Covers through Version 2.0 OMG UML Standard

UML Distilled Third Edition

A Brief Guide to the Standard Object Modeling Language

MARTIN FOWLER

Forewords by Cris Kobryn, Grady Booch, Ivar Jacobson, and Jim Rumbaugh

OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
ADDISON-WESLEY
SERIES EDITORS

Praise for *UML Distilled*

"*UML Distilled* remains the best introduction to UML notation. Martin's agile and pragmatic approach hits the sweet spot, and I wholeheartedly recommend it!"

—Craig Larman
Author of *Applying UML and Patterns*

"Fowler cuts through the complexity of UML to get users started quickly."

—Jim Rumbaugh
Author and originator of UML

"Martin Fowler's *UML Distilled* is an excellent way to get started with UML. In fact for most users, *UML Distilled* contains all you need to apply UML successfully. As Martin points out, UML can be used in many ways, but the most common is as a widely recognized notation for sketching designs. This book does an excellent job of distilling the essence of UML. Highly recommended."

—Steve Cook
Software Architect
Microsoft Corporation

"Short books on UML are better than long books on UML. This is still the best short book on UML. In fact, it's the best short book on many subjects."

—Alistair Cockburn
Author and President, Humans and Technology

"The book is immensely useful, readable, and—one of its great virtues—delightfully concise for the immense scope of its subject. If you only buy one book on UML, this should be it."

—Andy Carmichael
BetterSoftwareFaster, Ltd.

"If you're using UML, this book should never be out of reach."

—John Crupi
Distinguished Engineer, Sun Microsystems
Coauthor of *Core J2EE™ Patterns*

“Anyone doing UML modeling, learning UML, reading UML, or building UML tools should have this latest edition. (I own all editions.) There is lots of good, useful information; generally, just enough to be useful, but not too much to be dry. It’s a must-have reference for my bookshelf!”

—Jon Kern
Modeler

“This is a great starting point for learning the fundamentals of the UML.”

—Scott W. Ambler
Author of *Agile Modeling*

“An eminently sensible description of UML and its usage, with enough humor to hold one’s attention. ‘The swimming metaphor no longer holds water’ indeed!”

—Stephen J. Mellor
Coauthor of *Executable UML*

“This is the perfect book for those who want to use the UML but aren’t interested in reading thick UML reference books and research papers. Martin Fowler selects all the critical techniques needed to use the UML for design sketches, freeing the reader from complex and rarely used UML features. Readers will find no shortage of suggestions for further reading. He gives the reader advice based on experience. It’s a concise and readable book covering the essential aspects of the UML and related object-oriented concepts.”

—Pavel Hruby
Microsoft Business Solutions

“Like all good software developers, Fowler improves his product with each iteration. This is the only book I consider when teaching a class involving UML or if asked to recommend one that can be used to learn it.”

—Charles Ashbacher
President/CEO, Charles Ashbacher Technologies

“More books should be like *UML Distilled*—concise and readable. Martin Fowler selects the parts of UML that you need, and presents them in an easy to read style. More valuable than a mere description of the modeling language, however, is the author’s insight and experience in how to use this technique to communicate and document design.”

—Rob Purser
Purser Consulting, LLC.

UML Distilled

Third Edition

UML Distilled
Third Edition

A Brief Guide to the Standard Object Modeling Language

Martin Fowler

Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Fowler, Martin, 1963–
UML distilled : a brief guide to the standard object modeling language / Martin Fowler.—3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-19368-7 (alk. paper)
1. Object-oriented methods (Computer science) 2. Computer software—Development. 3. UML (Computer science) I. Title.

QA76.9.O35F695 2003
005.1'7—dc22 2003057759

ISBN-13: 978-0-321-19368-1
ISBN-10: 0-321-19368-7

22 17

For Cindy

Contents

This page intentionally left blank

List of Figures

This page intentionally left blank

Foreword to the Third Edition

Since ancient times, the most talented architects and the most gifted designers have known the law of parsimony. Whether it is stated as a paradox ("less is more"), or a koan ("Zen mind is beginner's mind"), its wisdom is timeless: Reduce everything to its essence so that form harmonizes with function. From the pyramids to the Sydney Opera House, from von Neumann architectures to UNIX and Smalltalk, the best architects and designers have strived to follow this universal and eternal principle.

Recognizing the value of shaving with Occam's Razor, when I architect and read I seek projects and books that adhere to the law of parsimony. Consequently, I applaud the book you are reading now.

You may find my last remark surprising at first. I am frequently associated with the voluminous and dense specifications that define the Unified Modeling Language (UML). These specifications allow tool vendors to implement the UML and methodologists to apply it. For seven years, I have chaired large international standardization teams to specify UML 1.1 and UML 2.0, as well as several minor revisions in between. During this time, the UML has matured in expressiveness and precision, but it has also added gratuitous complexity as a result of the standardization process. Regrettably, standardization processes are better known for design-by-committee compromises than parsimonious elegance.

What can a UML expert familiar with the arcane minutiae of the specification learn from Martin's distillation of UML 2.0? Quite a bit, as can you. To start with, Martin adroitly reduces a large and complex language into a pragmatic subset that he has proven effective in his practice. He has resisted the easy route of tacking on additional pages to the last edition of his book. As the language has grown, Martin has kept true to his goal of seeking the "fraction of UML that is most useful" and telling you just that. The fraction he refers to is the mythical 20 percent of UML that helps you do 80 percent of your work. Capturing and taming this elusive beast is no mean accomplishment!

It is even more impressive that Martin achieves this goal while writing in a wonderfully engaging conversational style. By sharing his opinions and anecdotes with us, he makes this book fun to read and reminds us that architecting and designing systems should be both creative and productive. If we pursue the parsimony koan to its full intent, we should find UML modeling projects to be as enjoyable as we found finger-painting and drawing classes in grammar school. UML should be a lightning rod for our creativity as well as a laser for precisely specifying system blueprints so that third parties can bid and build those systems. The latter is the acid test for any bona fide blueprint language.

