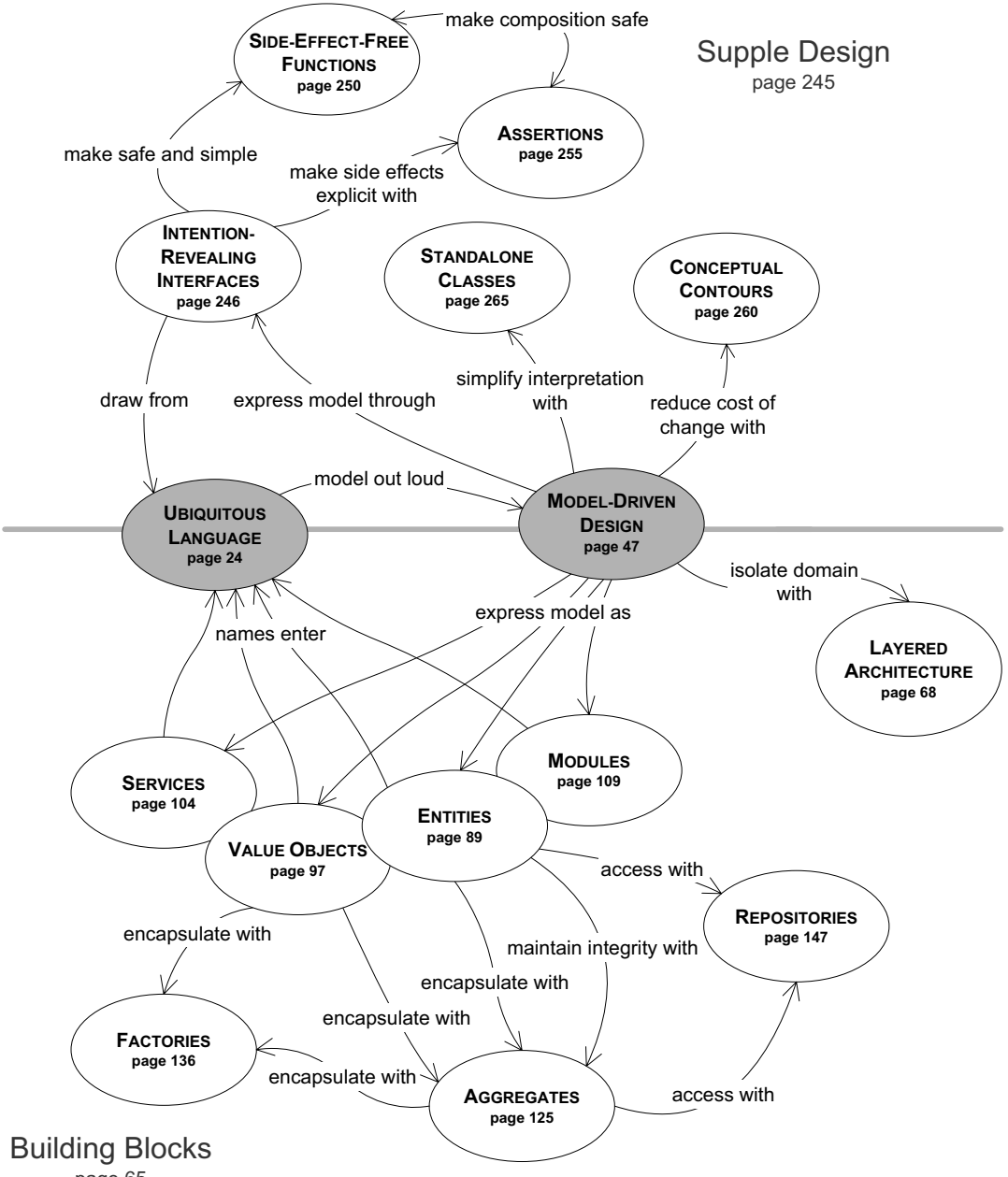
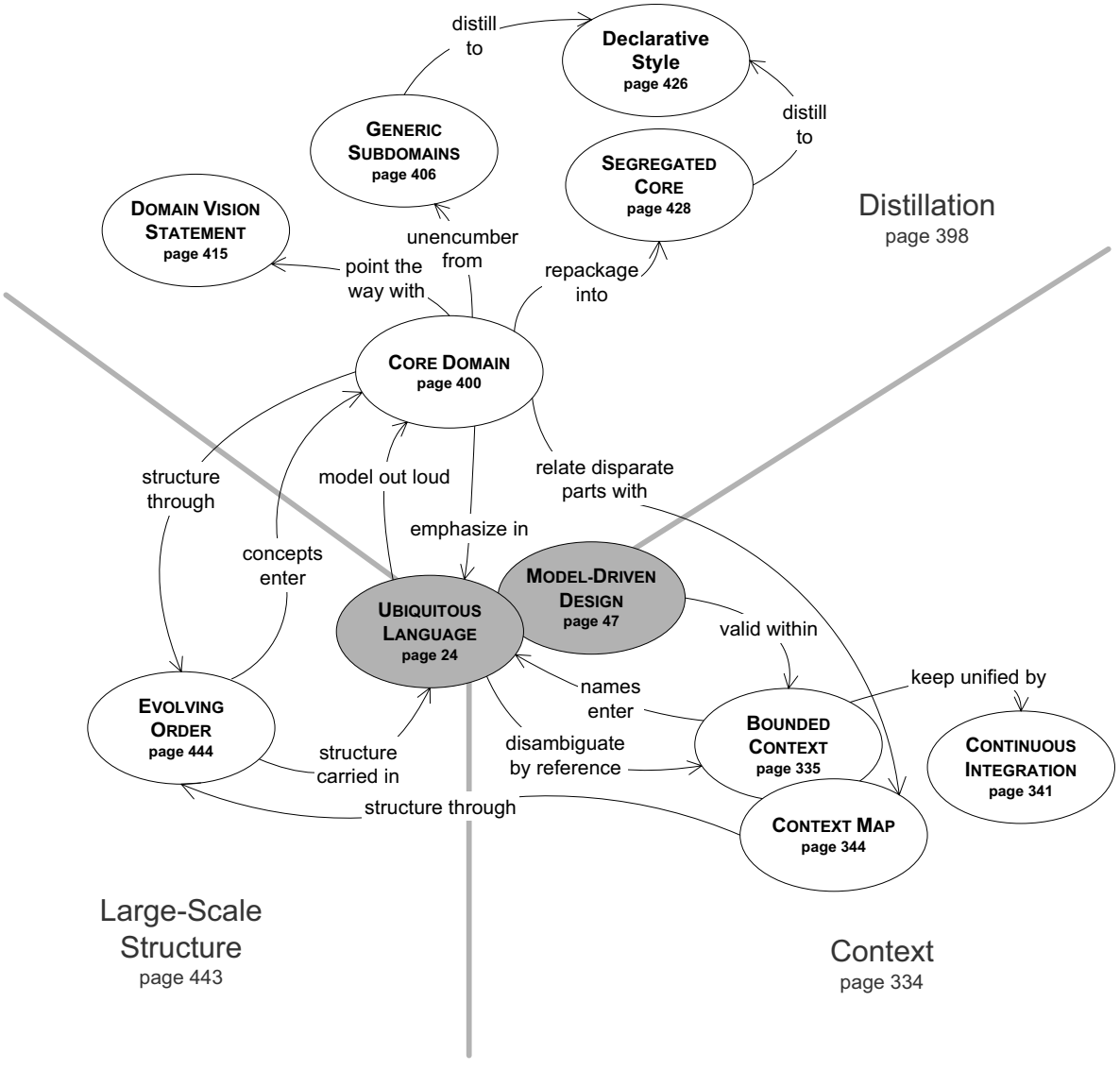
Domain-Driven DESIGN

