



DOMAIN-DRIVEN DESIGN DISTILLED

VAUGHN VERNON

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



Domain-Driven Design Distilled

This page intentionally left blank



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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

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

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2016936587

Copyright © 2016 Pearson Education, Inc.

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

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!

This page intentionally left blank

Contents

Preface	xi
Acknowledgments	xv
About the Author	xvii
Chapter 1 DDD for Me	1
Will DDD Hurt?	2
Good, Bad, and Effective Design	3
Strategic Design	7
Tactical Design	8
The Learning Process and Refining Knowledge	9
Let's Get Started!	10
Chapter 2 Strategic Design with Bounded Contexts and the Ubiquitous Language	11
Domain Experts and Business Drivers	17
Case Study	21
Fundamental Strategic Design Needed	25
Challenge and Unify	29
Developing a Ubiquitous Language	34
Putting Scenarios to Work	38
What about the Long Haul?	40
Architecture	41
Summary	44

Chapter 3 Strategic Design with Subdomains	45
What Is a Subdomain?	46
Types of Subdomains	46
Dealing with Complexity	47
Summary	50
Chapter 4 Strategic Design with Context Mapping	51
Kinds of Mappings	54
Partnership	54
Shared Kernel	55
Customer-Supplier	55
Conformist	56
Anticorruption Layer	56
Open Host Service	57
Published Language	58
Separate Ways	58
Big Ball of Mud	59
Making Good Use of Context Mapping	60
RPC with SOAP	61
RESTful HTTP	63
Messaging	65
An Example in Context Mapping	70
Summary	73
Chapter 5 Tactical Design with Aggregates	75
Why Used	76
Aggregate Rules of Thumb	81
Rule 1: Protect Business Invariants inside Aggregate Boundaries	82
Rule 2: Design Small Aggregates	83
Rule 3: Reference Other Aggregates by Identity Only	84
Rule 4: Update Other Aggregates Using Eventual Consistency	85
Modeling Aggregates	88
Choose Your Abstractions Carefully	93

Right-Sizing Aggregates	95
Testable Units	97
Summary	98
Chapter 6 Tactical Design with Domain Events	99
Designing, Implementing, and Using Domain Events	100
Event Sourcing	107
Summary	109
Chapter 7 Acceleration and Management Tools	111
Event Storming	112
Other Tools	124
Managing DDD on an Agile Project	125
First Things First	126
Use SWOT Analysis	127
Modeling Spikes and Modeling Debt	128
Identifying Tasks and Estimating Effort	129
Timeboxed Modeling	132
How to Implement	133
Interacting with Domain Experts	134
Summary	136
References	137
Index	139

This page intentionally left blank

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!

This page intentionally left blank

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.

This page intentionally left blank

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.

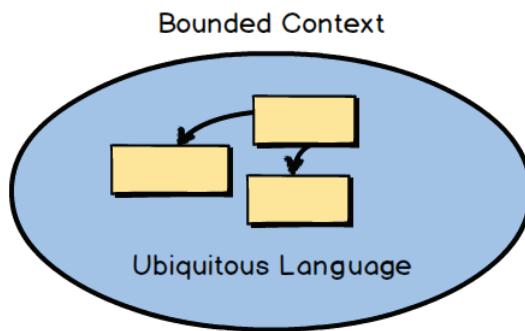
In Scrum, *knowledge acquisition* is done through experimentation and collaborative learning and is referred to as “buying information” [Essential Scrum]. Knowledge is never free, but in this book I do provide ways for you to accelerate how you come by it.