So, while this may be a small book, it is not a trivial one. You can learn as much from Martin's approach to modeling as you can learn from his explanations of UML 2.0.

I have enjoyed working with Martin to improve the selection and correctness of the UML 2.0 language features explained in this revision. We need to keep in mind that all living languages, both natural and synthetic, must evolve or perish. Martin's choices of new features, along with your preferences and those of other practitioners, are a crucial part of the UML revision process. They keep the language vital and help it evolve via natural selection in the marketplace.

Much challenging work remains before model-driven development becomes mainstream, but I am encouraged by books like this that explain UML modeling basics clearly and apply them pragmatically. I hope you will learn from it as I have and will use your new insights to improve your own software modeling practices.

Cris Kobryn
Chair, U2 Partners' UML 2.0 Submission Team
Chief Technologist, Telelogic

Foreword to the First Edition

When we began to craft the Unified Modeling Language, we hoped that we could produce a standard means of expressing design that would not only reflect the best practices of industry, but would also help demystify the process of software system modeling. We believed that the availability of a standard modeling language would encourage more developers to model their software systems before building them. The rapid and widespread adoption of the UML demonstrates that the benefits of modeling are indeed well known to the developer community.

The creation of the UML was itself an iterative and incremental process very similar to the modeling of a large software system. The end result is a standard built on, and reflective of, the many ideas and contributions made by numerous individuals and companies from the object community. We began the UML effort, but many others helped bring it to a successful conclusion; we are grateful for their contribution.

Creating and agreeing on a standard modeling language is a significant challenge by itself. Educating the development community, and presenting the UML in a manner that is both accessible and in the context of the software development process, is also a significant challenge. In this deceptively short book, updated to reflect the most recent changes to the UML, Martin Fowler has more than met this challenge.

In a clear and friendly style, Martin not only introduces the key aspects of UML, but also clearly demonstrates the role UML plays in the development process. Along the way, we are treated to abundant nuggets of modeling insight and wisdom drawn from Martin's 12-plus years of design and modeling experience.

The result is a book that has introduced many thousands of developers to UML, whetting their appetite to further explore the many benefits of modeling with this now standard modeling language.

We recommend the book to any modeler or developer interested in getting a first look at UML and in gaining a perspective on the key role it plays in the development process.

Grady Booch
Ivar Jacobson
James Rumbaugh

Preface

I've been lucky in a lot of ways in my life; one of my great strokes of fortune was being in the right place with the right knowledge to write the first edition of this book in 1997. Back then, the chaotic world of object-oriented (OO) modeling was just beginning to unify under the Unified Modeling Language (UML). Since then, the UML has become the standard for the graphical modeling of software, not just for objects. My fortune is that this book has been the most popular book on the UML, selling more than a quarter of a million copies.

Well, that's very nice for me, but should you buy this book?

I like to stress that this is a brief book. It's not intended to give you the details on every facet of the UML, which has grown and grown over the years. My intention is to find that fraction of the UML that is most useful and tell you just that. Although a bigger book gives you more detail, it also takes longer to read. And your time is the biggest investment you'll make in a book. By keeping this book small, I've spent the time selecting the best bits to save you from having to do that selection yourself. (Sadly, being smaller doesn't mean proportionately cheaper; there is a certain fixed cost to producing a quality technical book.)

One reason to have this book is to begin to learn about the UML. Because this is a short book, it will quickly get you up to speed on the essentials of the UML. With that under your belt, you can go into more detail on the UML with the bigger books, such as the *User Guide* [Booch, UML user] or the *Reference Manual* [Rumbaugh, UML Reference].

This book can also act as a handy reference to the most common parts of the UML. Although the book doesn't cover everything, it's a lot lighter to carry around than most other UML books.

It's also an opinionated book. I've been working with objects for a long time now, and I have definite ideas about what works and what doesn't. Any book reflects the opinions of the author, and I don't try to hide mine. So if you're looking for something that has a flavor of objectivity, you might want to try something else.

Although many people have told me that this book is a good introduction to objects, I didn't write it with that in mind. If you are after an introduction to OO design, I suggest Craig Larman's book [Larman].

Many people who are interested in the UML are using tools. This book concentrates on the standard and on conventional usage of the UML and doesn't get into the details of what various tools support. Although the UML did resolve the tower of Babel of pre-UML notations, many annoying differences remain between what tools show and allow when drawing UML diagrams.

I don't say much in this book about Model Driven Architecture (MDA). Although many people consider the two to be the same thing, many developers use the UML without being interested in MDA. If you want to learn more about MDA, I would start with this book to get an overview of the UML first and then move on to a book that's more specific about MDA.

Although the main point of this book is the UML, I've also added bits of other material about techniques, such as CRC cards, that are valuable for OO design. The UML is just a part of what you need to succeed with objects, and I think that it's important to introduce you to some other techniques.

In a brief book like this, it's impossible to go into detail about how the UML relates to source code, particularly as there is no standard way of making that correspondence. However, I do point out common coding techniques for implementing pieces of the UML. My code examples are in Java and C#, as I've found that these languages are usually the most widely understood. Don't assume that I prefer those languages; I've done too much Smalltalk for that!

Why Bother with the UML?

Graphical design notations have been with us for a while. For me, their primary value is in communication and understanding. A good diagram can often help communicate ideas about a design, particularly when you want to avoid a lot of details. Diagrams can also help you understand either a software system or a business process. As part of a team trying to figure out something, diagrams both help understanding and communicate that understanding throughout a team. Although they aren't, at least yet, a replacement for textual programming languages, they are a helpful assistant.

Many people believe that in the future, graphical techniques will play a dominant role in software development. I'm more skeptical of that, but it's certainly useful to have an appreciation of what these notations can and can't do.

Of these graphical notations, the UML's importance comes from its wide use and standardization within the OO development community. The UML has become not only the dominant graphical notation within the OO world but also a popular technique in non-OO circles.