Tackling Complexity in the Heart of Software

Eric Evans

Foreword by Martin Fowler


SIDE-EFFECT-FREE FUNCTIONS
page 250
make composition safe
Supple Design
page 245
ASSERTIONS
page 255
make safe and simple
make side effects explicit with
INTENTION-REVEALING INTERFACES
page 246
STANDALONE CLASSES
page 265
CONCEPTUAL CONTOURS
page 260
draw from
express model through
simplify interpretation with
reduce cost of change with
model out loud
UBIQUITOUS LANGUAGE
page 24
MODEL-DRIVEN DESIGN
page 47
isolate domain with
LAYERED ARCHITECTURE
page 68
express model as
names enter
SERVICES
page 104
MODULES
page 109
ENTITIES
page 89
VALUE OBJECTS
page 97
access with
REPOSITORIES
page 147
encapsulate with
maintain integrity with
encapsulate with
encapsulate with
FACTORIES
page 136
encapsulate with
AGGREGATES
page 125
access with
Building Blocks
page 65


distill to
Declarative Style
page 426
distill to
Generic Subdomains
page 406
Segregated Core
page 428
Domain Vision Statement
page 415
unencumber from
point the way with
repackage into
Distillation
page 398
Core Domain
page 400
structure through
model out loud
relate disparate parts with
emphasize in
concepts enter
Model-Driven Design
page 47
Ubiquitous Language
page 24
valid within
keep unified by
names enter
Evolving Order
page 444
Bounded Context
page 335
Continuous Integration
page 341
structure carried in
disambiguate by reference
structure through
Context Map
page 344
Large-Scale Structure
page 443
Context
page 334

Praise for *Domain-Driven Design*

"This book belongs on the shelf of every thoughtful software developer."

—Kent Beck

"Eric Evans has written a fantastic book on how you can make the design of your software match your mental model of the problem domain you are addressing.

"His book is very compatible with XP. It is not about drawing pictures of a domain; it is about how you think of it, the language you use to talk about it, and how you organize your software to reflect your improving understanding of it. Eric thinks that learning about your problem domain is as likely to happen at the end of your project as at the beginning, and so refactoring is a big part of his technique.

"The book is a fun read. Eric has lots of interesting stories, and he has a way with words. I see this book as essential reading for software developers—it is a future classic."

—Ralph Johnson, author of *Design Patterns*

"If you don't think you are getting value from your investment in object-oriented programming, this book will tell you what you've forgotten to do."

—Ward Cunningham

"What Eric has managed to capture is a part of the design process that experienced object designers have always used, but that we have been singularly unsuccessful as a group in conveying to the rest of the industry. We've given away bits and pieces of this knowledge . . . but we've never organized and systematized the principles of building domain logic. This book is important."

—Kyle Brown, author of *Enterprise Java Programming with IBM WebSphere*

"Eric Evans convincingly argues for the importance of domain modeling as the central focus of development and provides a solid framework and set of techniques for accomplishing it. This is timeless wisdom, and will hold up long after the methodologies *du jour* have gone out of fashion."

—Dave Collins, author of *Designing Object-Oriented User Interfaces*

"Eric weaves real-world experience modeling—and building—business applications into a practical, useful book. Written from the perspective of a trusted practitioner, Eric's descriptions of ubiquitous language, the benefits of sharing models with users, object life-cycle management, logical and physical application structuring, and the process and results of deep refactoring are major contributions to our field."

—Luke Hohmann, author of *Beyond Software Architecture*

Domain-Driven Design

This page intentionally left blank

Domain-Driven Design

TACKLING COMPLEXITY IN THE HEART OF SOFTWARE

Eric Evans

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

See page 517 for photo credits.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:

International Sales
international@pearsoned.com

Visit Addison-Wesley on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data
Evans, Eric, 1962–
Domain-driven design : tackling complexity in the heart of software / Eric Evans.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-12521-5
1. Computer software—Development. 2. Object-oriented programming (Computer science) I. Title.

QA76.76.D47E82 2003
005.1—dc21

2003050331

ISBN 0-321-12521-5
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
Twenty-First printing, July 2015

To Mom and Dad

This page intentionally left blank

CONTENTS

This page intentionally left blank

FOREWORD

There are many things that make software development complex. But the heart of this complexity is the essential intricacy of the problem domain itself. If you're trying to add automation to complicated human enterprise, then your software cannot dodge this complexity—all it can do is control it.

The key to controlling complexity is a good domain model, a model that goes beyond a surface vision of a domain by introducing an underlying structure, which gives the software developers the leverage they need. A good domain model can be incredibly valuable, but it's not something that's easy to make. Few people can do it well, and it's very hard to teach.

Eric Evans is one of those few who can create domain models well. I discovered this by working with him—one of those wonderful times when you find a client who's more skilled than you are. Our collaboration was short but enormous fun. Since then we've stayed in touch, and I've watched this book gestate slowly.

It's been well worth the wait.

This book has evolved into one that satisfies a huge ambition: To describe and build a vocabulary about the very art of domain modeling. To provide a frame of reference through which we can explain this activity as well as teach this hard-to-learn skill. It's a book that's given me many new ideas as it has taken shape, and I'd be astonished if even old hands at conceptual modeling don't get a raft of new ideas from reading this book.

Eric also cements many of the things that we've learned over the years. First, in domain modeling, you shouldn't separate the concepts from the implementation. An effective domain modeler can not only use a whiteboard with an accountant, but also write Java with a programmer. Partly this is true because you cannot build a

useful conceptual model without considering implementation issues. But the primary reason why concepts and implementation belong together is this: The greatest value of a domain model is that it provides a *ubiquitous language* that ties domain experts and technologists together.

Another lesson you'll learn from this book is that domain models aren't first modeled and then implemented. Like many people, I've come to reject the phased thinking of "design, then build." But the lesson of Eric's experience is that the really powerful domain models evolve over time, and even the most experienced modelers find that they gain their best ideas after the initial releases of a system.

I think, and hope, that this will be an enormously influential book. One that will add structure and cohesion to a very slippery field while it teaches a lot of people how to use a valuable tool. Domain models can have big consequences in controlling software development—in whatever language or environment they are implemented.

One final yet important thought. One of things I most respect about this book is that Eric is not afraid to talk about the times when he *hasn't* been successful. Most authors like to maintain an air of disinterested omnipotence. Eric makes it clear that like most of us, he's tasted both success and disappointment. The important thing is that he can learn from both—and more important for us is that he can pass on his lessons.

Martin Fowler
April 2003

PREFACE

Leading software designers have recognized domain modeling and design as critical topics for at least 20 years, yet surprisingly little has been written about what needs to be done or how to do it. Although it has never been formulated clearly, a philosophy has emerged as an undercurrent in the object community, a philosophy I call *domain-driven design*.

I have spent the past decade developing complex systems in several business and technical domains. In my work, I have tried best practices in design and development process as they have emerged from the leaders in object-oriented development. Some of my projects were very successful; a few failed. A feature common to the successes was a rich domain model that evolved through iterations of design and became part of the fabric of the project.

This book provides a framework for making design decisions and a technical vocabulary for discussing domain design. It is a synthesis of widely accepted best practices along with my own insights and experiences. Software development teams facing complex domains can use this framework to approach domain-driven design systematically.

Contrasting Three Projects

Three projects stand out in my memory as vivid examples of how dramatically domain design practice can affect development results. Although all three projects delivered useful software, only one achieved its ambitious objectives and produced complex software that continued to evolve to meet the ongoing needs of the organization.