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

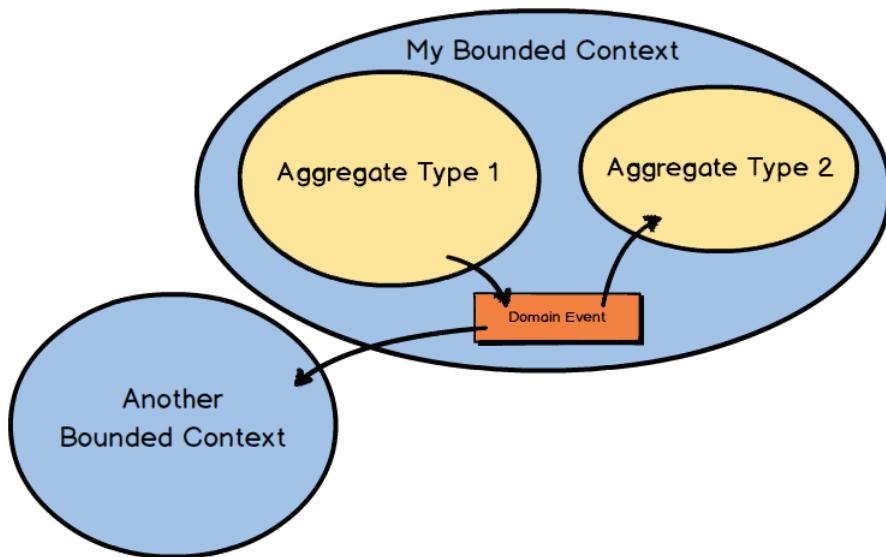
We begin with the all-important strategic design. You really cannot apply tactical design in an effective way unless you begin with strategic design. Strategic design is used like broad brushstrokes prior to getting into the details of implementation. It highlights what is strategically important to 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.”

Index

A

Abstractions

- modeling Aggregates, 93–95
- software design problems, 5

Acceleration and management tools

- Event Storming, 112–124
- managing DDD projects. *See*
 - Managing DDD on Agile Project
- other tools, 124
- overview of, 111–112
- summary, 136

Acceptance tests

- implementing DDD on Agile Project, 134
- using with Event Storming, 124
- validating domain model, 39–40

Accuracy, managing in project, 130–131

Actor model

- caching Aggregates' state, 109
- handling transactions, 78
- using with DDD, 43

Adapters, 41–42

Aggregate Root, defined, 77

Aggregates

- associating with Commands, 120–122
- choosing abstractions carefully, 93–95
- creating Big Ball of Mud via, 59
- designing as testable units, 97–98
- Domain Experts refining, 134–136

Event Sourcing Domain Events for, 107–109

- identifying tasks/estimating effort, 129–131

integrating using messaging, 65–70

modeling, 88–93

overview, 75

right-sizing, 95–97

scenario using, 104–105

summary, 98

in tactical design, 8–9

transactions and, 78–81

why they are used, 76–78

Aggregates, design rules

- commit one instance in one transaction, 79–81

protect business invariants within boundaries, 82

reference by identity only, 84–85

small size, 83–84

update with eventual consistency, 85–88

Agile Project Management Context and Context Mapping, 52

modeling abstractions for Aggregates, 93–95

moving concepts to other Bounded Contexts, 51

Anemic Domain Model, avoiding in Aggregates, 88–89, 92

- Anticorruption Layer
 Context Mapping, 56–57
 integrating with Big Ball of Mud via, 60
 in Open Host Service, 57
 RPC with SOAP using, 62
- Application Services
 Bounded Contexts architecture, 42
 modeling Aggregates, 89
- Architecture, Bounded Contexts, 41–43
- Arrowheads, in Event Storming, 123
- Asynchronous messaging, 65–70
- At-Least-Once Delivery, messaging pattern, 68–69
- Atomic database transactions, 78–79
- B**
- Bad design, in software development, 3–7
- Behavior-Driven Development (BDD),
 Ubiquitous Language, 39
- Big Ball of Mud
 case study, 21–24
 Context Mapping and, 59–60
 turning new software into, 17
 using business experts to avoid, 18–20
 using Subdomains for legacy systems, 48–49
- Big-picture Event Storming, 114
- Black marker pens, for Event Storming, 115, 122–123
- Book Design: A Practical Introduction* (Martin), 5–6
- Boundaries, Aggregate
 design steps for right-sizing, 95–97
 protecting business invariants within, 82
 transactional consistency and, 78–81
- Bounded Contexts
 aligning with single Subdomain, 49–50
 architectural components of, 41–43
 case study, 21–24
 drawing boundaries in Event Storming, 122–124
- Context Mapping between. *See* Context Mapping
 as fundamental strategic design tool, 25–29
 modeling business policies into separate, 20
 showing flow on modeling surface in Event Storming, 122–123
 in strategic design, 7–8
 Subdomains in. *See* Subdomains in tactical design, 9
 teams and source code repositories for, 14
 understanding, 11–14
- Brandolini, Alberto, 112–113
- Business
 Aggregate boundaries protecting invariants of, 82, 95–96
 Domain Expert focus on, 17–20, 27–29
 Event Storming focus on, 113
 Event Storming process via Domain Events, 116–118
 eventual consistency driven by, 97
 focus on complexity of, 29
 leaking logic when modeling
 Aggregates, 89
 software design vs. purposes of, 4–5
 Subdomains within domains of, 46
 unit testing vs. validating specifications for, 97–98
- C**
- Caching, Aggregate performance, 109
- Causal consistency, Domain Events for, 99–100
- Challenge, 29
- Claims, 19–20, 70–73
- Classes, 90–94
- Collaboration Context
 challenging/unifying mental models, 33
 and Context Mapping, 52
- Command Message, 67
- Command Query Responsibility Segregation (CQRS), 43, 109