Structure of the Book

Chapter 1 gives an introduction to the UML: what it is, the different meanings it has to different people, and where it came from.

Chapter 2 talks about software process. Although this is strictly independent of the UML, I think that it's essential to understand process in order to see the context of something like the UML. In particular, it's important to understand the role of iterative development, which has been the underlying approach to process for most of the OO community.

I've organized the rest of the book around the diagram types within the UML. Chapters 3 and 4 discuss the two most useful parts of the UML: class diagrams (core) and sequence diagrams. Even though this book is slim, I believe that you can get the most value out of the UML by using the techniques that I talk about in these chapters. The UML is a large and growing beast, but you don't need all of it.

Chapter 5 goes into detail on the less essential but still useful parts of class diagrams. Chapters 6 through 8 describe three useful diagrams that shed further light on the *structure* of a system: object diagrams, package diagrams, and deployment diagrams.

Chapters 9 through 11 show three further useful *behavioral* techniques: use cases, state diagrams (although officially known as state machine diagrams, they are generally called state diagrams), and activity diagrams. Chapters 12 through 17 are very brief and cover diagrams that are generally less important, so for these, I've only provided a quick example and explanation.

The inside covers summarize the most useful parts of the notation. I've often heard people say that these covers are the most valuable part of the book. You'll probably find it handy to refer to them as you're reading some of the other parts of the book.

Changes for the Third Edition

If you have earlier editions of this book, you're probably wondering what is different and, more important, whether you should buy the new edition.

The primary trigger for the third edition was the appearance of UML 2. UML 2 has added a lot of new stuff, including several new diagram types. Even familiar diagrams have a lot of new notation, such as interaction frames in sequence diagrams. If you want to be aware of what's happened but don't want to wade through the specification (I certainly don't recommend that!), this book should give you a good overview.

I've also taken this opportunity to completely rewrite most of the book, bringing the text and examples up to date. I've incorporated much that I've learned in teaching and using the UML over the past five years. So although the spirit of this ultrathin UML book is intact, most of the words are new.

Over the years, I've worked hard to keep this book as current as is possible. As the UML has gone through its changes, I've done my best to keep pace. This book is based on the UML 2 drafts that were accepted by the relevant committee in June 2003. It's unlikely that further changes will occur between that vote and more formal votes, so I feel that UML 2 is now stable enough for my revision to go into print. I'll post information any further updates on my Web site (**http://martinfowler.com**).

Acknowledgments

Over many years, many people have been part of the success of this book. My first thanks go Carter Shanklin and Kendall Scott. Carter was the editor at Addison-Wesley who suggested this book to me. Kendall Scott helped me put together the first two editions, working over the text and graphics. Between them, they pulled off the impossible in getting the first edition out in an impossibly short time, while keeping up the high quality that people expect from Addison-Wesley. They also kept pushing out changes during the early days of the UML when nothing seemed stable.

Jim Odell has been my mentor and guide for much of the early part of my career. He's also been deeply involved with the technical and personal issues of making opinionated methodologists settle their differences and agree to a common standard. His contribution to this book is both profound and difficult to measure, and I bet it's the same for the UML too.

The UML is a creature of standards, but I'm allergic to standards bodies. So to know what's going on, I need a network of spies who can keep me up to date on all the machinations of the committees. Without these spies, including Conrad Bock, Steve Cook, Cris Kobryn, Jim Odell, Guus Ramackers, and Jim

Rumbaugh, I would be sunk. They've all given me useful tips and answered stupid questions.

Grady Booch, Ivar Jacobson, and Jim Rumbaugh are known as the Three Amigos. Despite the playful jibes I've given them over the years, they have given me much support and encouragement with this book. Never forget that my jabs usually sprout from fond appreciation.

Reviewers are the key to a book's quality, and I learned from Carter that you can never have too many reviewers. The reviewers of the previous editions of this book were Simmi Kochhar Bhargava, Grady Booch, Eric Evans, Tom Hadfield, Ivar Jacobson, Ronald E. Jeffries, Joshua Kerievsky, Helen Klein, Jim Odell, Jim Rumbaugh, and Vivek Salgar.

The third edition also had a fine group of reviewers:

Conrad Bock
Andy Carmichael
Alistair Cockburn
Steve Cook
Luke Hohmann
Pavel Hruby
Jon Kern
Cris Kobryn
Craig Larman
Steve Mellor
Jim Odell
Alan O'Callaghan
Guus Ramackers
Jim Rumbaugh
Tim Seltzer

All these reviewers spent time reading the manuscript, and every one of them found at least one embarrassing howler. My sincere thanks to all of them. Any howlers that remain are entirely my responsibility. I will post an errata sheet to the books section of **martinfowler.com** when I find them.

The core team that designed and wrote the UML specification are Don Baisley, Morgan Björkander, Conrad Bock, Steve Cook, Philippe Desfray, Nathan Dykman, Anders Ek, David Frankel, Eran Gery, Øystein Haugen, Sridhar Iyengar, Cris Kobryn, Birger Møller-Pedersen, James Odell, Gunnar Övergaard, Karin Palmkvist, Guus Ramackers, Jim Rumbaugh, Bran Selic, Thomas Weigert, and Larry Williams. Without them, I would have nothing to write about.

Pavel Hruby developed some excellent Visio templates that I use a lot for UML diagrams; you can get them at **http://phruby.com**.

Many people have contacted me on the Net and in person with suggestions and questions and to point out errors. I haven't been able to keep track of you all, but my thanks are no less sincere.

The people at my favorite technical bookstore, SoftPro in Burlington, Massachusetts, let me spend many hours there looking at their stock to find how people use the UML in practice and fed me good coffee while I was there.

