the *Command* just to the left of the *Domain Event* that it causes. They are associated in pairs: *Command/Event* , *Command/Event* , *Command/Event* , and so on. Remember that some *Domain Events* will occur because of time limits being reached and so may not have a corresponding *Command* that explicitly causes them.
- If there is a specific user role that performs an action, and it is important to specify, you can place a small, bright yellow sticky note on the lower left corner of the light blue *Command* with a stick figure and the name of the role. In the above figure, "Product Owner" would be the role that performs the *Command.*
- Sometimes a *Command* will cause a *Process* to be run. This could be a single step or multiple complex steps. Each *Command* that causes a *Process* to be executed should be captured and named on a lilac sticky note. Draw a line with an arrowhead from the *Command* to the named *Process* (lilac sticky note). The *Process* will actually cause one or more *Commands* and subsequent *Domain Events* , and if you know what those are now, create sticky notes for them and show them emitting from the *Process.*
- Continue to move from left to right in time order just as you did when first creating each of the *Domain Events.*

- It is possible that creating *Commands* will cause you to think about *Domain Events* (as above when discovering lilac *Processes* , or other ones) that you didn't previously envision. Go ahead and address this discovery by placing the newly discovered *Domain Event* on the modeling surface along with its corresponding *Command.*
- You may also find that there is only one *Command* that causes multiple *Domain Events.* That's fine; model the one *Command* and place it to the left of the multiple *Domain Events* that it causes.

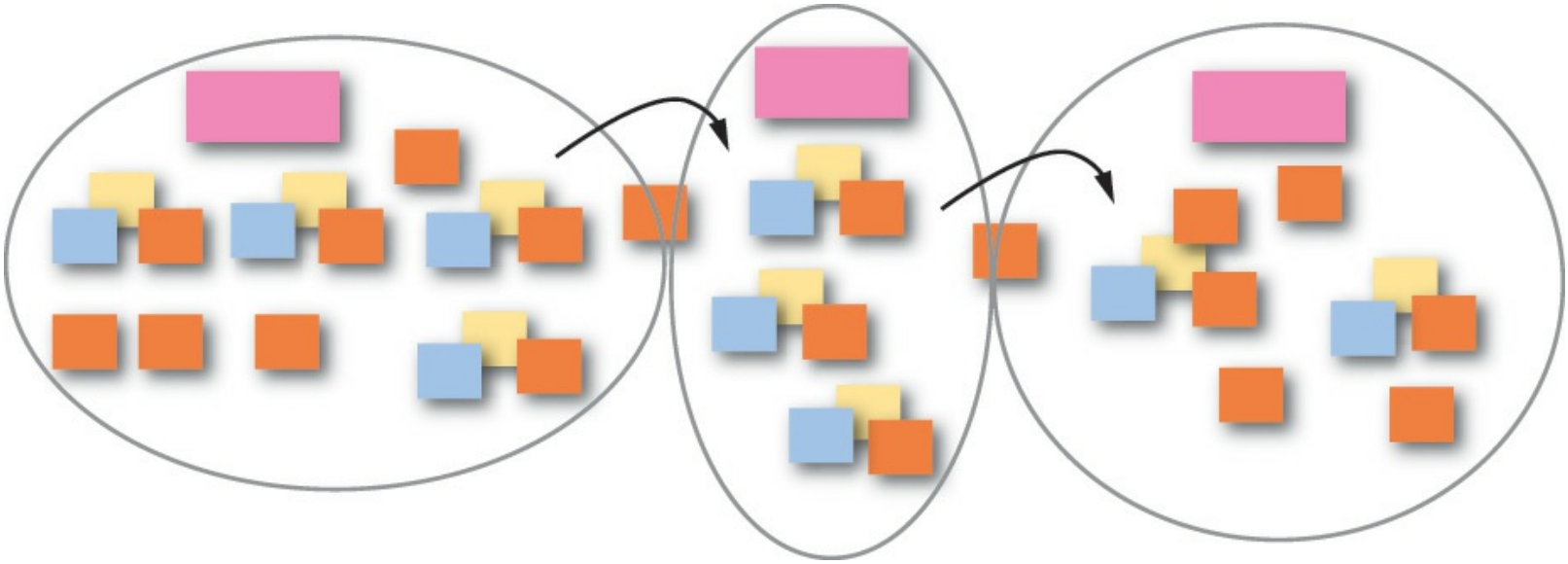
Once you have all of the *Commands* associated with the *Domain Events* that they cause, you are ready to move on to the next step.



3. *Associate the Entity/Aggregate on which the Command is executed and that produces the Domain Event outcome.* This is the data holder where *Commands* are executed and *Domain Events* are emitted. *Entity* relationship diagrams are often the first and most popular step in today's IT world, but it is a big mistake to start here. Business people don't understand them well, and they can shut down conversations quickly. In fact, this step has been relegated to third place in *Event Storming* , because we are more focused on the business process than on the data. Even so, we do need to think about data at some point, and that point is now. At this stage, business experts will likely understand that the data comes into play. Here are some guidelines for modeling the *Aggregates* :

- If the business people don't like the word *Aggregate* , or if it confuses them in any way, you should use another name. Usually they can understand *Entity* , or you could just call it *Data.* The important thing is that the sticky allows the team to communicate clearly about the concept that it represents. Use the pale yellow sticky notes for all *Aggregates* and write the name of an *Aggregate* on each sticky note. This is a noun, such as `Product` or `BacklogItem` . You will do this for each *Aggregate* in your model.
- Place the *Aggregate* sticky note behind and slightly above the *Command* and *Domain Event* pairs. In other words, you should be able to read the noun written on the *Aggregate* sticky note, but the *Command* and *Domain Event* pairs should adhere to the lower part of the *Aggregate* sticky to indicate that they are associated. If you really want to put a little space between the stickies that's fine, but just be clear which *Commands* and *Domain Events* belong to which *Aggregate.*
- As you move across your business process timeline, you will probably find that *Aggregates* are repeatedly used. Don't rearrange your timeline in order to move all *Command/Event* pairs under a single *Aggregate* sticky note. Rather, create the same *Aggregate* noun on multiple sticky notes and place them repeatedly on the timeline where the corresponding *Command/Event* pairs occur. The main point is to model the business process; the business process occurs over time.
- It's possible that as you think about the data associated with various actions, you may discover new *Domain Events.* Don't ignore these. Rather, place the newly discovered *Domain Events* along with the corresponding *Commands* and *Aggregates* on the modeling surface. You may also discover that some of the *Aggregates* are too complex, and you need to break these into a managed *Process* (lilac sticky). Don't ignore these opportunities.