I watched one project get out of the gate fast, by delivering a useful, simple Web-based trading system. Developers were flying by the

seat of their pants, but this didn't hinder them because simple software can be written with little attention to design. As a result of this initial success, expectations for future development were sky-high. That is when I was asked to work on the second version. When I took a close look, I saw that they lacked a domain model, or even a common language on the project, and were saddled with an unstructured design. The project leaders did not agree with my assessment, and I declined the job. A year later, the team found itself bogged down and unable to deliver a second version. Although their use of technology was not exemplary, it was the business logic that overcame them. Their first release had ossified prematurely into a high-maintenance legacy.

Lifting this ceiling on complexity calls for a more serious approach to the design of domain logic. Early in my career, I was fortunate to end up on a project that did emphasize domain design. This project, in a domain at least as complex as the first one, also started with a modest initial success, delivering a simple application for institutional traders. But in this case, the initial delivery was followed up with successive accelerations of development. Each iteration opened exciting new options for integrating and elaborating the functionality of the previous release. The team was able to respond to the needs of the traders with flexibility and expanding capability. This upward trajectory was directly attributable to an incisive domain model, repeatedly refined and expressed in code. As the team gained new insight into the domain, the model deepened. The quality of communication improved not only among developers but also between developers and domain experts, and the design—far from imposing an ever-heavier maintenance burden—became easier to modify and extend.

Unfortunately, projects don't arrive at such a virtuous cycle just by taking models seriously. One project from my past started with lofty aspirations to build a global enterprise system based on a domain model, but after years of disappointment, it lowered its sights and settled into conventionality. The team had good tools and a good understanding of the business, and it gave careful attention to modeling. But a poorly chosen separation of developer roles disconnected modeling from implementation, so that the design did not reflect the deep analysis that was going on. In any case, the design of detailed business objects was not rigorous enough to support combining them

in elaborate applications. Repeated iteration produced no improvement in the code, due to uneven skill levels among developers, who had no awareness of the informal body of style and technique for creating model-based objects that also function as practical, running software. As months rolled by, development work became mired in complexity and the team lost its cohesive vision of the system. After years of effort, the project did produce modest, useful software, but the team had given up its early ambitions along with the model focus.

The Challenge of Complexity

Many things can put a project off course: bureaucracy, unclear objectives, and lack of resources, to name a few. But it is the approach to design that largely determines how complex software can become. When complexity gets out of hand, developers can no longer understand the software well enough to change or extend it easily and safely. On the other hand, a good design can create opportunities to exploit those complex features.

Some design factors are technological. A great deal of effort has gone into the design of networks, databases, and other technical dimensions of software. Many books have been written about how to solve these problems. Legions of developers have cultivated their skills and followed each technical advancement.

Yet the most significant complexity of many applications is not technical. It is in the domain itself, the activity or business of the user. When this domain complexity is not handled in the design, it won't matter that the infrastructural technology is well conceived. A successful design must systematically deal with this central aspect of the software.

The premise of this book is twofold:

1. For most software projects, the primary focus should be on the domain and domain logic.
2. Complex domain designs should be based on a model.

Domain-driven design is both a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with

complicated domains. To accomplish that goal, this book presents an extensive set of design practices, techniques, and principles.

Design Versus Development Process

Design books. Process books. They seldom even reference each other. Each topic is complex in its own right. This is a design book, but I believe that design and process are inextricable. Design concepts must be implemented successfully or else they will dry up into academic discussion.

When people learn design techniques, they feel excited by the possibilities. Then the messy realities of a real project descend on them. They can't fit the new design ideas with the technology they must use. Or they don't know when to let go of a particular design aspect in the interest of time and when to dig in their heels and find a clean solution. Developers can and do talk with each other abstractly about the application of design principles, but it is more natural to talk about how real things get done. So, although this is a design book, I'm going to barge right across that artificial boundary into process when I need to. This will help put design principles in context.

This book is not tied to a particular methodology, but it is oriented toward the new family of "Agile development processes." Specifically, it assumes that a couple of practices are in place on the project. These two practices are prerequisites for applying the approach in this book.

1. *Development is iterative.* Iterative development has been advocated and practiced for decades, and it is a cornerstone of Agile development methods. There are many good discussions in the literature of Agile development and Extreme Programming (or XP), among them, *Surviving Object-Oriented Projects* (Cockburn 1998) and *Extreme Programming Explained* (Beck 1999).

2. *Developers and domain experts have a close relationship.* Domain-driven design crunches a huge amount of knowledge into a model that reflects deep insight into the domain and a focus on the key concepts. This is a collaboration between those who know the domain and those who know how to build software.

Because development is iterative, this collaboration must continue throughout the project's life.

Extreme Programming, conceived by Kent Beck, Ward Cunningham, and others (see *Extreme Programming Explained* [Beck 2000]), is the most prominent of the Agile processes and the one I have worked with most. Throughout this book, to make explanations concrete, I will use XP as the basis for discussion of the interaction of design and process. The principles illustrated are easily adapted to other Agile processes.

In recent years there has been a rebellion against elaborate development methodologies that burden projects with useless, static documents and obsessive upfront planning and design. Instead, the Agile processes, such as XP, emphasize the ability to cope with change and uncertainty.

Extreme Programming recognizes the importance of design decisions, but it strongly resists upfront design. Instead, it puts an admirable effort into communication and improving the project's ability to change course rapidly. With that ability to react, developers can use the "simplest thing that could work" at any stage of a project and then continuously refactor, making many small design improvements, ultimately arriving at a design that fits the customer's true needs.

This minimalism has been a much-needed antidote to some of the excesses of design enthusiasts. Projects have been bogged down by cumbersome documents that provided little value. They have suffered from "analysis paralysis," with team members so afraid of an imperfect design that they made no progress at all. Something had to change.

Unfortunately, some of these process ideas can be misinterpreted. Each person has a different definition of "simplest." Continuous refactoring is a series of small redesigns; developers without solid design principles will produce a code base that is hard to understand or change—the opposite of agility. And although fear of unanticipated requirements often leads to overengineering, the attempt to

avoid overengineering can develop into another fear: a fear of doing any deep design thinking at all.

In fact, XP works best for developers with a sharp design sense. The XP process assumes that you can improve a design by refactoring, and that you will do this often and rapidly. But past design choices make refactoring itself either easier or harder. The XP process attempts to increase team communication, but model and design choices clarify or confuse communication.

This book intertwines design and development practice and illustrates how domain-driven design and Agile development reinforce each other. A sophisticated approach to domain modeling within the context of an Agile development process will accelerate development. The interrelationship of process with domain development makes this approach more practical than any treatment of "pure" design in a vacuum.

The Structure of This Book

The book is divided into four major sections:

Part I: Putting the Domain Model to Work presents the basic goals of domain-driven development; these goals motivate the practices in later sections. Because there are so many approaches to software development, Part I defines terms and gives an overview of the implications of using the domain model to drive communication and design.

Part II: The Building Blocks of a Model-Driven Design condenses a core of best practices in object-oriented domain modeling into a set of basic building blocks. This section focuses on bridging the gap between models and practical, running software. Sharing these standard patterns brings order to the design. Team members more easily understand each other's work. Using standard patterns also contributes terminology to a common language, which all team members can use to discuss model and design decisions.

But the main point of this section is to focus on the kinds of decisions that keep the model and implementation aligned with each other, each reinforcing the other's effectiveness. This align-

ment requires attention to the detail of individual elements. Careful crafting at this small scale gives developers a steady foundation from which to apply the modeling approaches of Parts III and IV.