For the third edition, the acquisition editor was Mike Hendrickson. Kim Arney Mulcahy managed the project, as well as did the layout and clean-up of the diagrams. John Fuller, at Addison-Wesley, was the production editor, while Evelyn Pyle and Rebecca Rider helped with the copyediting and proofreading of the book. I thank them all.

Cindy has stayed with me while I persist in writing books. She then plants the proceeds in the garden.

My parents started me off with a good education, from which all else springs.

Martin Fowler
Melrose, Massachusetts
http://martinfowler.com

Chapter 1

Introduction

What Is the UML?

The Unified Modeling Language (UML) is a family of graphical notations, backed by single meta-model, that help in describing and designing software systems, particularly software systems built using the object-oriented (OO) style. That's a somewhat simplified definition. In fact, the UML is a few different things to different people. This comes both from its own history and from the different views that people have about what makes an effective software engineering process. As a result, my task in much of this chapter is to set the scene for this book by explaining the different ways in which people see and use the UML.

Graphical modeling languages have been around in the software industry for a long time. The fundamental driver behind them all is that programming languages are not at a high enough level of abstraction to facilitate discussions about design.

Despite the fact that graphical modeling languages have been around for a long time, there is an enormous amount of dispute in the software industry about their role. These disputes play directly into how people perceive the role of the UML itself.

The UML is a relatively open standard, controlled by the Object Management Group (OMG), an open consortium of companies. The OMG was formed to build standards that supported interoperability, specifically the interoperability of object-oriented systems. The OMG is perhaps best known for the CORBA (Common Object Request Broker Architecture) standards.

The UML was born out of the unification of the many object-oriented graphical modeling languages that thrived in the late 1980s and early 1990s. Since its appearance in 1997, it has relegated that particular tower of Babel to history. That's a service I, and many other developers, am deeply thankful for.

Ways of Using the UML

At the heart of the role of the UML in software development are the different ways in which people want to use it, differences that carry over from other graphical modeling languages. These differences lead to long and difficult arguments about how the UML should be used.

To untangle this, Steve Mellor and I independently came up with a characterization of the three modes in which people use the UML: sketch, blueprint, and programming language. By far the most common of the three, at least to my biased eye, is **UML as sketch**. In this usage, developers use the UML to help communicate some aspects of a system. As with blueprints, you can use sketches in a forward-engineering or reverse-engineering direction. **Forward engineering** draws a UML diagram before you write code, while **reverse engineering** builds a UML diagram from existing code in order to help understand it.

The essence of sketching is selectivity. With forward sketching, you rough out some issues in code you are about to write, usually discussing them with a group of people on your team. Your aim is to use the sketches to help communicate ideas and alternatives about what you're about to do. You don't talk about all the code you are going to work on, only important issues that you want to run past your colleagues first or sections of the design that you want to visualize before you begin programming. Sessions like this can be very short: a 10-minute session to discuss a few hours of programming or a day to discuss a 2-week iteration.

With reverse engineering, you use sketches to explain how some part of a system works. You don't show every class, simply those that are interesting and worth talking about before you dig into the code.

Because sketching is pretty informal and dynamic, you need to do it quickly and collaboratively, so a common medium is a whiteboard. Sketches are also useful in documents, in which case the focus is communication rather than completeness. The tools used for sketching are lightweight drawing tools, and often people aren't too particular about keeping to every strict rule of the UML. Most UML diagrams shown in books, such as my other books, are sketches. Their emphasis is on selective communication rather than complete specification.

In contrast, **UML as blueprint** is about completeness. In forward engineering, the idea is that blueprints are developed by a designer whose job is to build a detailed design for a programmer to code up. That design should be sufficiently complete in that all design decisions are laid out, and the programmer should be able to follow it as a pretty straightforward activity that requires little thought. The designer may be the same person as the programmer, but usually the designer is a more senior developer who designs for a team of programmers. The inspiration for this approach is other forms of engineering in which professional engineers create engineering drawings that are handed over to construction companies to build.

Blueprinting may be used for all details, or a designer may draw blueprints to a particular area. A common approach is for a designer to develop blueprint-level models as far as interfaces of subsystems but then let developers work out the details of implementing those details.

In reverse engineering, blueprints aim to convey detailed information about the code either in paper documents or as an interactive graphical browser. The blueprints can show every detail about a class in a graphical form that's easier for developers to understand.

Blueprints require much more sophisticated tools than sketches do in order to handle the details required for the task. Specialized CASE (computer-aided software engineering) tools fall into this category, although the term CASE has become a dirty word, and vendors try to avoid it now. Forward-engineering tools support diagram drawing and back it up with a repository to hold the information. Reverse-engineering tools read source code and interpret from it into the repository and generate diagrams. Tools that can do both forward and reverse engineering like this are referred to as **round-trip** tools.

Some tools use the source code itself as the repository and use diagrams as a graphic viewport on the code. These tools tie much more closely into programming and often integrate directly with programming editors. I like to think of these as **tripless** tools.

The line between blueprints and sketches is somewhat blurry, but the distinction, I think, rests on the fact that sketches are deliberately incomplete, highlighting important information, while blueprints intend to be comprehensive, often with the aim of reducing programming to a simple and fairly mechanical activity. In a sound bite, I'd say that sketches are explorative, while blueprints are definitive.

As you do more and more in the UML and the programming gets increasingly mechanical, it becomes obvious that the programming should be automated. Indeed, many CASE tools do some form of code generation, which automates building a significant part of a system. Eventually, however, you reach the point at which all the system can be specified in the UML, and you reach **UML as programming language**. In this environment, developers draw UML diagrams that are compiled directly to executable code, and the UML becomes the source code. Obviously, this usage of UML demands particularly sophisticated tooling. (Also, the notions of forward and reverse engineering don't make any sense for this mode, as the UML and source code are the same thing.)

Model Driven Architecture and Executable UML

When people talk about the UML, they also often talk about **Model Driven Architecture** (**MDA**) [Kleppe et al.]. Essentially, MDA is a standard approach to using the UML as a programming language; the standard is controlled by the OMG, as is the UML. By producing a modeling environment that conforms to the MDA, vendors can create models that can also work with other MDA-compliant environments.