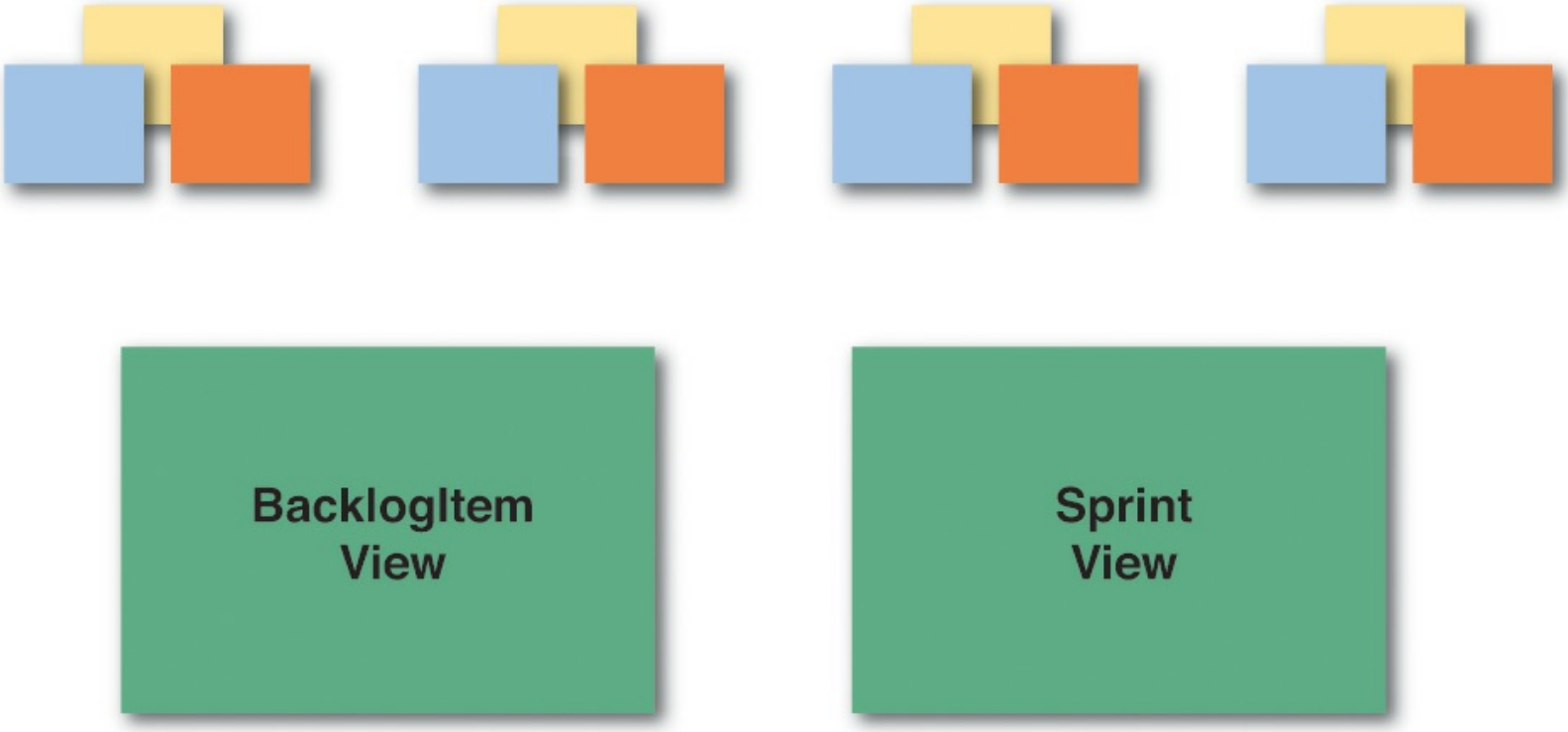
Once you have completed this part of the design stage, you are approaching some of the extra steps that you can perform if you choose to. Also understand that if you are using *Event Sourcing* , as described in the previous chapter, you have already come a long way toward understanding your *Core Domain* implementation, because there is a large overlap in *Event Storming* and *Event Sourcing.* Of course, the closer your storming is to the big picture, the further it potentially is from actual implementation. Still, you can use this same technique to reach a design-level view. In my experience teams tend to move in and out of big-picture and design-level within the same sessions. In the end your need to learn certain details will drive you beyond the big picture to reach a design-level model where it is essential.



4. *Draw boundaries and lines with arrows to show flow on your modeling surface.* You have very likely discovered that there are multiple models in play, and *Domain Events* that flow between models, in your *Event Storming* sessions. Here's how to deal with that:

- In summary, you will very likely find boundaries under the following conditions: departmental divisions, when different business people have conflicting definitions for the same term, or when a concept is important but not really part of the *Core Domain.*
- You can use your black marker pens to draw on the paper modeling surface. Show context and other boundaries. Use solid lines for *Bounded Contexts* and dashed lines for *Subdomains. Obviously drawing boundaries on the paper is permanent, so be sure you understand this level of detail before wading in. If you want to start by bounding models with less permanence, use the pink stickies to mark general areas and withhold drawing boundaries with permanent markers until your confidence justifies it.*
- Place the pink sticky notes inside various boundaries and put the name that applies inside the boundary on those sticky notes. This names your *Bounded Contexts.*
- Draw lines with arrowheads to show the direction of *Domain Events* flowing between *Bounded Contexts.* This is an easy way to communicate how some *Domain Events* arrive in your system without being caused by a *Command* in your *Bounded Context.*

Any other details about these steps should be intuitively obvious. Just use boundaries and lines to communicate.



5. *Identify the various views that your users will need to carry out their actions, and important roles for various users.*
   - You won't necessarily need to show every view that your user interface will provide, or any at all for that matter. If you decide to show any views, they should be those that are significant and require some special care in creating. These view artifacts can be represented by green sticky notes on the modeling surface. If it helps, draw a quick mockup (or wireframe) of the user interface views that are most important.
   - You can also use bright yellow sticky notes to represent various important user roles. Again, show these only if you need to communicate something of significance about the user's interaction with the system, or something that the system does for a specific role of user.

It could well be that the fourth and fifth steps are all the extras you will need to incorporate with your *Event Storming* exercises.