- Commands, Event Storming
associate Entity/Aggregate to, 120–122
causing Domain Events, 118–120
Domain Events vs., 107
identifying tasks/estimating effort, 129–131
using Domain Experts to refine, 134–136
Complex behavior, modeling
Aggregates, 93
Complexity, Domain-Driven Design
reducing, 2–3
Conformist
Context Mapping, 56
Domain Event consumers and, 67
in Open Host Service, 57
RESTful HTTP mistakes and, 64
Context Mapping
defined, 52
example in, 70–73
making good use of, 60–61
overview of, 51–53
in problem space, 12
strategic design with, 8
summary, 73
using messaging, 65–70
using RESTful HTTP, 63–65
using RPC with SOAP, 61–63
Context Mapping, types of
Anticorruption Layer, 56–57
Big Ball of Mud, 59–60
Conformist, 56
Customer-Supplier, 55–56
Open Host Service, 57
Partnership, 54
Published Language, 58
Separate Ways, 58
Shared Kernel, 55
Core concepts
Bounded Contexts for, 25–26
case study, 21–24
Core Domain
challenging/unifying mental models
to create, 29–34
and Context Mapping, 52
dealing with complexity, 47–50
defined, 12
developing Ubiquitous Language, 34–41
Event Sourcing saving record of occurrences in, 109
Event Storming to understand, 113–114
solution space implementing, 12
as type of Subdomain within project, 46–47
Ubiquitous Language maintenance vs., 41
understanding, 13
Cost
Event Storming advantages, 113
false economy of no design, 5
software design vs. affordable, 4–5
Cloud computing, using with DDD, 43
Coupled services, software design vs. strongly, 5
CQRS (Command Query Responsibility Segregation), 43, 109
Customer-Supplier Context Mapping, 55–56
- ## D
- Database
atomic transactions, 78–79
software design and, 4–5
DDD (Domain-Driven Design)
complexity of, 2–3
good, bad and effective design, 3–7
learning process and refining knowledge, 9–10
managing. *See* Managing DDD on Agile Project
overview of, 1–2
strategic design, 7–8
tactical design, 8–9
DELETE operation, RESTful HTTP, 63–65
Design-level modeling, Event Storming, 114
Diagrams, 36
Domain Events
Context Mapping example, 70–73
creating interface, 101