MDA is often talked about in the same breath as the UML because MDA uses the UML as its basic modeling language. But, of course, you don't have to be using MDA to use the UML.

MDA divides development work into two main areas. Modelers represent a particular application by creating a **Platform Independent Model** (**PIM**). The PIM is a UML model that is independent of any particular technology. Tools can then turn a PIM into a **Platform Specific Model** (**PSM**). The PSM is a model of a system targeted to a specific execution environment. Further tools then take the PSM and generate code for that platform. The PSM could be UML but doesn't have to be.

So if you want to build a warehousing system using MDA, you would start by creating a single PIM of your warehousing system. If you then wanted this warehousing system to run on J2EE and .NET, you would use some vendor tools to create two PSMs: one for each platform. Then further tools would generate code for the two platforms.

If the process of going from PIM to PSM to final code is completely automated, we have the UML as programming language. If any of the steps is manual, we have blueprints.

Steve Mellor has long been active in this kind of work and has recently used the term **Executable UML** [Mellor and Balcer]. Executable UML is similar to MDA but uses slightly different terms. Similarly, you begin with a platform-independent model that is equivalent to MDA's PIM. However, the next step is to use a Model Compiler to turn that UML model into a deployable system in a single step; hence, there's no need for the PSM. As the term *compiler* suggests, this step is completely automatic.

The model compilers are based on reusable archetypes. An **archetype** describes how to take an executable UML model and turn it into a particular programming platform. So for the warehousing example, you would buy a model compiler and two archetypes (J2EE and .NET). Run each archetype on your executable UML model, and you have your two versions of the warehousing system.

Executable UML does not use the full UML standard; many constructs of UML are considered to be unnecessary and are therefore not used. As a result, Executable UML is simpler than full UML.

All this sounds good, but how realistic is it? In my view, there are two issues here. First is the question of the tools: whether they are mature enough to do the job. This is something that changes over time; certainly, as I write this, they aren't widely used, and I haven't seen much of them in action.

A more fundamental issue is the whole notion of the UML as a programming language. In my view, it's worth using the UML as a programming language only if it results in something that's significantly more productive than using another programming language. I'm not convinced that it is, based on various graphical development environments I've worked with in the past. Even if it is more productive, it still needs to get a critical mass of users for it to make the mainstream. That's a big hurdle in itself. Like many old Smalltalkers, I consider Smalltalk to be much more productive than current mainstream languages. But as Smalltalk is now only a niche language, I don't see many projects using it. To avoid Smalltalk's fate, the UML has to be luckier, even if it is superior.

One of the interesting questions around the UML as programming language is how to model behavioral logic. UML 2 offers three ways of behavioral modeling: interaction diagrams, state diagrams, and activity diagrams. All have their proponents for programming in. If the UML does gain popularity as a programming language, it will be interesting to see which of these techniques become successful.

Another way in which people look at the UML is the range between using it for conceptual and for software modeling. Most people are familiar with the UML used for software modeling. In this **software perspective**, the elements of the UML map pretty directly to elements in a software system. As we shall see, the mapping is by no means prescriptive, but when we use the UML, we are talking about software elements.

With the **conceptual perspective**, the UML represents a description of the concepts of a domain of study. Here, we aren't talking about software elements so much as we are building a vocabulary to talk about a particular domain.

There are no hard-and-fast rules about perspective; as it turns out, there's really quite a large range of usage. Some tools automatically turn source code into the UML diagrams, treating the UML as an alternative view of the source.

That's very much a software perspective. If you use UML diagrams to try and understand the various meanings of the terms *asset pool* with a bunch of accountants, you are in a much more conceptual frame of mind.

In previous editions of this book, I split the software perspective into specification (interface) and implementation. In practice, I found that it was too hard to draw a precise line between the two, so I feel that the distinction is no longer worth making a fuss about. However, I'm always inclined to emphasize interface rather than implementation in my diagrams.

These different ways of using the UML lead to a host of arguments about what UML diagrams mean and what their relationship is to the rest of the world. In particular, it affects the relationship between the UML and source code. Some people hold the view that the UML should be used to create a design that is independent of the programming language that's used for implementation. Others believe that language-independent design is an oxymoron, with a strong emphasis on the moron.

Another difference in viewpoints is what the essence of the UML is. In my view, most users of the UML, particularly sketchers, see the essence of the UML to be the diagrams. However, the creators of the UML see the diagrams as secondary; the essence of the UML is the meta-model. Diagrams are simply a presentation of the meta-model. This view also makes sense to blueprinters and UML programming language users.

So whenever you read anything involving the UML, it's important to understand the point of view of the author. Only then can you make sense of the often fierce arguments that the UML encourages.

Having said all that, I need to make my biases clear. Almost all the time, my use of the UML is as sketches. I find the UML sketches useful with forward and reverse engineering and in both conceptual and software perspectives.

I'm not a fan of detailed forward-engineered blueprints; I believe that it's too difficult to do well and slows down a development effort. Blueprinting to a level of subsystem interfaces is reasonable, but even then you should expect to change those interfaces as developers implement the interactions across the interface. The value of reverse-engineered blueprints is dependent on how the tool works. If it's used as a dynamic browser, it can be very helpful; if it generates a large document, all it does is kill trees.

I see the UML as programming language as a nice idea but doubt that it will ever see significant usage. I'm not convinced that graphical forms are more productive than textual forms for most programming tasks and that even if they are, it's very difficult for a language to be widely accepted.

As a result of my biases, this book focuses much more on using the UML for sketching. Fortunately, this makes sense for a brief guide. I can't do justice to the UML in its other modes in a book this size, but a book this size makes a good introduction to other books that can. So if you're interested in the UML in its other modes, I'd suggest that you treat this book as an introduction and move on to other books as you need them. If you're interested only in sketches, this book may well be all you need.