Other Tools

Of course this doesn't prevent you from experimenting, such as placing other drawings on your modeling surface and trying other modeling steps in your *Event Storming* session. Remember, this is about learning and communicating a design. Use whatever tools you need to model as a close-knit team. Just be careful to reject ceremony, because that's going to cost a lot. Here are some other ideas:

Introduce high-level executable specifications that follow the given/when/then approach. These are also known as acceptance tests. You can read more about this in the book *Specification by Example* by Gojko Adzic [Specification] , and I provide an example in Chapter 2 , "Strategic Design with Bounded Contexts and the Ubiquitous Language ." Just be careful not to go overboard with these, where they become all-consuming and take precedence over the actual domain model. I estimate that it requires somewhere in the range of 15% to 25% more time and effort to use and maintain executable specifications instead of common unit-testing-based approaches (also demonstrated in Chapter 2), and it is easy to get caught up in keeping the specifications relevant to the current business direction as the model changes over time.

Try *Impact Mapping* [Impact Mapping] to make sure the software you are designing is a *Core Domain* and not some less important model. This is a technique also defined by Gojko Adzic.

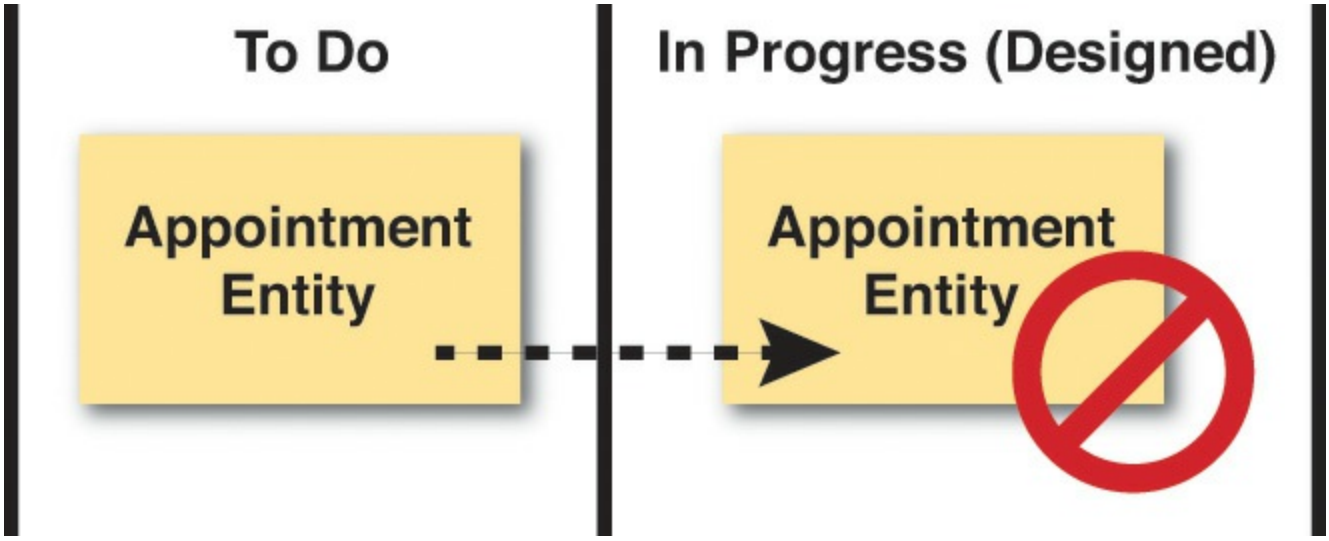
Look into *User Story Mapping* by Jeff Patton [User Story Mapping] . This is used to place your focus on the *Core Domain* and understand what software features you should be investing in.

The previous three add-on tools have a large overlap with the DDD philosophy and would be quite suitable to introduce into any DDD project. All of them are meant to be used in a highly accelerated project, are low ceremony, and are very cheap to use.

Managing DDD on an Agile Project

I previously mentioned that there has been a movement around what is called *No Estimates*. This is an approach that rejects typical estimation approaches such as story points or task hours. It focuses on delivering value over controlling cost, and not estimating any task that would likely require only a few months to complete. I don't dismiss this approach. Yet, at the time of writing this, the clients I work with are still required to provide estimates and to timebox tasks, such as programming effort needed to implement even fine-grained features. If *No Estimates* works for you and your project situation, use it.

I am also aware that some in the DDD community have basically defined their own process or process execution framework for using DDD and performing with it on a project. This may work well and be effective when it is accepted by a given team, but it can be more difficult to get buy-in from organizations that have already invested in an agile execution framework, such as Scrum.



I have observed that recently Scrum has come under considerable criticism. While I don't take sides in this criticism, I will openly state that often or even most times Scrum is being misused. I have already mentioned the tendency for teams to "design" using what I call "the task-board shuffle." It's just not the way Scrum was meant to be used on a software project. And, to repeat myself again, *knowledge acquisition* is both a Scrum tenet and a major goal of DDD but is largely ignored in exchange for relentless delivery with Scrum. Even so, Scrum is still heavily used in our industry, and I doubt that it will be displaced anytime soon.