Part III: Refactoring Toward Deeper Insight goes beyond the building blocks to the challenge of assembling them into practical models that provide the payoff. Rather than jumping directly into esoteric design principles, this section emphasizes the discovery process. Valuable models do not emerge immediately; they require a deep understanding of the domain. That understanding comes from diving in, implementing an initial design based on a probably naive model, and then transforming it again and again. Each time the team gains insight, the model is transformed to reveal that richer knowledge, and the code is refactored to reflect the deeper model and make its potential available to the application. Then, once in a while, this onion peeling leads to an opportunity to break through to a much deeper model, attended by a rush of profound design changes.

Exploration is inherently open-ended, but it does not have to be random. Part III delves into modeling principles that can guide choices along the way, and techniques that help direct the search.

Part IV: Strategic Design deals with situations that arise in complex systems, larger organizations, and interactions with external systems and legacy systems. This section explores a triad of principles that apply to the system as a whole: context, distillation, and large-scale structure. Strategic design decisions are made by teams, or even among teams. Strategic design enables the goals of Part I to be realized on a larger scale, for a big system or an application that fits into a sprawling, enterprise-wide network.

Throughout the book, discussions are illustrated not with oversimplified, "toy" problems, but with realistic examples adapted from actual projects.

Much of the book is written as a set of "patterns." Readers should be able to understand the material without concern about this

device, but those who are interested in the style and format of the patterns may want to read the appendix.

Supplemental materials can be found at http://domaindrivendesign.org, including additional example code and community discussion.

Who Should Read This Book

This book is written primarily for developers of object-oriented software. Most members of a software project team can benefit from some parts of the book. It will make the most sense to people who are currently involved with a project, trying to do some of these things as they go through, and to people who already have deep experience with such projects.

Some knowledge of object-oriented modeling is necessary to benefit from this book. The examples include UML diagrams and Java code, so the ability to read those languages at a basic level is important, but it is unnecessary to have mastered the details of either. Knowledge of Extreme Programming will add perspective to the discussions of development process, but the material should be understandable to those without background knowledge.

For intermediate software developers—readers who already know something of object-oriented design and may have read one or two software design books—this book will fill in gaps and provide perspective on how object modeling fits into real life on a software project. The book will help intermediate developers learn to apply sophisticated modeling and design skills to practical problems.

Advanced or expert software developers will be interested in the book's comprehensive framework for dealing with the domain. This systematic approach to design will help technical leaders guide their teams down this path. Also, the coherent terminology used throughout the book will help advanced developers communicate with their peers.

This book is a narrative, and it can be read from beginning to end, or from the beginning of any chapter. Readers of various backgrounds may wish to take different paths through the book, but I do recommend that all readers start with the introduction to Part I, as well as

Chapter 1. Beyond that, the core is probably Chapters 2, 3, 9, and 14. A skimmer who already has some grasp of a topic should be able to pick up the main points by reading headings and bold text. A very advanced reader may want to skim Parts I and II and will probably be most interested in Parts III and IV.

In addition to this core readership, analysts and relatively technical project managers will also benefit from reading the book. Analysts can draw on the connection between model and design to make more effective contributions in the context of an Agile project. Analysts may also use some of the principles of strategic design to better focus and organize their work.

Project managers should be interested in the emphasis on making a team more effective and more focused on designing software meaningful to business experts and users. And because strategic design decisions are interrelated with team organization and work styles, these design decisions necessarily involve the leadership of the project and have a major impact on the project's trajectory.

A Domain-Driven Team

Although an individual developer who understands domain-driven design will gain valuable design techniques and perspective, the biggest gains come when a team joins together to apply a domain-driven design approach and to move the domain model to the project's center of discourse. By doing so, the team members will share a language that enriches their communication and keeps it connected to the software. They will produce a lucid implementation in step with a model, giving leverage to application development. They will share a map of how the design work of different teams relates, and they will systematically focus attention on the features that are most distinctive and valuable to the organization.

Domain-driven design is a difficult technical challenge that can pay off big, opening opportunities just when most software projects begin to ossify into legacy.

This page intentionally left blank

ACKNOWLEDGMENTS

I have been working on this book, in one form or another, for more than four years, and many people have helped and supported me along the way.

I thank those people who have read manuscripts and commented. This book would simply not have been possible without that feedback. A few have given their reviews especially generous attention. The Silicon Valley Patterns Group, led by Russ Rufer and Tracy Bialek, spent seven weeks scrutinizing the first complete draft of the book. The University of Illinois reading group led by Ralph Johnson also spent several weeks reviewing a later draft. Listening to the long, lively discussions of these groups had a profound effect. Kyle Brown and Martin Fowler contributed detailed feedback, valuable insights, and invaluable moral support (while sitting on a fish). Ward Cunningham's comments helped me shore up some important weak points. Alistair Cockburn encouraged me early on and helped me find my way through the publication process, as did Hilary Evans. David Siegel and Eugene Wallingford have helped me avoid embarrassing myself in the more technical parts. Vibhu Mohindra and Vladimir Gitlevich painstakingly checked all the code examples.

Rob Mee read some of my earliest explorations of the material, and brainstormed ideas with me when I was groping for some way to communicate this style of design. He then pored over a much later draft with me.

Josh Kerievsky is responsible for one of the major turning points in the book's development: He persuaded me to try out the "Alexandrian" pattern format, which became so central to the book's organization. He also helped me to bring together some of the material now in Part II into a coherent form for the first time, during the intensive

"shepherding" process preceding the PLoP conference in 1999. This became a seed around which much of the rest of the book formed.

Also I thank Awad Faddoul for the hundreds of hours I sat writing in his wonderful café. That retreat, along with a lot of windsurfing, helped me keep going.

And I'm very grateful to Martine Jousset, Richard Paselk, and Ross Venables for creating some beautiful photographs to illustrate a few key concepts (see photo credits on page 517).

Before I could have conceived of this book, I had to form my view and understanding of software development. That formation owed a lot to the generosity of a few brilliant people who acted as informal mentors to me, as well as friends. David Siegel, Eric Gold, and Iseult White, each in a different way, helped me develop my way of thinking about software design. Meanwhile, Bruce Gordon, Richard Freyberg, and Dr. Judith Segal, also in very different ways, helped me find my way in the world of successful project work.

My own notions naturally grew out of a body of ideas in the air at that time. Some of those contributions will be clear in the main text and referenced where possible. Others are so fundamental that I don't even realize their influence on me.

My master's thesis advisor, Dr. Bala Subramanium, turned me on to mathematical modeling, which we applied to chemical reaction kinetics. Modeling is modeling, and that work was part of the path that led to this book.

Even before that, my way of thinking was shaped by my parents, Carol and Gary Evans. And a few special teachers awakened my interest or helped me lay foundations, especially Dale Currier (a high school math teacher), Mary Brown (a high school English composition teacher), and Josephine McGlamery (a sixth-grade science teacher).

Finally, I thank my friends and family, and Fernando De Leon, for their encouragement all along the way.

I

Putting the Domain Model to Work

This eighteenth-century Chinese map represents the whole world. In the center and taking up most of the space is China, surrounded by perfunctory representations of other countries. This was a model of the world appropriate to that society, which had intentionally turned inward. The worldview that the map represents must not have been helpful in dealing with foreigners. Certainly it would not serve modern China at all. Maps are models, and every model represents some aspect of reality or an idea that is of interest. A model is a simplification. It is an interpretation of reality that abstracts the aspects relevant to solving the problem at hand and ignores extraneous detail.

Every software program relates to some activity or interest of its user. That subject area to which the user applies the program is the *domain* of the software. Some domains involve the physical world: The domain of an airline-booking program involves real people getting on real aircraft. Some domains are intangible: The domain of an accounting program is money and finance. Software domains usually have little to do with computers, though there are exceptions: The domain of a source-code control system is software development itself.