- Domain Events (*continued*)**
- enriching with additional data, 104
 - Event Sourcing and, 107–109
 - going asynchronous with REST, 65
 - in messaging, 65–70
 - naming types of, 101–102
 - properties, 103–104
 - scenario using, 104–107
 - summary, 109–110
 - in tactical design, 9, 99–100
- Domain Events, Event Storming**
- associate Entity/Aggregate to Command, 120–122
 - create Commands causing, 118–120
 - creating for business process, 116–118
 - identifying tasks/estimating effort, 129–131
 - identifying views/roles for users, 123–124
 - showing flow on modeling surface, 122–123
 - using Domain Experts to refine, 134–136
- Domain Experts**
- business drivers and, 17–20
 - developing Ubiquitous Language as scenarios, 35–41
 - focus on business complexity, 28
 - identifying core concepts, 26–29
 - implementing DDD on Agile Project, 133–134
 - interacting with, 134–136
 - modeling abstractions for Aggregates, 93–95
 - for one or more Subdomains, 46
 - in rapid design. *See* Event Storming
 - right-sizing Aggregates, 95–96
 - Scrum, 27
 - in strategic design, 7–8
- E**
- Effective design, 6–7
- Effort, estimating for Agile Project, 129–131
- Enrichment, Domain Event, 71–72
- Entities**
- Aggregates composed of, 77
 - associating with Commands, 120–122
 - defined, 76
 - implementing Aggregate design, 90–91
 - right-sizing Aggregates, 95
 - Value Objects describing/quantifying, 77
- Estimates**
- managing tasks in Agile Project, 129–131
 - timeboxed modeling of tasks via, 132–134
- Event-driven architecture, with DDD, 42, 112–113**
- Event Sourcing**
- in atomic database transactions, 78–79
 - overlap between Event Storming and, 121–122
 - persisting Domain Events for Aggregates, 107–109
- Event Storming**
- advantages of, 113–114
 - associate Entity/Aggregate to Command, 120–122
 - Commands causing Domain Events, 118–120
 - concrete scenarios, 35
 - Domain Events for business process, 116–118
 - Domain Experts for, 134
 - event-driven modeling vs., 112–113
 - identify tasks/estimate effort, 129–131
 - identify views/roles for users, 123–124
 - implement DDD on Agile Project, 133–134
 - modeling spikes on DDD projects via, 129
 - other tools used with, 124
 - show flow on modeling surface, 122–123
 - supplies needed for, 115–116

Events, in Event Storming, 113, 115
 Eventual consistency
 right-sizing Aggregates, 97
 updating Aggregates, 85–88
 working with, 88
 working with scenarios, 38

F

Functional programming, modeling
 Aggregates, 89

G

Generic Subdomain, 47
 GET operation
 Context Mapping example, 72
 integration using RESTful HTTP, 63–65
 Globally unique identity, Aggregate design, 90–91
 Good design, software development, 3–7

I

IDDD Workshop, 3
 Idempotent Receiver, messaging, 68
 Impact Mapping, 124
Implementing Domain-Driven Design (IDDD), Vaughn, 1, 3
 Input Adapters, Bounded Contexts architecture, 42
 Inspections policy, 19–20
 Iterations
 accuracy of long-term estimates for, 131
 identifying tasks/estimating effort, 130
 implementing DDD on Agile Project, 134
 incurring modeling debt during, 128–129
 as sprints, 126

K

Knowledge, 9–10
 Knowledge acquisition, 4–5, 6

L

Language
 evolution of terminology in human, 15
 Ubiquitous. *See* Ubiquitous Language
 Latency
 in message exchange, 65
 RESTful HTTP failures due to, 64
 Learning process, refining knowledge in, 9–10
 Legacy systems, using Subdomains with, 47–50

M

Maintenance phase, Ubiquitous Language, 40–41
 Managing DDD on Agile Project
 accuracy and, 130–131
 Event Storming, 112–124
 hiring good people, 126
 how to implement, 133–134
 identifying tasks/estimating effort, 129–131
 interacting with Domain Experts, 134–136
 modeling spikes/debt, 128–129
 other tools, 124
 overview of, 125–126
 summary, 136
 timeboxed modeling, 132–134
 using SWOT analysis, 127–128
 Martin, Douglas, 5–6
 Memory footprint, designing small Aggregates, 83
 Messaging, 65–70
 Metrics-based approach, identify tasks/estimate effort, 129–131
 Microservices, using with DDD, 43
 Modeling
 debt and spikes on DDD projects, 128–129
 development of roads and, 6
 overview of, 1
 Modules, segregating Subdomains into, 50