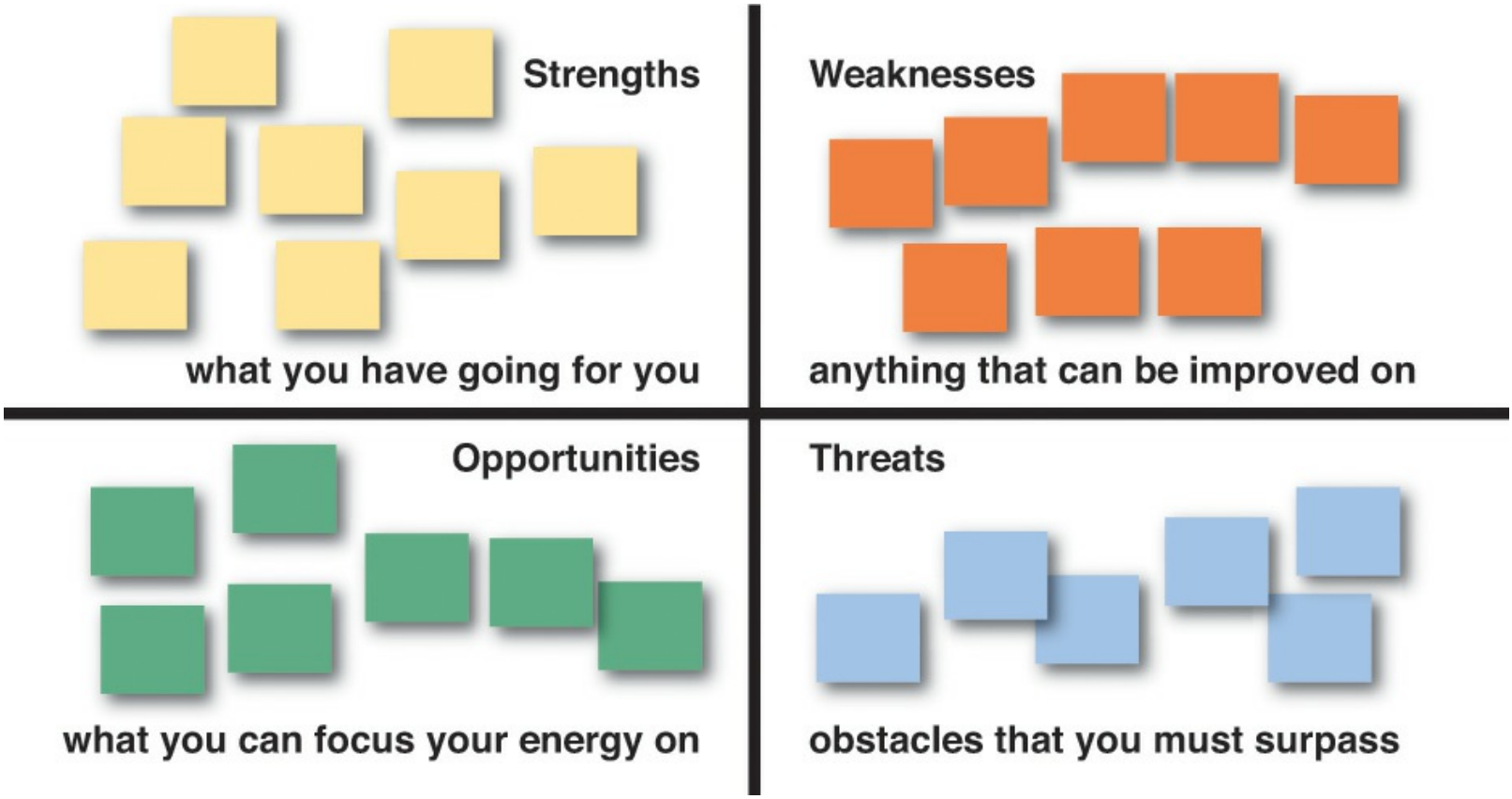
Therefore, what I will do here is to show you how you can make DDD work in a Scrum-based project. The techniques I show you should be equally applicable with other agile project approaches, such as when using Kanban. There is nothing here that is exclusive to Scrum, although some of the guidance is stated in terms of Scrum. And since many of you will already be familiar with Scrum by putting it into practice in some form, most of my guidance here will be regarding the domain model

and learning, experimenting, and designing with DDD. You will need to look elsewhere for general guidance on using Scrum, Kanban, or another agile approach.

Where I use the term *task* or *task board* , this should be compatible with agile in general, and even Kanban. Where I use the term *sprint* , I will also try to include the words *iteration* for agile in general and *WIP* (work in progress) as a reference to Kanban. It may not always be a perfect fit, as I am not trying to define an actual process here. I hope you will simply benefit from the ideas and find a way to apply them appropriately in your specific agile execution framework.

First Things First

One of the most important means to successfully employing DDD on a project is to hire good people. There is simply no replacement for good people, and above-average developers for that matter. DDD is an advanced philosophy and technique for developing software, and it calls for above-average developers, even very good developers, to put it to use. Never underestimate the importance of hiring the right people with the right skills and self-motivation.



Use SWOT Analysis

In case you are unfamiliar with SWOT analysis [SWOT] , it stands for Strengths, Weaknesses, Opportunities, and Threats. SWOT analysis is a way for you to think about your project in very specific ways, gaining maximum knowledge as early as possible. Here are the basic ideas behind what you are looking to identify on a project:

- *Strengths:* characteristics of the business or project that give it an advantage over others
- *Weaknesses:* characteristics that place the business or project at a disadvantage relative to others
- *Opportunities:* elements that the project could exploit to its advantage
- *Threats:* elements in the environment that could cause trouble for the business or project

At any time on any Scrum or other agile project you should feel free and inclined to use SWOT analysis to determine your project's current situation:

1. Draw a large matrix with four quadrants.
2. Going back to the sticky notes, choose a different color for each of the four SWOT quadrants.
3. Now, identity the Strengths of your project, the Weaknesses of your project, the Opportunities on your project, and the Threats to your project.
4. Write these on the sticky notes and place them in the matrix within the appropriate quadrant.
5. Use these SWOT characteristics of the project (we are particularly thinking domain model here) to plan what you are going to do about them. The next steps you take to promote the good areas and mitigate the troublesome areas could be critical to your success.

You will have the opportunity to place these actions on the task board as you perform project planning, as discussed later.

Modeling Spikes and Modeling Debt

Does it surprise you to learn that you can have modeling spikes and modeling debt to pay on a DDD project?

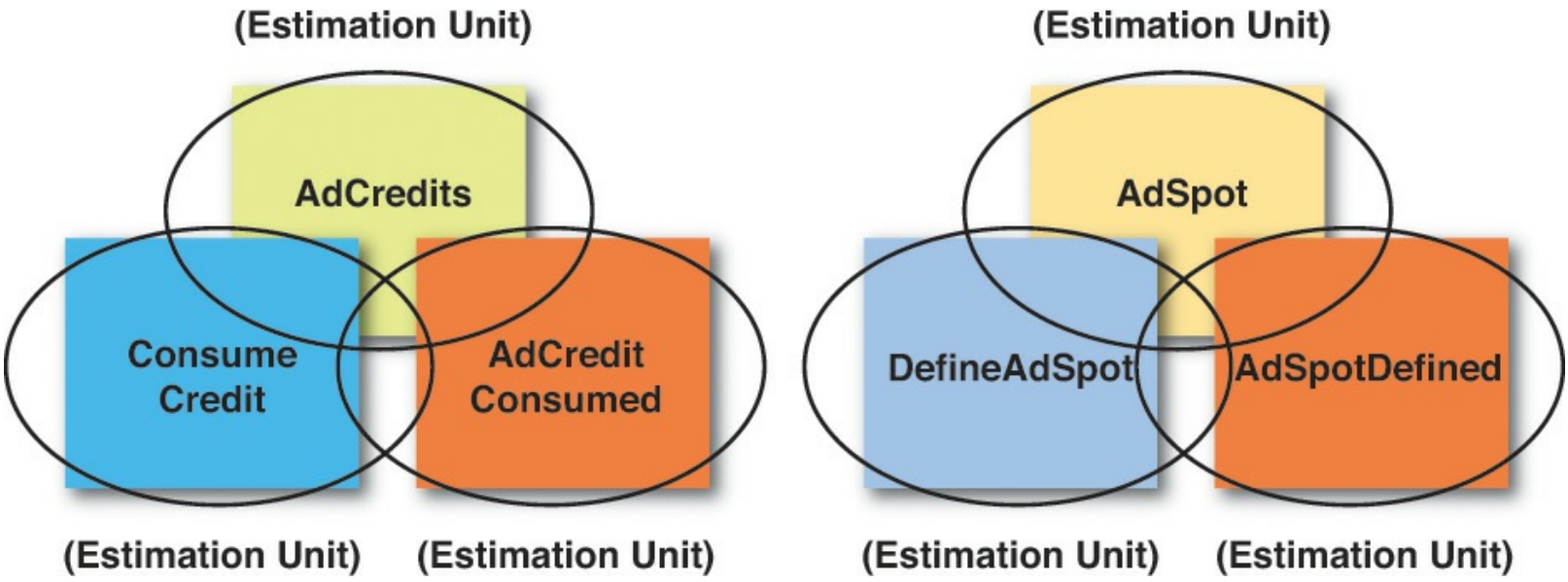
One of the best things you can do at the inception of a project is to use *Event Storming* . This and related modeling experiments would constitute a modeling spike. You will have to "buy" knowledge about your Scrum product, and sometimes the payment is a spike, and a spike during project inception is almost certain. Still, I have already shown you how using *Event Storming* can greatly reduce the cost of necessary investment.