To create software that is valuably involved in users' activities, a development team must bring to bear a body of knowledge related to those activities. The breadth of knowledge required can be daunting. The volume and complexity of information can be overwhelming. Models are tools for grappling with this overload. A model is a selectively simplified and consciously structured form of knowledge. An appropriate model makes sense of information and focuses it on a problem.

A domain model is not a particular diagram; it is the idea that the diagram is intended to convey. It is not just the knowledge in a domain expert's head; *it is a rigorously organized and selective abstraction of that knowledge*. A diagram can represent and communicate a model, as can carefully written code, as can an English sentence.

Domain modeling is not a matter of making as "realistic" a model as possible. Even in a domain of tangible real-world things, our model is an artificial creation. Nor is it just the construction of a software mechanism that gives the necessary results. It is more like moviemaking, loosely representing reality to a particular purpose. Even a documentary film does not show unedited real life. Just as a moviemaker selects aspects of experience and presents them in an idiosyncratic way to tell a story or make a point, a domain modeler chooses a particular model for its utility.

The Utility of a Model in Domain-Driven Design

In domain-driven design, three basic uses determine the choice of a model.

1. *The model and the heart of the design shape each other.* It is the intimate link between the model and the implementation that makes the model relevant and ensures that the analysis that went into it applies to the final product, a running program. This binding of model and implementation also helps during maintenance and continuing development, because the code can be interpreted based on understanding the model. (See Chapter 3.)

2. *The model is the backbone of a language used by all team members.* Because of the binding of model and implementation, developers can talk about the program in this language. They can communicate with domain experts without translation. And because the language is based on the model, our natural linguistic abilities can be turned to refining the model itself. (See Chapter 2.)

3. *The model is distilled knowledge.* The model is the team's agreed-upon way of structuring domain knowledge and distinguishing the elements of most interest. A model captures how we choose to think about the domain as we select terms, break down concepts, and relate them. The shared language allows developers and domain experts to collaborate effectively as they wrestle information into this form. The binding of model and implementation makes experience with early versions of the software applicable as feedback into the modeling process. (See Chapter 1.)

The next three chapters set out to examine the meaning and value of each of these contributions in turn, and the ways they are intertwined. Using a model in these ways can support the development of software with rich functionality that would otherwise take a massive investment of ad hoc development.

The Heart of Software

The heart of software is its ability to solve domain-related problems for its user. All other features, vital though they may be, support this basic purpose. When the domain is complex, this is a difficult task, calling for the concentrated effort of talented and skilled people. Developers have to steep themselves in the domain to build up knowledge of the business. They must hone their modeling skills and master domain design.

Yet these are not the priorities on most software projects. Most talented developers do not have much interest in learning about the specific domain in which they are working, much less making a major commitment to expand their domain-modeling skills. Technical people enjoy quantifiable problems that exercise their technical skills. Domain work is messy and demands a lot of complicated new knowledge that doesn't seem to add to a computer scientist's capabilities.

Instead, the technical talent goes to work on elaborate frameworks, trying to solve domain problems with technology. Learning about and modeling the domain is left to others. Complexity in the heart of software has to be tackled head-on. To do otherwise is to risk irrelevance.

In a TV talk show interview, comedian John Cleese told a story of an event during the filming of *Monty Python and the Holy Grail*. They had been shooting a particular scene over and over, but somehow it wasn't funny. Finally, he took a break and consulted with fellow comedian Michael Palin (the other actor in the scene), and they came up with a slight variation. They shot one more take, and it turned out funny, so they called it a day.

The next morning, Cleese was looking at the rough cut the film editor had put together of the previous day's work. Coming to the scene they had struggled with, Cleese found that it wasn't funny; one of the earlier takes had been used.

He asked the film editor why he hadn't used the last take, as directed. "Couldn't use it. Someone walked in-shot," the editor replied. Cleese watched the scene again, and then again. Still he could see nothing wrong. Finally, the editor stopped the film and pointed out a coat sleeve that was visible for a moment at the edge of the picture.

The film editor was focused on the precise execution of his own specialty. He was concerned that other film editors who saw the movie would judge his work based on its technical perfection. In the process, the heart of the scene had been lost ("The Late Late Show with Craig Kilborn," CBS, September 2001).

Fortunately, the funny scene was restored by a director who understood comedy. In just the same way, leaders within a team who understand the centrality of the domain can put their software project back on course when development of a model that reflects deep understanding gets lost in the shuffle.

This book will show that domain development holds opportunities to cultivate very sophisticated design skills. The messiness of most

software domains is actually an interesting technical challenge. In fact, in many scientific disciplines, "complexity" is one of the most exciting current topics, as researchers attempt to tackle the messiness of the real world. A software developer has that same prospect when facing a complicated domain that has never been formalized. Creating a lucid model that cuts through that complexity is exciting.

There are systematic ways of thinking that developers can employ to search for insight and produce effective models. There are design techniques that can bring order to a sprawling software application. Cultivation of these skills makes a developer much more valuable, even in an initially unfamiliar domain.

ONE

Crunching Knowledge

A few years ago, I set out to design a specialized software tool for printed-circuit board (PCB) design. One catch: I didn't know anything about electronic hardware. I had access to some PCB designers, of course, but they typically got my head spinning in three minutes. How was I going to understand enough to write this software? I certainly wasn't going to become an electrical engineer before the delivery deadline!

We tried having the PCB designers tell me exactly what the software should do. Bad idea. They were great circuit designers, but their software ideas usually involved reading in an ASCII file, sorting it, writing it back out with some annotation, and producing a report. This was clearly not going to lead to the leap forward in productivity that they were looking for.

The first few meetings were discouraging, but there was a glimmer of hope in the reports they asked for. They always involved "nets" and various details about them. A net, in this domain, is essentially a wire conductor that can connect any number of components on a PCB and carry an electrical signal to everything it is connected to. We had the first element of the domain model.

Figure 1.1

I started drawing diagrams for them as we discussed the things they wanted the software to do. I used an informal variant of object interaction diagrams to walk through scenarios.

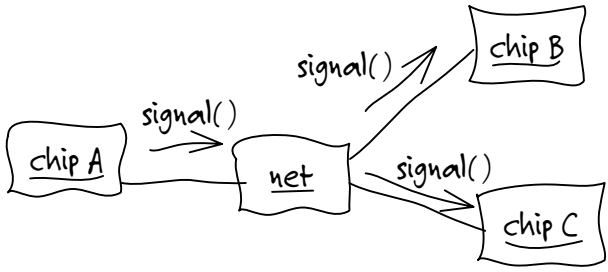


Figure 1.2

PCB Expert 1: The components wouldn't have to be chips.

Developer (Me): So I should just call them "components"?

Expert 1: We call them "component instances." There could be many of the same component.

Expert 2: The "net" box looks just like a component instance.

Expert 1: He's not using our notation. Everything is a box for them, I guess.

Developer: Sorry to say, yes. I guess I'd better explain this notation a little more.

They constantly corrected me, and as they did I started to learn. We ironed out collisions and ambiguities in their terminology and differences between their technical opinions, and they learned. They began to explain things more precisely and consistently, and we started to develop a model together.

Expert 1: It isn't enough to say a signal arrives at a ref-des, we have to know the pin.

Developer: Ref-des?

Expert 2: Same thing as a component instance. Ref-des is what it's called in a particular tool we use.

Expert 1: Anyhow, a net connects a particular pin of one instance to a particular pin of another.

Developer: Are you saying that a pin belongs to only one component instance and connects to only one net?

Expert 1: Yes, that's right.

Expert 2: Also, every net has a topology, an arrangement that determines the way the elements of the net connect.

Developer: OK, how about this?