N

Naming

- of Aggregates, 91–92
- of Domain Event types, 101–102
- Domain Experts refining Aggregate, 134–136
- Root Entity of Aggregate, 78
- Network providers, RESTful HTTP
 - failures due to, 64
- No Design, false economy of, 5
- No Estimates approach, 125
- Nouns, in Ubiquitous Language, 34–36

O

- Object-oriented programming,
 - Aggregates, 88–89, 91–92
- Open Host Service
 - consuming Published Language, 58
 - Context Mapping, 57
 - RESTful HTTP using, 63
 - RPC with SOAP using, 62
- Opportunities, identifying Agile Project, 127–128
- Output Adapters, Bounded Contexts
 - architecture, 42

P

- Paper, conducting Event Storming on, 115–116
- Parallel processing, Event Storming of business process, 117
- Partnership Context Mapping, 54
- Performance, caching and snapshots for, 109
- Persistence operations, software design
 - vs., 5
- Persistence store, Aggregates by identity
 - for, 85
- Pictures, in domain model development, 36
- Policies
 - business group, 18–20
 - Context Mapping example of, 70–73
 - segregating into Bounded Contexts, 20
- Ports, using with DDD, 41–42
- POST operation, RESTful HTTP, 63–65

Problem space

- Bounded Contexts in, 12
- Event Storming advantages for, 114
- overview of, 12
- using Subdomains for discussing, 47
- Process, Event Storming of business, 117
- Product owner, Scrum, 27, 119
- Properties, Domain Event, 103–104
- Published Language
 - in Context Mapping, 58
 - integrating bounded contexts via, 67
 - RESTful HTTP using, 63
 - RPC with SOAP using, 62
- PUT operation, RESTful HTTP, 63–65

Q

- Query-back trade-offs, Domain Events, 71–72

R

- Rapid design. *See* Event Storming
- Reactive Model, using with DDD, 43
- Reference by identity only, Aggregates, 84–85
- References, used in this book, 137–138
- Remote Procedure Calls (RPC) with
 - SOAP, 61–63
- Representational State Transfer (REST), 43, 65
- Request-Response communications, messaging, 69–70
- REST in Practice (RIP), 63–65
- REST (Representational State Transfer), 43, 65
- RESTful HTTP, 63–65, 72
- Roads, modeling of, 6
- Robustness, RPC lacking, 62
- Roles, identifying for users in Event Storming, 123–124
- Root Entity, Aggregate
 - defined, 78
 - implementing Aggregate design, 90–91
 - right-sizing, 95
- RPC (Remote Procedure Calls) with
 - SOAP, 61–63

S

Scenarios

- developing Ubiquitous Language as, 35–38
- implementing DDD on Agile Project, 133–134
- include Domain Experts in, 134–136
- putting to work, 38–40

Scrum

- criticisms of, 125–126
- DDD Domain Expert vs. product owner in, 27
- good, bad and effective design in, 3–7
- managing project. *See* Managing DDD on Agile Project

Semantic contextual boundaries,

Bounded Contexts, 12

- Separate Ways Context Mapping, 58
- Service-Oriented Architecture (SOA), 43
- Services, Open Host Service, 57
- Shared Kernel Context Mapping, 55
- Simple Object Access Protocol (SOAP),
 - using RPC with, 61–63

Single Responsibility Principle (SRP),
Aggregates, 84Size. *See* Boundaries, AggregateSnapshots, of Aggregate performance,
109

SOA (Service-Oriented Architecture), 43

SOAP (Simple Object Access Protocol),
using RPC with, 61–63

Software developers

- developing Ubiquitous Language as scenarios, 35–41
- Domain Experts vs., 26–29
- finding good, 126
- rapid design for. *See* Event Storming

Solution space

Bounded Contexts used in, 12

overview of, 12

segregating Subdomain in, 50

Source code repositories, for Bounded
Contexts, 14*Specification* (Adzic), 124Specification by Example, refining
Ubiquitous Language, 39

Sprints

- accuracy of long-term estimates for, 131
- identifying tasks/estimating effort, 130
- incurring modeling debt during, 128–129