For certain, you can't expect to model your domain perfectly from the start, even if you think in terms of the inception of your project as having a valuable modeling spike. You won't even be perfect as you use *Event Storming* . For one thing, business and our understanding of it change over time, and so will your domain model.

Furthermore, if you intend to timebox your modeling efforts as tasks on a task board, expect to incur some modeling debt during each sprint (or iteration, or WIP). You simply won't have time to carry out every desired modeling task to perfection when you are timeboxed. For one thing, you will start a design and realize after experimentation that the design you have does not fit the business needs as well as you expected. Yet the time limit that you are under will require you to move on.

The worst thing you could do now is to just forget everything that you learned from the modeling efforts that called out for a different, improved design. Rather, make a note that this needs to go into a later sprint (or iteration, WIP). This can be brought to your retrospective meeting[1] and turned in as a new task at your next sprint planning meeting (or iteration planning meeting, or added to the Kanban queue).

1 . In Kanban you can actually have retrospectives every day, so don't wait for long to present the need to improve the model.



Identifying Tasks and Estimating Effort

Event Storming is a tool that can be used at any time, not just during project inception. As you work in an *Event Storming* session, you will naturally create a number of artifacts. Each of the *Domain Events* , *Commands* , and *Aggregates* that you storm out in your paper model can be used as estimation units. How so?

Component Type	Easy (Hours)	Moderate (Hours)	Complex (Hours)
Domain Event	0.1	0.2	0.3
Command	0.1	0.2	0.3
Aggregate	1	2	4
. . .	. . .	. . .	. . .

One of the easiest and most accurate ways to estimate is by using a metrics-based approach. As you see here, create a simple table with estimation units for each component type that you will need to implement. This will take the guesswork out of estimates and provide science around the process of creating estimations of effort. Here is how the table works:

1. Create one column for *Component Type* to describe the specific kind of component for which the estimation units are defined.
2. Create three other columns, one each for *Easy* , *Moderate* , and *Complex*. These columns will reflect the estimation unit, which is in hours or fractions of hours, for the specific unit type.
3. Now create one row for each component type in your architecture. Shown are *Domain Event* , *Command* , and *Aggregate* types. However, don't limit yourself to those. Create a row for the

various user interface components, services, persistence, *Domain Event* serializers and deserializers, and so on. Feel free to create a row for every single kind of artifact that you will create in source code. (If, for example, you normally create a *Domain Event* serializer and deserializer along with each *Domain Event* as a composite step, assign an estimation value to *Domain Events* that reflects the creation of all of those components together in each column.)

4. Now fill in the hours or fraction of an hour needed for each level of complexity: easy, moderate, and complex. These estimates not only include the time needed for implementation but may also include further design and testing efforts. Make these accurate, and be realistic.
5. When you know the backlog item tasks (WIP) that you will work on, obtain a metric for each of the tasks and identify it clearly. You might use a spreadsheet for this.
6. Add up all the estimation units for all components in the current sprint (iteration or WIP), and this becomes your total estimation.

As you execute each sprint (iteration or WIP), tune your metrics to reflect the hours or fractions of hours that were actually required.