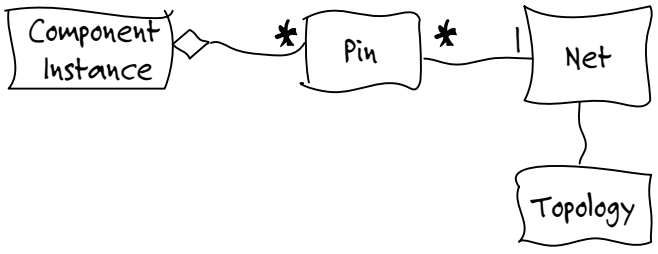


Figure 1.3

To focus our exploration, we limited ourselves, for a while, to studying one particular feature. A "probe simulation" would trace the propagation of a signal to detect likely sites of certain kinds of problems in the design.

Developer: I understand how the signal gets carried by the **Net** to all the **Pins** attached, but how does it go any further than that? Does the **Topology** have something to do with it?

Expert 2: No. The component pushes the signal through.

Developer: We certainly can't model the internal behavior of a chip. That's way too complicated.

Expert 2: We don't have to. We can use a simplification. Just a list of pushes through the component from certain **Pins** to certain others.

Developer: Something like this?

[With considerable trial-and-error, together we sketched out a scenario.]

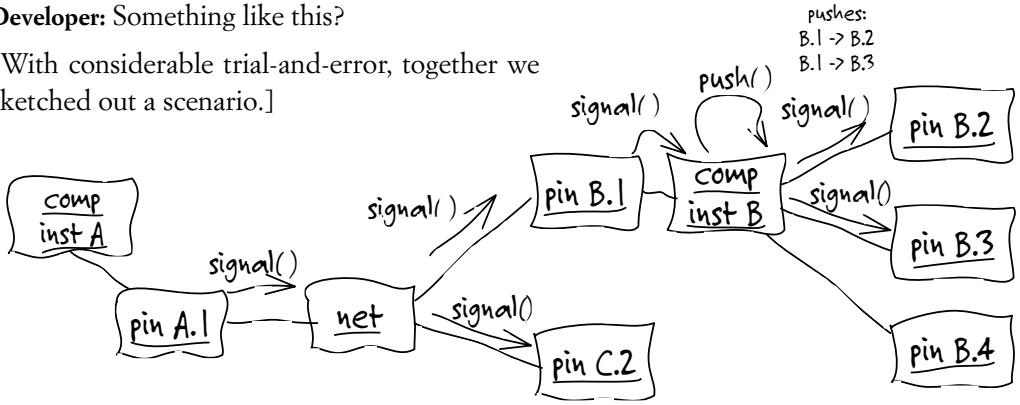


Figure 1.4

Developer: But what exactly do you need to know from this computation?

Expert 2: We'd be looking for long signal delays—say, any signal path that was more than two or three hops. It's a rule of thumb. If the path is too long, the signal may not arrive during the clock cycle.

Developer: More than three hops. . . . So we need to calculate the path lengths. And what counts as a hop?

Expert 2: Each time the signal goes over a **Net**, that's one hop.

Developer: So we could pass the number of hops along, and a **Net** could increment it, like this.

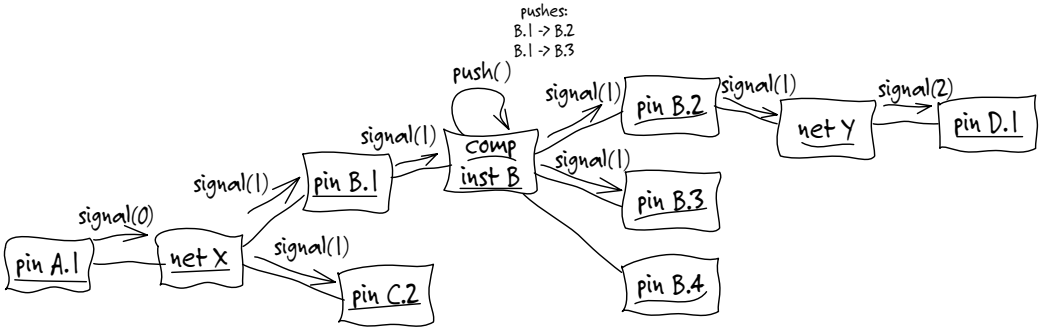


Figure 1.5

Developer: The only part that isn't clear to me is where the "pushes" come from. Do we store that data for every **Component Instance**?

Expert 2: The pushes would be the same for all the instances of a component.

Developer: So the type of component determines the pushes. They'll be the same for every instance?

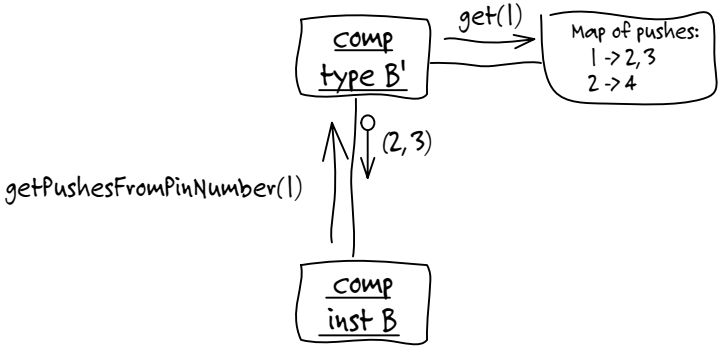


Figure 1.6

Expert 2: I'm not sure exactly what some of this means, but I would imagine storing push-throughs for each component would look something like that.

Developer: Sorry, I got a little too detailed there. I was just thinking it through. . . . So, now, where does the **Topology** come into it?

Expert 1: That's not used for the probe simulation.

Developer: Then I'm going to drop it out for now, OK? We can bring it back when we get to those features.

And so it went (with much more stumbling than is shown here). Brainstorming and refining; questioning and explaining. The model developed along with my understanding of the domain and their understanding of how the model would play into the solution. A class diagram representing that early model looks something like this.

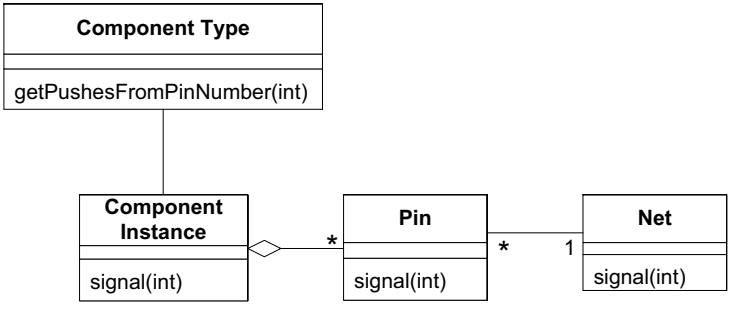


Figure 1.7

After a couple more part-time days of this, I felt I understood enough to attempt some code. I wrote a very simple prototype, driven by an automated test framework. I avoided all infrastructure. There was no persistence, and no user interface (UI). This allowed me to concentrate on the behavior. I was able to demonstrate a simple probe simulation in just a few more days. Although it used dummy data and wrote raw text to the console, it was nonetheless doing the actual computation of path lengths using Java objects. Those Java objects reflected a model shared by the domain experts and myself.

The concreteness of this prototype made clearer to the domain experts what the model meant and how it related to the functioning software. From that point, our model discussions became more interactive,

as they could see how I incorporated my newly acquired knowledge into the model and then into the software. And they had concrete feedback from the prototype to evaluate their own thoughts.

Embedded in that model, which naturally became much more complicated than the one shown here, was knowledge about the domain of PCB relevant to the problems we were solving. It consolidated many synonyms and slight variations in descriptions. It excluded hundreds of facts that the engineers understood but that were not directly relevant, such as the actual digital features of the components. A software specialist like me could look at the diagrams and in minutes start to get a grip on what the software was about. He or she would have a framework to organize new information and learn faster, to make better guesses about what was important and what was not, and to communicate better with the PCB engineers.