How We Got to the UML

I'll admit, I'm a history buff. My favorite idea of light reading is a good history book. But I also know that it's not everybody's idea of fun. I talk about history here because I think that in many ways, it's hard to understand where the UML is without understanding the history of how it got here.

In the 1980s, objects began to move away from the research labs and took their first steps toward the "real" world. Smalltalk stabilized into a platform that people could use, and C++ was born. At that time, various people started thinking about object-oriented graphical design languages.

The key books about object-oriented graphical modeling languages appeared between 1988 and 1992. Leading figures included Grady Booch [Booch, OOAD]; Peter Coad [Coad, OOA], [Coad, OOD]; Ivar Jacobson (Objectory) [Jacobson, OOSE]; Jim Odell [Odell]; Jim Rumbaugh (OMT) [Rumbaugh, insights], [Rumbaugh, OMT]; Sally Shlaer and Steve Mellor [Shlaer and Mellor, data], [Shlaer and Mellor, states]; and Rebecca Wirfs-Brock (Responsibility Driven Design) [Wirfs-Brock].

Each of those authors was now informally leading a group of practitioners who liked those ideas. All these methods were very similar, yet they contained a number of often annoying minor differences among them. The same basic concepts would appear in very different notations, which caused confusion to my clients.

During that heady time, standardization was as talked about as it was ignored. A team from the OMG tried to look at standardization but got only an open letter of protest from all the key methodologists. (This reminds me of an old joke. Question: What is the difference between a methodologist and a terrorist? Answer: You can negotiate with a terrorist.)

The cataclysmic event that first initiated the UML was when Jim Rumbaugh left GE to join Grady Booch at Rational (now a part of IBM). The Booch/Rumbaugh alliance was seen from the beginning as one that could get a critical mass of market share. Grady and Jim proclaimed that "the methods war is over—we won," basically declaring that they were going to achieve standardization "the Microsoft way." A number of other methodologists suggested forming an Anti-Booch Coalition.

By OOPSLA '95, Grady and Jim had prepared their first public description of their merged method: version 0.8 of the *Unified Method* documentation. Even more significant, they announced that Rational Software had bought Objectory and that therefore, Ivar Jacobson would be joining the Unified team. Rational held a well-attended party to celebrate the release of the 0.8 draft. (The highlight of the party was the first public display of Jim Rumbaugh's singing; we all hope it's also the last.)

The next year saw a more open process emerge. The OMG, which had mostly stood on the sidelines, now took an active role. Rational had to incorporate Ivar's ideas and also spent time with other partners. More important, the OMG decided to take a major role.

At this point, it's important to realize why the OMG got involved. Methodologists, like book authors, like to think that they are important. But I don't think that the screams of book authors would even be heard by the OMG. What got the OMG involved were the screams of tools vendors, all of which were frightened that a standard controlled by Rational would give Rational tools an unfair competitive advantage. As a result, the vendors energized the OMG to do something about it, under the banner of CASE tool interoperability. This banner was important, as the OMG was all about interoperability. The idea was to create a UML that would allow CASE tools to freely exchange models.

Mary Loomis and Jim Odell chaired the initial task force. Odell made it clear that he was prepared to give up his method to a standard, but he did not want a Rational-imposed standard. In January 1997, various organizations submitted proposals for a methods standard to facilitate the interchange of models. Rational collaborated with a number of other organizations and released version 1.0 of the UML documentation as their proposal, the first animal to answer to the name Unified Modeling Language.

Then followed a short period of arm twisting while the various proposals were merged. The OMG adopted the resulting 1.1 as an official OMG standard. Some revisions were made later on. Revision 1.2 was entirely cosmetic. Revision 1.3 was more significant. Revision 1.4 added a number of detailed concepts around components and profiles. Revision 1.5 added action semantics.

When people talk about the UML, they credit mainly Grady Booch, Ivar Jacobson, and Jim Rumbaugh as its creators. They are generally referred to as the Three Amigos, although wags like to drop the first syllable of the second word. Although they are most credited with the UML, I think it somewhat unfair to give them the dominant credit. The UML notation was first formed in the Booch/Rumbaugh Unified Method. Since then, much of the work has been led by OMG committees. During these later stages, Jim Rumbaugh is the only one of the three to have made a heavy commitment. My view is that it's these members of the UML committee process that deserve the principal credit for the UML.

Notations and Meta-Models

The UML, in its current state, defines a notation and a meta-model. The **notation** is the graphical stuff you see in models; it is the graphical syntax of the modeling language. For instance, class diagram notation defines how items and concepts, such as class, association, and multiplicity, are represented.

Of course, this leads to the question of what exactly is meant by an association or multiplicity or even a class. Common usage suggests some informal definitions, but many people want more rigor than that.

The idea of rigorous specification and design languages is most prevalent in the field of formal methods. In such techniques, designs and specifications are represented using some derivative of predicate calculus. Such definitions are mathematically rigorous and allow no ambiguity. However, the value of these definitions is by no means universal. Even if you can prove that a program satisfies a mathematical specification, there is no way to prove that the mathematical specification meets the real requirements of the system.

Most graphical modeling languages have very little rigor; their notation appeals to intuition rather than to formal definition. On the whole, this does not seem to have done much harm. These methods may be informal, but many people still find them useful—and it is usefulness that counts.

However, methodologists are looking for ways to improve the rigor of methods without sacrificing their usefulness. One way to do this is to define a **meta-model:** a diagram, usually a class diagram, that defines the concepts of the language.

Figure 1.1, a small piece of the UML meta-model, shows the relationship among features. (The extract is there to give you a flavor of what meta-models are like. I'm not even going to try to explain it.)

How much does the meta-model affect a user of the modeling notation? The answer depends mostly on the mode of usage. A sketcher usually doesn't care too much; a blueprinter should care rather more. It's vitally important to those who use the UML as a programming language, as it defines the abstract syntax of that language.