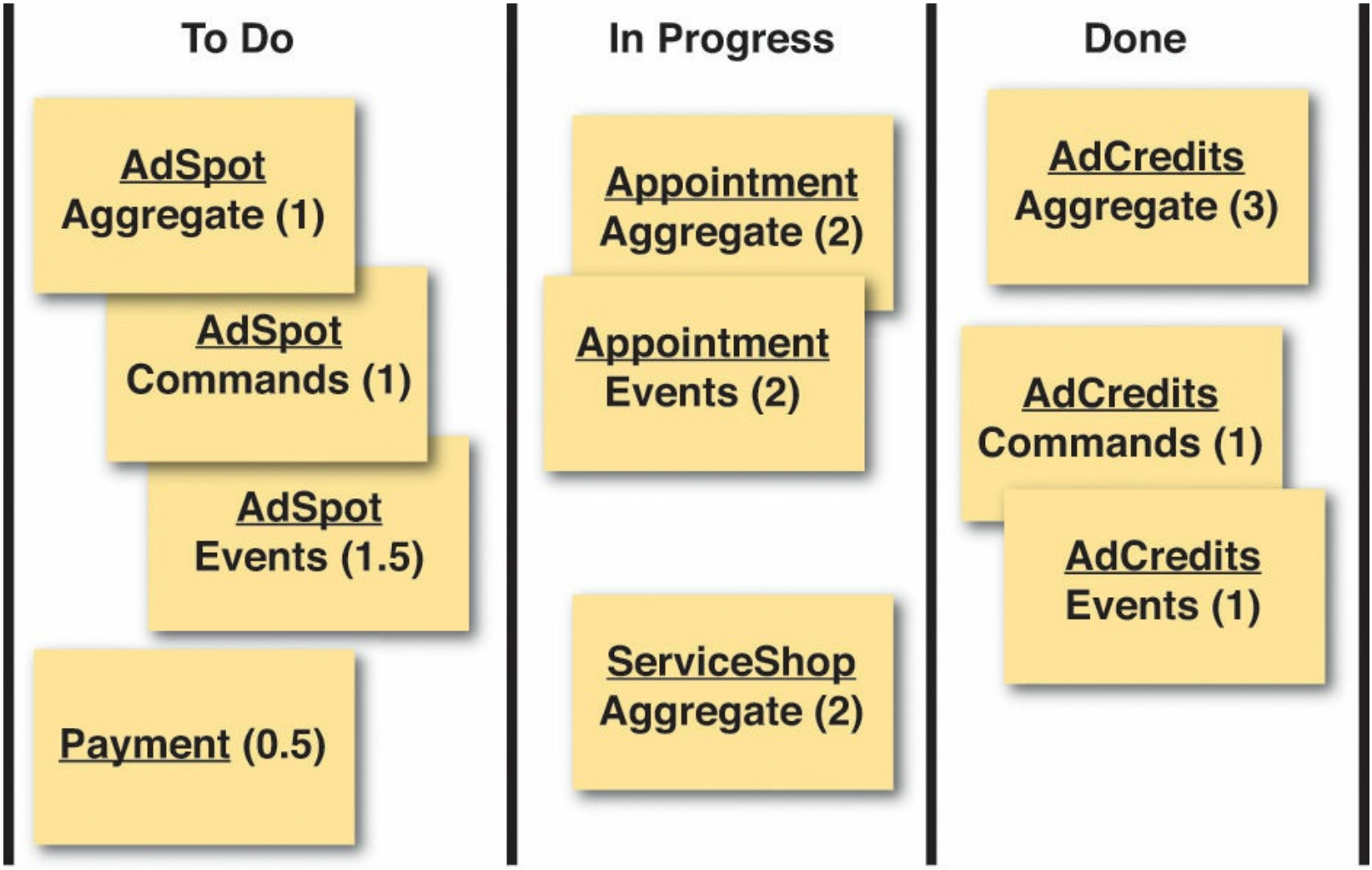
If you are using Scrum and you have come to detest hour estimates, understand that this approach is much more forgiving and also much more accurate. As you learn your cadence, you will fine-tune your estimation metrics to be more accurate and realistic. It might require a few sprints to get it right. Also realize that as time and experience progress, you will probably either tune your numbers lower or use the *Easy* or *Moderate* columns more readily.

If you are using Kanban and you think that estimates are completely fallacious and unnecessary, ask yourself a question: How do I know how to determine an accurate WIP in the first place in order to correctly limit our work queue? Regardless of what you may think, you are still estimating the effort involved and hoping that it is correct. Why not add a little science to the process and use this simple and accurate estimation approach?

A Comment on Accuracy

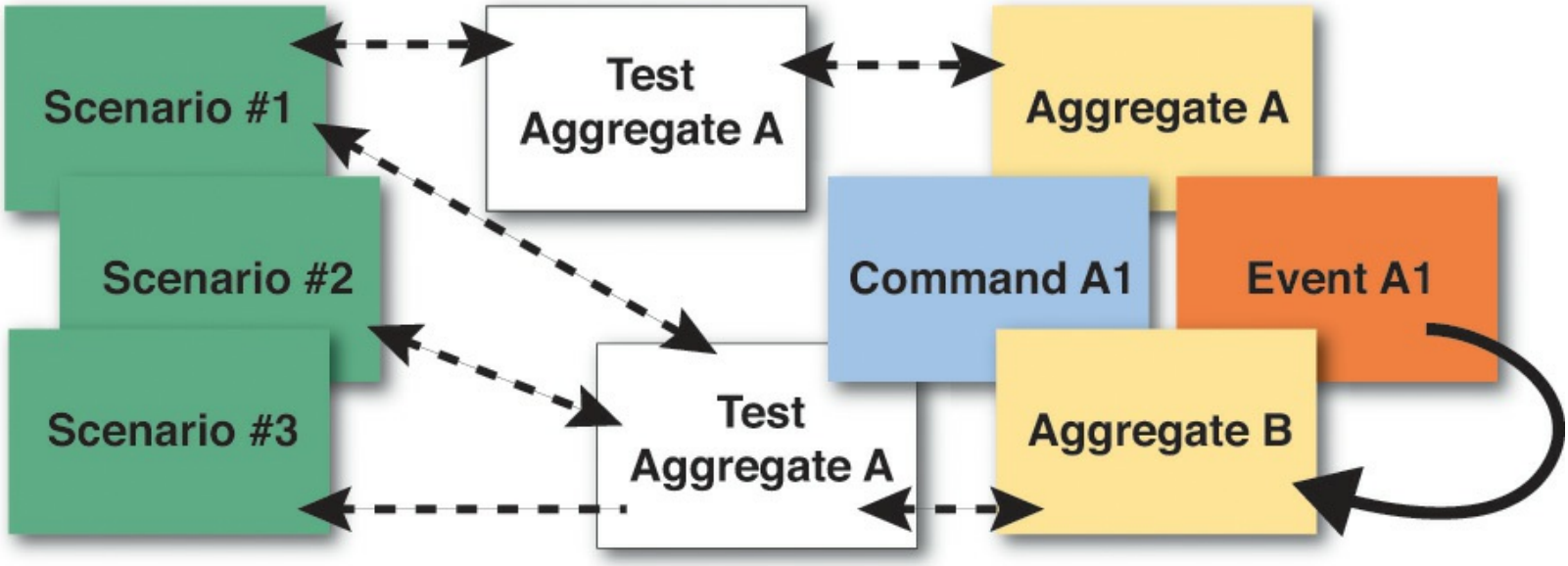
This approach works. On one large corporate program the organization demanded estimates for a large and complex project within the overall program. Two teams were assigned to this task. First there was a team of high-cost consultants who worked with Fortune 500 companies to estimate and manage projects. They were accountants and had doctorates and were outfitted with everything that would both intimidate and give them the clear advantage. The second team of architects and developers was empowered with this metrics-based estimation process. The project was in the $20 million range, and in the end when both estimates came in, they were within approximately $200,000 of each other (the technical team's being the slightly lower estimate). Not bad for techies.

You should be able to get within 20% accuracy on long-term estimates, and much better on shorter-term estimates, such as for sprints, iterations, and WIP queues.