As the engineers described new features they needed, I made them walk me through scenarios of how the objects interacted. When the model objects couldn't carry us through an important scenario, we brainstormed new ones or changed old ones, crunching their knowledge. We refined the model; the code coevolved. A few months later the PCB engineers had a rich tool that exceeded their expectations.

Ingredients of Effective Modeling

Certain things we did led to the success I just described.

1. *Binding the model and the implementation.* That crude prototype forged the essential link early, and it was maintained through all subsequent iterations.

2. *Cultivating a language based on the model.* At first, the engineers had to explain elementary PCB issues to me, and I had to explain what a class diagram meant. But as the project proceeded, any of us could take terms straight out of the model, organize them into sentences consistent with the structure of the model, and be unambiguously understood without translation.

3. *Developing a knowledge-rich model.* The objects had behavior and enforced rules. The model wasn't just a data schema; it was

integral to solving a complex problem. It captured knowledge of various kinds.

4. *Distilling the model.* Important concepts were added to the model as it became more complete, but equally important, concepts were dropped when they didn't prove useful or central. When an unneeded concept was tied to one that was needed, a new model was found that distinguished the essential concept so that the other could be dropped.

5. *Brainstorming and experimenting.* The language, combined with sketches and a brainstorming attitude, turned our discussions into laboratories of the model, in which hundreds of experimental variations could be exercised, tried, and judged. As the team went through scenarios, the spoken expressions themselves provided a quick viability test of a proposed model, as the ear could quickly detect either the clarity and ease or the awkwardness of expression.

It is the creativity of brainstorming and massive experimentation, leveraged through a model-based language and disciplined by the feedback loop through implementation, that makes it possible to find a knowledge-rich model and distill it. This kind of *knowledge crunching* turns the knowledge of the team into valuable models.

Knowledge Crunching

Financial analysts crunch numbers. They sift through reams of detailed figures, combining and recombining them looking for the underlying meaning, searching for a simple presentation that brings out what is really important—an understanding that can be the basis of a financial decision.

Effective domain modelers are knowledge crunchers. They take a torrent of information and probe for the relevant trickle. They try one organizing idea after another, searching for the simple view that makes sense of the mass. Many models are tried and rejected or transformed. Success comes in an emerging set of abstract concepts that makes sense of all the detail. This distillation is a rigorous expression of the particular knowledge that has been found most relevant.

Knowledge crunching is not a solitary activity. A team of developers and domain experts collaborate, typically led by developers. Together they draw in information and crunch it into a useful form. The raw material comes from the minds of domain experts, from users of existing systems, from the prior experience of the technical team with a related legacy system or another project in the same domain. It comes in the form of documents written for the project or used in the business, and lots and lots of talk. Early versions or prototypes feed experience back into the team and change interpretations.

In the old waterfall method, the business experts talk to the analysts, and analysts digest and abstract and pass the result along to the programmers, who code the software. This approach fails because it completely lacks feedback. The analysts have full responsibility for creating the model, based only on input from the business experts. They have no opportunity to learn from the programmers or gain experience with early versions of software. Knowledge trickles in one direction, but does not accumulate.

Other projects use an iterative process, but they fail to build up knowledge because they don't abstract. Developers get the experts to describe a desired feature and then they go build it. They show the experts the result and ask what to do next. If the programmers practice refactoring, they can keep the software clean enough to continue extending it, but if programmers are not interested in the domain, they learn only what the application should do, not the principles behind it. Useful software can be built that way, but the project will never arrive at a point where powerful new features unfold as corollaries to older features.

Good programmers will naturally start to abstract and develop a model that can do more work. But when this happens only in a technical setting, without collaboration with domain experts, the concepts are naive. That shallowness of knowledge produces software that does a basic job but lacks a deep connection to the domain expert's way of thinking.

The interaction between team members changes as all members crunch the model together. The constant refinement of the domain model forces the developers to learn the important principles of the business they are assisting, rather than to produce functions mechanically. The domain experts often refine their own understanding by being forced to distill what they know to essentials, and they come to understand the conceptual rigor that software projects require.

All this makes the team members more competent knowledge crunchers. They winnow out the extraneous. They recast the model into an ever more useful form. Because analysts and programmers are feeding into it, it is cleanly organized and abstracted, so it can provide leverage for the implementation. Because the domain experts are feeding into it, the model reflects deep knowledge of the business. The abstractions are true business principles.

As the model improves, it becomes a tool for organizing the information that continues to flow through the project. The model focuses requirements analysis. It intimately interacts with programming and design. And in a virtuous cycle, it deepens team members' insight into the domain, letting them see more clearly and leading to further refinement of the model. These models are never perfect; they evolve. They must be practical and useful in making sense of the domain. They must be rigorous enough to make the application simple to implement and understand.

Continuous Learning

When we set out to write software, we never know enough. Knowledge on the project is fragmented, scattered among many people and documents, and it's mixed with other information so that we don't even know which bits of knowledge we really need. Domains that seem less technically daunting can be deceiving: we don't realize how much we don't know. This ignorance leads us to make false assumptions.

Meanwhile, all projects leak knowledge. People who have learned something move on. Reorganization scatters the team, and the knowledge is fragmented again. Crucial subsystems are outsourced in such a way that code is delivered but knowledge isn't. And with typical design approaches, the code and documents don't

express this hard-earned knowledge in a usable form, so when the oral tradition is interrupted for any reason, the knowledge is lost.

Highly productive teams grow their knowledge consciously, practicing *continuous learning* (Kerievsky 2003). For developers, this means improving technical knowledge, along with general domain-modeling skills (such as those in this book). But it also includes serious learning about the specific domain they are working in.

These self-educated team members form a stable core of people to focus on the development tasks that involve the most critical areas. (For more on this, see Chapter 15.) The accumulated knowledge in the minds of this core team makes them more effective knowledge crunchers.

At this point, stop and ask yourself a question. Did you learn something about the PCB design process? Although this example has been a superficial treatment of that domain, there should be some learning when a domain model is discussed. I learned an enormous amount. I did not learn how to be a PCB engineer. That was not the goal. I learned to talk to PCB experts, understand the major concepts relevant to the application, and sanity-check what we were building.

In fact, our team eventually discovered that the probe simulation was a low priority for development, and the feature was eventually dropped altogether. With it went the parts of the model that captured understanding of pushing signals through components and counting hops. The core of the application turned out to lie elsewhere, and the model changed to bring those aspects onto center stage. The domain experts had learned more and had clarified the goal of the application. (Chapter 15 discusses these issues in depth.)

Even so, the early work was essential. Key model elements were retained, but more important, that work set in motion the process of knowledge crunching that made all subsequent work effective: the knowledge gained by team members, developers, and domain experts alike; the beginnings of a shared language; and the closing of a feedback loop through implementation. A voyage of discovery has to start somewhere.

Knowledge-Rich Design

The kind of knowledge captured in a model such as the PCB example goes beyond "find the nouns." Business activities and rules are as central to a domain as are the entities involved; any domain will have various categories of concepts. Knowledge crunching yields models that reflect this kind of insight. In parallel with model changes, developers refactor the implementation to express the model, giving the application use of that knowledge.

It is with this move beyond entities and values that knowledge crunching can get intense, because there may be actual inconsistency among business rules. Domain experts are usually not aware of how complex their mental processes are as, in the course of their work, they navigate all these rules, reconcile contradictions, and fill in gaps with common sense. Software can't do this. It is through knowledge crunching in close collaboration with software experts that the rules are clarified, fleshed out, reconciled, or placed out of scope.

