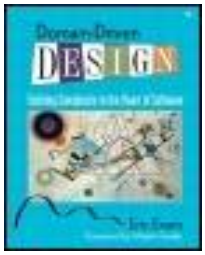


Domain-Driven Design: Tackling Complexity in the Heart of Software

By Eric Evans

Start Reading

ISBN: 0-321-12521-5
Pages: 560

- Table of Contents

Top

Copyright

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
(317) 581-3793
international@pearsontechgroup.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

Text printed on recycled paper

1 2 3 4 5 6 7 8 9 10—CRW—0706050403

First printing, August 2003

Dedication

To Mom and Dad

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 *dujour* 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*

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.

Preface

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 over-came 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.

Previous Next
Top

Preface

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.

Preface

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 muchneeded 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 misinter-preted. 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.

Preface

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 alignment 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 openended, 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 over-simplified, "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.

Preface

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 through-out 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.

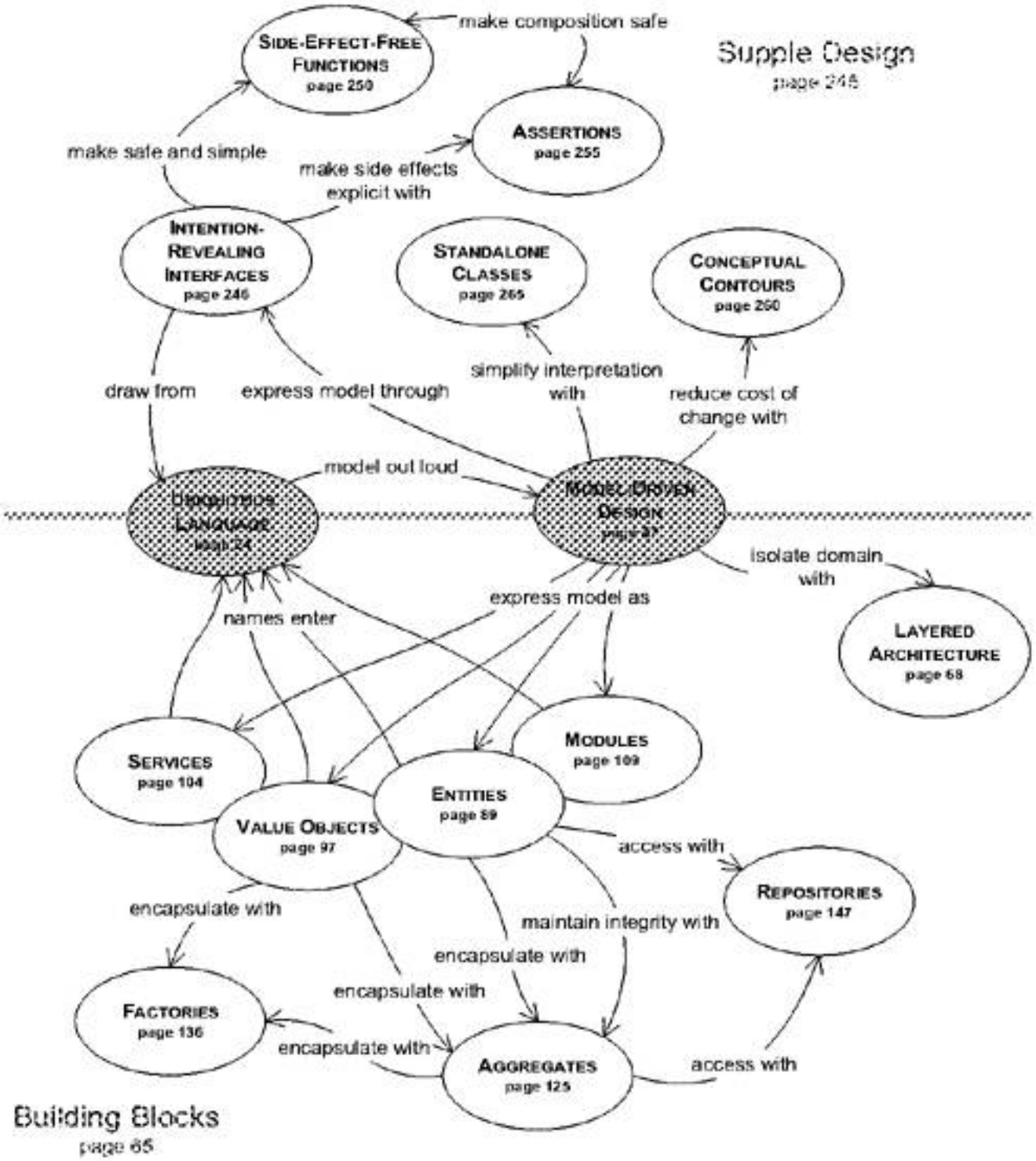
Previous Next
Top

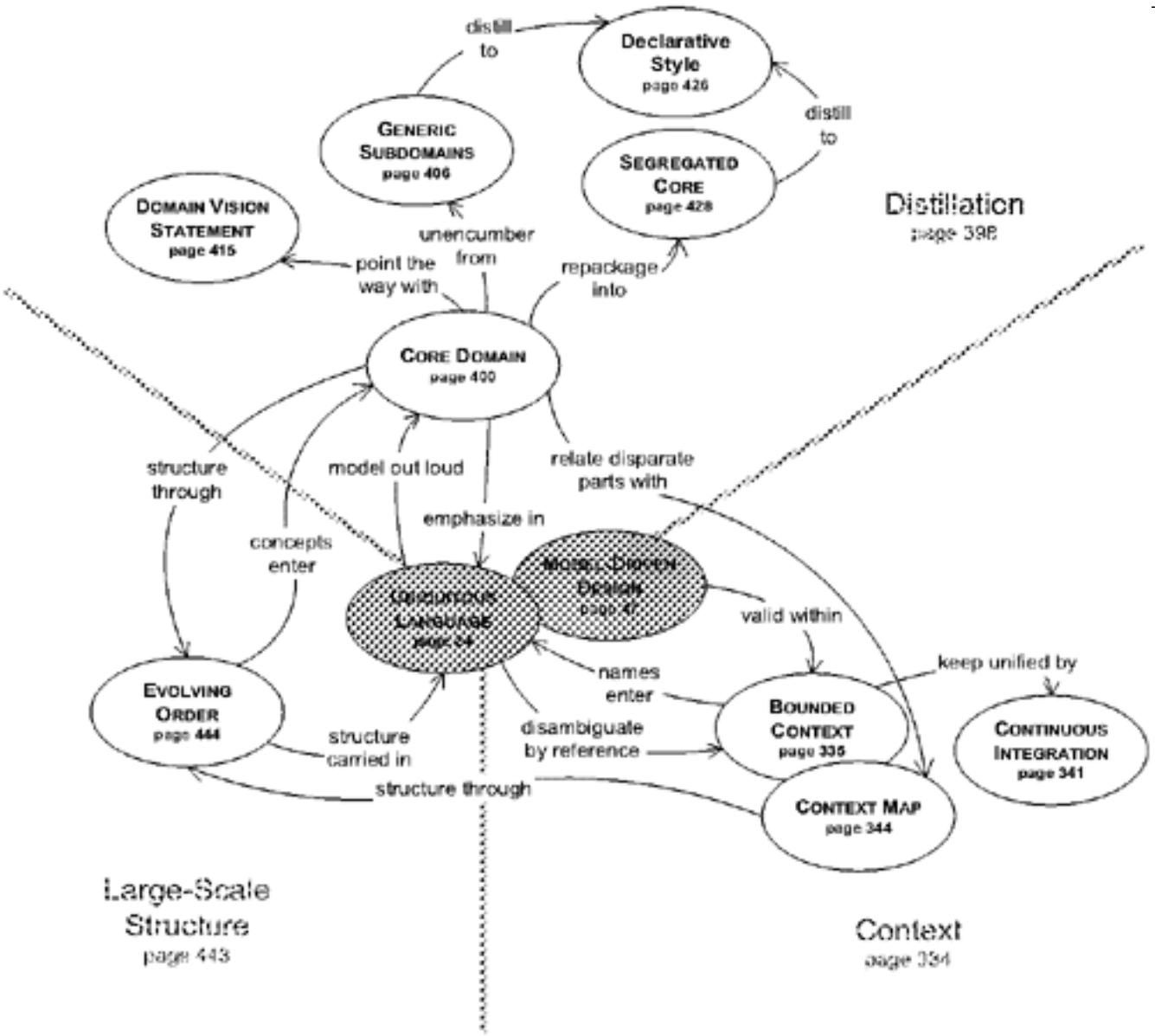
Preface

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.


Supple Design
page 244
SIDE-EFFECT-FREE FUNCTIONS
page 250
make composition safe
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
LANGUAGE
page 24
MODEL-DRIVEN DESIGN
page 97
isolate domain with
LAYERED ARCHITECTURE
page 68
names enter
express model as
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 85



Previous Next
Top

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