Timeboxed Modeling

Now that you have estimates for each component type, you can base your tasks directly off of those components. You might choose to keep each component as a single task with a number of hours or fraction thereof, or you might choose to break your tasks down a bit further. However, I suggest being careful with breaking tasks down to be too fine-grained, so as not to make the task board overly complex. As shown previously, it might even be best to combine all of the *Commands* and all of the *Domain Events* used by a single *Aggregate* into a single task.



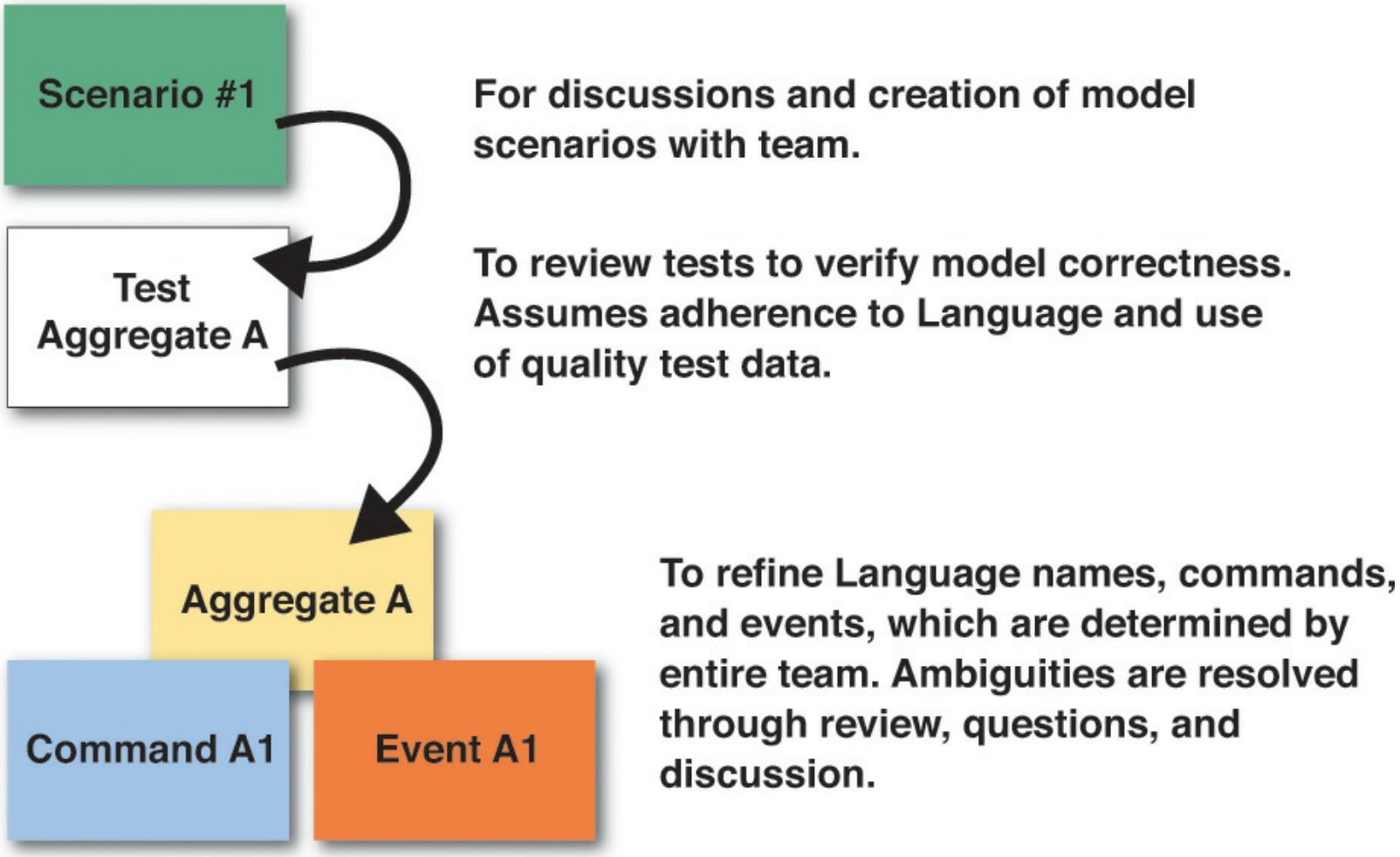
How to Implement

Even with artifacts identified by *Event Storming* , you will not necessarily have all the knowledge you need to work on a specific domain scenario, story, and use case. If more is needed, be sure to include time for further knowledge acquisition in your estimates. But time for what? Recall that in Chapter 2 I introduced you to creating concrete scenarios around your domain model. This can be one of the best ways to acquire knowledge about your *Core Domain* , beyond what you can get out of *Event Storming*. Concrete scenarios and *Event Storming* are two tools that should be used together. Here's how it works:

- Perform a quick session of *Event Storming* , perhaps just for an hour or so. You will almost certainly discover that you need to develop more concrete scenarios around some of your quick modeling discoveries.
- Partner with a *Domain Expert* to discuss one or more concrete scenarios that need to be refined. This identifies how the software model will be used. Again, these are not just procedures but should be stated with the goal of identifying actual domain model elements (e.g., objects), how the elements collaborate, and how they interact with users. (Refer to Chapter 2 as needed.)
- Create a set of acceptance tests (or executable specifications) that exercise each of the scenarios. (Refer to Chapter 2 as needed.)
- Create the components to allow the tests/specifications to execute. Iterate (briefly and quickly) as you refine the tests/specifications and the components until they do what your *Domain Expert* expects.
- Very likely some of the iteration (brief and quick) will cause you to consider other scenarios, create additional tests/specifications, and refine existing and create new components.

Continue this until you have acquired all the knowledge necessary to meet a limited business objective, or until your timebox expires. If you haven't reached the desired point, make sure to incur modeling debt so that this can be addressed in the (ideally near) future.

Yet how much time will you need from *Domain Experts* ?



Interacting with Domain Experts

One of the major challenges of employing DDD is getting time with *Domain Experts* , and without overdoing it. Many times those who are *Domain Experts* on a project will have loads of other responsibilities, hours of meetings, and possibly travel. With such potential absences from the modeling environment, it can be difficult to find enough time with them. So, we had better make the time we use count and limit it to just what is necessary. Unless you make modeling sessions fun and efficient, you stand a good chance of losing their help at just the wrong time. If they find it valuable, enlightening, and rewarding, you will likely establish the strong partnership that you will need.

So the first questions to answer are "When do we need time with *Domain Experts* ? What tasks do they need to help us perform?"