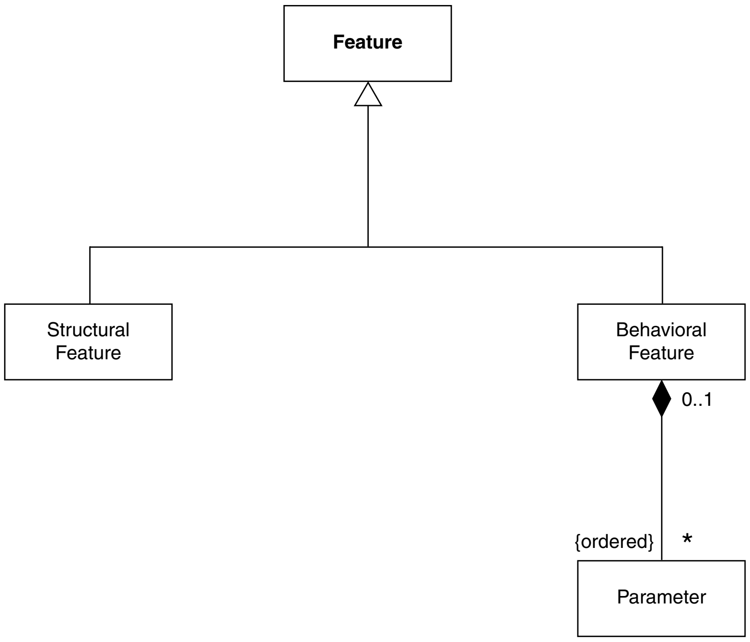

Figure 1.1 *A small piece of the UML meta-model*

Many of the people who are involved in the ongoing development of the UML are interested primarily in the meta-model, particularly as this is important to the usage of the UML as a programming language. Notational issues often run second place, which is important to bear in mind if you ever try to get familiar with the standards documents themselves.

As you get deeper into the more detailed usage of the UML, you realize that you need much more than the graphical notation. This is why UML tools are so complex.

I am not rigorous in this book. I prefer the traditional methods path and appeal mainly to your intuition. That's natural for a small book like this written by an author who's inclined mostly to a sketch usage. If you want more rigor, you should turn to more detailed tomes.

UML Diagrams

UML 2 describes 13 official diagram types listed in Table 1.1 and classified as indicated on Figure 1.2. Although these diagram types are the way many people

Table 1.1 *Official Diagram Types of the UML*

Diagram	Book Chapters	Purpose	Lineage
Activity	11	Procedural and parallel behavior	In UML 1
Class	3, 5	Class, features, and relationships	In UML 1
Communication	12	Interaction between objects; emphasis on links	UML 1 collaboration diagram
Component	14	Structure and connections of components	In UML 1
Composite structure	13	Runtime decomposition of a class	New to UML 2
Deployment	8	Deployment of artifacts to nodes	In UML 1
Interaction overview	16	Mix of sequence and activity diagram	New to UML 2
Object	6	Example configurations of instances	Unofficially in UML 1
Package	7	Compile-time hierarchic structure	Unofficially in UML 1
Sequence	4	Interaction between objects; emphasis on sequence	In UML 1
State machine	10	How events change an object over its life	In UML 1
Timing	17	Interaction between objects; emphasis on timing	New to UML 2
Use case	9	How users interact with a system	In UML 1

approach the UML and how I've organized this book, the UML's authors do not see diagrams as the central part of the UML. As a result, the diagram types are not particularly rigid. Often, you can legally use elements from one diagram type on another diagram. The UML standard indicates that certain elements are typically drawn on certain diagram types, but this is not a prescription.