SRP (Single Responsibility Principle),
Aggregates, 84

Stakeholders, software design vs., 4–5

Sticky notes, Event Storming

- associate Entity/Aggregate to Command, 121–122

create Commands causing Domain
Events, 118–120

defined, 113

Domain Events for business process,
116–117

identifying roles for users, 124

overview of, 115–116

showing flow on modeling surface,
123Storage, referencing Aggregates by
identity for, 85

Strategic design

architectural components, 41–43

Bounded Contexts in, 11–17

case study, 21–24

challenging assumptions/unifying
mental models, 29–34with Context Mapping. *See* Context
MappingDomain Experts and business drivers
in, 17–20

focus on business complexity, 28

fundamental need for, 25–29

with Subdomains. *See* Subdomains

summary, 43

Ubiquitous Language in, 11–17,
34–41

understanding, 7–8

Strengths, identifying Agile Project,
127–128

Subdomains

for complexity, 47–50

overview of, 45–46

- Subdomains (*continued*)
 showing flow in Event Storming, 122–123
 strategic design with, 7–8
 summary, 50
 types of, 46–47
 understanding, 46
- Supplies, for Event Storming, 115–116
- Supporting Domain, 47, 50
- SWOT (Strengths, Weaknesses, Opportunities, and Threats) analysis, Agile Projects, 127–128
- T**
- Tactical design
 with Aggregates. *See* Aggregates
 with Domain Events. *See* Domain Events
 understanding, 8–9
- Taskboard shuffle
 project estimates/use of, 5
 software design vs., 3–4
 tendency to design using, 126
- Tasks
 identifying/estimating in Agile Project, 129–131
 timeboxed modeling of, 132–133
- Teams
 assigning to each Bounded Context, 14
 Conformist relationship between, 56
 Context Mapping integrating, 53
 Customer-Supplier relationship between, 55–56
 Event Storming advantages for, 113–114
 Partnership relationship between, 54
 Shared Kernel relationship between, 55
 Ubiquitous Language spoken within, 13–14
- Technology
 Context Mapping integrating, 53
 keeping domain model free of, 42
 modeling Aggregates, 90
 software design vs., 4–5
- Testing
 as benefit of Bounded Contexts, 25
 designing Aggregates for, 97–98
 implementing DDD on Agile Project, 134
 using Domain Experts in, 136
 validating domain model, 39–40
- Threats, identifying Agile Project, 127–128
- Timeline control, Scrum for, 4–5
- Transactional consistency boundary, Aggregates, 78–81
- Transactions, Aggregates, 78–81, 83–84
- U**
- Ubiquitous Language
 advantages of Event Storming, 113
 in Anticorruption Layer Context
 Mapping relationship, 56–57
 challenging/unifying mental models to create, 29–34
 in Conformist Context Mapping
 relationship, 56
 for Core Domain, 46–47
 developing, 34–41
 developing by collaborative feedback loop, 29
 as fundamental strategic design tool, 25–29
 maintenance phase of, 40–41
 modeling abstractions for Aggregates, 93–95
 modeling Aggregates, 93
 naming Domain Event types, 101–102
 in strategic design, 7
 understanding, 11, 13–14
 using Domain Experts to refine, 134–136
- Ubiquitous Languages
 integrating different, 53
- Separate Ways Context Mapping and, 58
 translating with Published Languages, 58
- UML (Unified Modeling Language), 90, 112–113

Unbounded legacy systems, complexity of, 48
Underwriting, Context Mapping example, 70–73
Underwriting policy, 19–20
Unit testing, 40, 97–98
Updates, Aggregate, 85–88, 96–97
User interface, abstractions for Aggregates, 94
User role, Event Storming, 119
User Story Mapping, Event Storming, 124
User Story Mapping (Patton), 124

V

Value Objects, and Aggregates, 77, 91
Views, for users in Event Storming, 123–124

W

Wall, conducting Event Storming on, 115
Weaknesses, identifying Agile Project, 127–128
Whack-a-mole issues, Big Ball of Mud, 59
Who? in Ubiquitous Language development, 36–38
Work in progress (WIP) accuracy of long-term estimates for, 131
identifying tasks/estimating effort, 130
incurring modeling debt during, 128–129
sprints as, 126