- Always include *Domain Experts* in *Event Storming* activities. Developers will always have a lot of questions, and *Domain Experts* will have the answers. Make sure they are in the *Event Storming* sessions together.
- You will need *Domain Experts'* input on discussions and the creation of model scenarios. See Chapter 2 for examples.
- *Domain Experts* will be needed to review tests to verify model correctness. This assumes that the developers have already made a conscientious effort to adhere to the *Ubiquitous Language* and to use quality, realistic test data.
- You will need *Domain Experts* to refine the *Ubiquitous Language* and its *Aggregate* names, *Commands* , and *Domain Events* , which are determined by the entire team. Ambiguities are resolved through review, questions, and discussion. Even so, *Event Storming* sessions should

have already resolved most of the questions about the *Ubiquitous Language*.

So, now that you know what you will need from *Domain Experts* , how much time should you require of them for each of these responsibilities?

- *Event Storming* sessions should be limited to a few hours (two or three) each. You may need to hold sessions on consecutive days, such as for three or four days.
- Block out generous amounts of time for scenario discussion and refinement, but try to maximize the time for each scenario. You should be able to discuss and iterate on one scenario over perhaps 10 to 20 minutes of time.
- For tests, you will need some time with *Domain Experts* to review what you have written. But don't expect them to sit there as you write the code. Maybe they will, and that's a bonus, but don't expect it. Accurate models require less time to review and verify. Don't underestimate the ability of *Domain Experts* to read through a test with your help. They can do it, especially if the test data is realistic. Your tests should allow the *Domain Expert* to understand and verify around one test every one to two minutes, or thereabouts.
- During test reviews, *Domain Experts* can provide input on *Aggregates* , *Commands* , and *Domain Events* , and perhaps other artifacts, as to how they adhere to the *Ubiquitous Language*. This can be accomplished in brief amounts of time.

This guidance should help you use just the right amount of time with *Domain Experts* , and to limit the amount of time that you need to spend with them.

Summary

In summary, in this chapter you learned:

- About *Event Storming* , how it can be used, and how to perform sessions with your team, all with a view to accelerating your modeling efforts
- About other tools that can be used along with *Event Storming*
- How to use DDD on a project and how to manage estimates and the time you need with *Domain Experts*

For an exhaustive reference covering the implementation of DDD on projects, see *Implementing Domain-Driven Design* [IDDD] .

References

[BDD] North, Dan. “Behavior-Driven Development.” 2006.
http://dannorth.net/introducing-bdd/ .

[Causal] Lloyd, Wyatt, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. “Don’t Settle for Eventual Consistency: Stronger Properties for Low-Latency Geo-replicated Storage.”
http://queue.acm.org/detail.cfm?id=2610533 .

[DDD] Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Boston: Addison-Wesley, 2004.

[Essential Scrum] Rubin, Kenneth S. *Essential Scrum: A Practical Guide to the Most Popular Agile Process.* Boston: Addison-Wesley, 2012.

[IDDD] Vernon, Vaughn. *Implementing Domain-Driven Design.* Boston: Addison-Wesley, 2013.

[Impact Mapping] Adzic, Gojko. *Impact Mapping: Making a Big Impact with Software Products and Projects.* Provoking Thoughts, 2012.

[Microservices] Newman, Sam. *Building Microservices.* Sebastopol, CA: O’Reilly Media, 2015.

[Reactive] Vernon, Vaughn. *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka.* Boston: Addison-Wesley, 2015.

[RiP] Webber, Jim, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture.* Sebastopol, CA: O’Reilly Media, 2010.

[Specification] Adzic, Gojko. *Specification by Example: How Successful Teams Deliver the Right Software.* Manning Publications, 2011.

[SRP] Wikipedia. “Single Responsibility Principle.”
http://en.wikipedia.org/wiki/Single_responsibility_principle .

[SWOT] Wikipedia. “SWOT Analysis.”
https://en.wikipedia.org/wiki/SWOT_analysis .

[User Story Mapping] Patton, Jeff. *User Story Mapping: Discover the Whole Story, Build the Right Product.* Sebastopol, CA: O’Reilly Media, 2014.

[WSJ] Andreessen, Marc. “Why Software Is Eating the World.” *Wall Street Journal* , August 20, 2011.

[Ziobrando] Brandolini, Alberto. “Introducing EventStorming.”
https://leanpub.com/introducing_eventstorming .

Index

A

B

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

REGISTER YOUR PRODUCT at informit.com/register

Access Additional Benefits and SAVE 35% on Your Next Purchase

- Download available product updates.
- Access bonus material when applicable.
- Receive exclusive offers on new editions and related products. (Just check the box to hear from us when setting up your account.)
- Get a coupon for 35% for your next purchase, valid for 30 days. Your code will be available in your InformIT cart. (You will also find it in the Manage Codes section of your account page.)

Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.

InformIT.com–The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com you can

- Shop our books, eBooks, software, and video training.
- Take advantage of our special offers and promotions (informit.com/promotions).
- Sign up for special offers and content newsletters (informit.com/newsletters).
- Read free articles and blogs by information technology experts.
- Access thousands of free chapters and video lessons.

Connect with InformIT–Visit informit.com/community

Learn about InformIT community events and programs.

informIT.com
the trusted technology learning source

Addison-Wesley • Cisco Press • IBM Press • Microsoft Press • Pearson IT Certification • Prentice Hall • Que • Sams • VMware Press

Code Snippets

```
Scenario:  The product owner commits a backlog item to a sprint
   Given a backlog item that is scheduled for release
   And the product owner of the backlog item
   And a sprint for commitment
   And a quorum of team approval for commitment
   When the product owner commits the backlog item to the sprint
   Then the backlog item is committed to the sprint
   And the backlog item committed event is created
```

```
/*
 The product owner commits a backlog item to a sprint.
 The backlog item may be committed only if it is already
 scheduled for release, and if a quorum of team members
 have approved commitment. When the commitment completes,
 notify the sprint to which it is now committed.
*/

[Test]
public void ShouldCommitBacklogItemToSprint()
{
  // Given
  var backlogItem = BacklogItemScheduledForRelease();

  var productOwner = ProductOwnerOf(backlogItem);

  var sprint = SprintForCommitment();

  var quorum = QuorumOfTeamApproval(backlogItem, sprint);

  // When
  backlogItem.CommitTo(sprint, productOwner, quorum);

  // Then
  Assert.IsTrue(backlogItem.IsCommitted());

  var backlogItemCommitted =
    backlogItem.Events.OfType<BacklogItemCommitted>().SingleOrDefault();

  Assert.IsNotNull(backlogItemCommitted);
}
```