Example

Extracting a Hidden Concept

Let's start with a very simple domain model that could be the basis of an application for booking cargos onto a voyage of a ship.

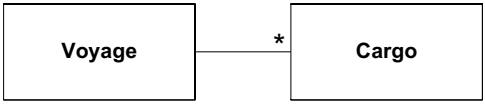


Figure 1.8

We can state that the booking application's responsibility is to associate each **Cargo** with a **Voyage**, recording and tracking that relationship. So far so good. Somewhere in the application code there could be a method like this:

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

Because there are always last-minute cancellations, standard practice in the shipping industry is to accept more cargo than a particular vessel can carry on a voyage. This is called "overbooking."

Sometimes a simple percentage of capacity is used, such as booking 110 percent of capacity. In other cases complex rules are applied, favoring major customers or certain kinds of cargo.

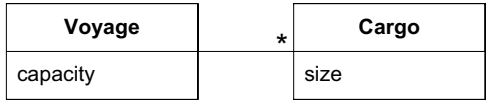
This is a basic strategy in the shipping domain that would be known to any businessperson in the shipping industry, but it might not be understood by all technical people on a software team.

The requirements document contains this line:

Allow 10% overbooking.

The class diagram and code now look like this:

Figure 1.9




```
public int makeBooking(Cargo cargo, Voyage voyage) {
   double maxBooking = voyage.capacity() * 1.1;
   if ((voyage.bookedCargoSize() + cargo.size()) > maxBooking)
       return -1;
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

Now an important business rule is hidden as a guard clause in an application method. Later, in Chapter 4, we'll look at the principle of LAYERED ARCHITECTURE, which would guide us to move the overbooking rule into a domain object, but for now let's concentrate on how we could make this knowledge more explicit and accessible to everyone on the project. This will bring us to a similar solution.

1. As written, it is unlikely that any business expert could read this code to verify the rule, even with the guidance of a developer.
2. It would be difficult for a technical, non-businessperson to connect the requirement text with the code.

If the rule were more complex, that much more would be at stake.

We can change the design to better capture this knowledge. The overbooking rule is a policy. *Policy* is another name for the design pattern known as STRATEGY (Gamma et al. 1995). It is usually moti-

vated by the need to substitute different rules, which is not needed here, as far as we know. But the concept we are trying to capture does fit the *meaning* of a policy, which is an equally important motivation in domain-driven design. (See Chapter 12, "Relating Design Patterns to the Model.")

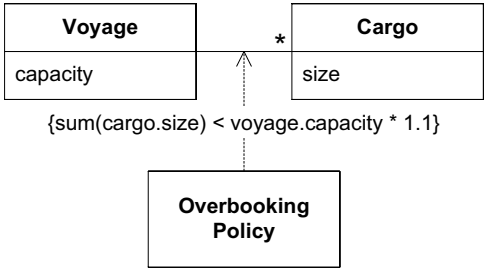


Figure 1.10

The code is now:

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   if (!overbookingPolicy.isAllowed(cargo, voyage)) return -1;
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

The new **Overbooking Policy** class contains this method:

```
public boolean isAllowed(Cargo cargo, Voyage voyage) {
   return (cargo.size() + voyage.bookedCargoSize()) <=
         (voyage.capacity() * 1.1);
}
```

It will be clear to all that overbooking is a distinct policy, and the implementation of that rule is explicit and separate.

Now, *I am not recommending that such an elaborate design be applied to every detail of the domain.* Chapter 15, "Distillation," goes into depth on how to focus on the important and minimize or separate everything else. This example is meant to show that a domain model and corresponding design can be used to secure and share knowledge. The more explicit design has these advantages:

1. In order to bring the design to this stage, the programmers and everyone else involved will have come to understand the nature

of overbooking as a distinct and important business rule, not just an obscure calculation.

2. Programmers can show business experts technical artifacts, even code, that should be intelligible to domain experts (with guidance), thereby closing the feedback loop.

Deep Models

Useful models seldom lie on the surface. As we come to understand the domain and the needs of the application, we usually discard superficial model elements that seemed important in the beginning, or we shift their perspective. Subtle abstractions emerge that would not have occurred to us at the outset but that pierce to the heart of the matter.

The preceding example is loosely based on one of the projects that I'll be drawing on for several examples throughout the book: a container shipping system. The examples in this book will be kept accessible to non-shipping experts. But on a real project, where continuous learning prepares the team members, models of utility and clarity often call for sophistication both in the domain and in modeling technique.

On that project, because a shipment begins with the act of booking cargo, we developed a model that allowed us to describe the cargo, its itinerary, and so on. This was all necessary and useful, yet the domain experts felt dissatisfied. There was a way they looked at their business that we were missing.

Eventually, after months of knowledge crunching, we realized that the handling of cargo, the physical loading and unloading, the movements from place to place, was largely carried out by subcontractors or by operational people in the company. In the view of our shipping experts, there was a series of transfers of responsibility between parties. A process governed that transfer of legal and practical responsibility, from the shipper to some local carrier, from one carrier to another, and finally to the consignee. Often, the cargo would sit in a warehouse while important steps were being taken. At other times, the cargo would move through complex physical steps that were not relevant to the shipping company's business decisions. Rather than

the logistics of the itinerary, what came to the fore were legal documents such as the bill of lading, and processes leading to the release of payments.

This deeper view of the shipping business did not lead to the removal of the Itinerary object, but the model changed profoundly. Our view of shipping changed from moving containers from place to place, to transferring responsibility for cargo from entity to entity. Features for handling these transfers of responsibility were no longer awkwardly attached to loading operations, but were supported by a model that came out of an understanding of the significant relationship between those operations and those responsibilities.

Knowledge crunching is an exploration, and you can't know where you will end up.

This page intentionally left blank

TWO

Communication and the Use of Language

A domain model can be the core of a common language for a software project. The model is a set of concepts built up in the heads of people on the project, with terms and relationships that reflect domain insight. These terms and interrelationships provide the semantics of a language that is tailored to the domain while being precise enough for technical development. This is a crucial cord that weaves the model into development activity and binds it with the code.

This model-based communication is not limited to diagrams in Unified Modeling Language (UML). To make most effective use of a model, it needs to pervade every medium of communication. It increases the utility of written text documents, as well as the informal diagrams and casual conversation reemphasized in Agile processes. It improves communication through the code itself and through the tests for that code.

The use of language on a project is subtle but all-important. . . .

Ubiquitous Language

> For first you write a sentence,
> And then you chop it small;
> Then mix the bits, and sort them out
> Just as they chance to fall:
> The order of the phrases makes
> No difference at all.
> —*Lewis Carroll, "Poeta Fit, Non Nascitur"*

To create a supple, knowledge-rich design calls for a versatile, shared team language, and a lively experimentation with language that seldom happens on software projects.

* * *

Domain experts have limited understanding of the technical jargon of software development, but they use the jargon of their field —probably in various flavors. Developers, on the other hand, may understand and discuss the system in descriptive, functional terms, devoid of the meaning carried by the experts' language. Or developers may create abstractions that support their design but are not understood by the domain experts. Developers working on different parts of the problem work out their own design concepts and ways of describing the domain.

Across this linguistic divide, the domain experts vaguely describe what they want. Developers, struggling to understand a domain new to them, vaguely understand. A few members of the team manage to become bilingual, but they become bottlenecks of information flow, and their translations are inexact.

On a project without a common language, developers have to translate for domain experts. Domain experts translate between developers and still other domain experts. Developers even translate for each other. Translation muddles model concepts, which leads to destructive refactoring of code. The indirectness of communication conceals the formation of schisms—different team members use terms differently but don't realize it. This leads to unreliable software that doesn't fit together (see Chapter 14). The effort of translation prevents the interplay of knowledge and ideas that lead to deep model insights.