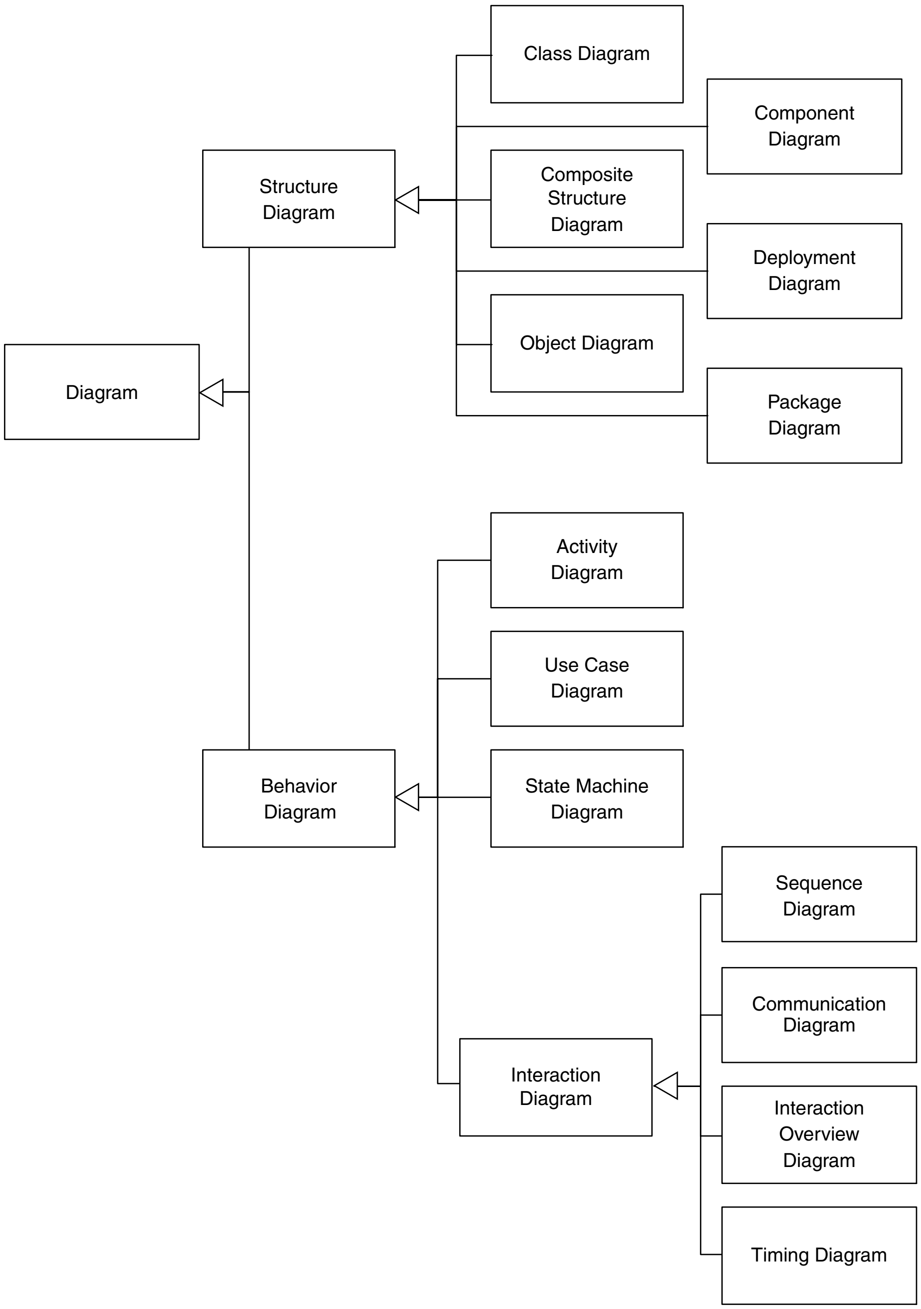

Figure 1.2 *Classification of UML diagram types*

What Is Legal UML?

At first blush, this should be a simple question to answer: Legal UML is what is defined as well formed in the specification. In practice, however, the answer is a bit more complicated.

An important part of this question is whether the UML has descriptive or prescriptive rules. A language with **prescriptive rules** is controlled by an official body that states what is or isn't legal in the language and what meaning you give to utterances in that language. A language with **descriptive rules** is one in which you understand its rules by looking at how people use the language in practice. Programming languages tend to have prescriptive rules set by a standards committee or dominant vendor, while natural languages, such as English, tend to have descriptive rules whose meaning is set by convention.

UML is quite a precise language, so you might expect it to have prescriptive rules. But UML is often considered to be the software equivalent of the blueprints in other engineering disciplines, and these blueprints are not prescriptive notations. No committee says what the legal symbols are on a structural engineering drawing; the notation has been accepted by convention, similarly to a natural language. Simply having a standards body doesn't do the trick either, because people in the field may not follow everything the standards body says; just ask the French about the Académie Française. In addition, the UML is so complex that the standard is often open to multiple interpretations. Even the UML leaders who reviewed this book would disagree on interpretation of the UML standard.

This issue is important both for me writing this book and for you using the UML. If you want to understand a UML diagram, it's important to realize that understanding the UML standard is not the whole picture. People do adopt conventions, both in the industry widely and within a particular project. As a result, although the UML standard can be the primary source of information on the UML, it can't be the only one.

My attitude is that, for most people, the UML has descriptive rules. The UML standard is the biggest single influence on what UML means, but it isn't the only one. I think that this will become particularly true with UML 2, which introduces some notational conventions that conflict with either UML 1's definition or the conventional usage of UML, as well as adds yet more complexity to the UML. In this book, therefore, I'm trying to summarize the UML as I find it: both the standards and the conventional usage. When I have to make a distinction in this book, I'll use the term **conventional use** to indicate something that isn't in the standard but that I think is widely used. For something that conforms to the standard, I'll use the terms **standard** or **normative**. (Normative is the term standards people use to mean a statement that you must conform to be valid in the standard. So non-normative UML is a fancy way of saying that something is strictly illegal according to the UML standard.)

When you are looking at a UML diagram, you should bear in mind that a general principle in the UML is that any information may be **suppressed** for a particular diagram. This suppression can occur either generally—hide all attributes—or specifically—don't show these three classes. In a diagram, therefore, you can never infer anything by its absence. If a multiplicity is missing, you cannot infer what value it might be. Even if the UML meta-model has a default, such as [1] for attributes, if you don't see the information on the diagram, it may be because it's the default or because it's suppressed.

Having said that, there are some general conventions, such as multivalued properties being sets. In the text, I'll point out these default conventions.

It's important to not put too much emphasis on having legal UML if you're a sketcher or blueprinter. It's more important to have a good design for your system, and I would rather have a good design in illegal UML than a legal but poor design. Obviously, good and legal is best, but you're better off putting your energy into having a good design than worrying about the arcana of UML. (Of course, you have to be legal in UML as programming language, or your program won't run properly!)

The Meaning of UML

One of the awkward issues about the UML is that, although the specification describes in great detail what well-formed UML is, it doesn't have much to say about what the UML means outside of the rarefied world of the UML meta-model. No formal definition exists of how the UML maps to any particular programming language. You cannot look at a UML diagram and say *exactly* what the equivalent code would look like. However, you can get a *rough idea* of what the code would look like. In practice, that's enough to be useful. Development teams often form their local conventions for these, and you'll need to be familiar with the ones in use.

UML Is Not Enough

Although the UML provides quite a considerable body of various diagrams that help to define an application, it's by no means a complete list of all the useful diagrams that you might want to use. In many places, different diagrams can be useful, and you shouldn't hesitate to use a non-UML diagram if no UML diagram suits your purpose.

Figure 1.3, a screen flow diagram, shows the various screens on a user interface and how you move between them. I've seen and used these screen flow diagrams for many years. I've never seen more than a very rough definition of what they mean; there isn't anything like it in the UML, yet I've found it a very useful diagram.

Table 1.2 shows another favorite: the decision table. Decision tables are a good way to show complicated logical conditions. You can do this with an activity diagram, but once you get beyond simple cases, the table is both more compact and more clear. Again, many forms of decision tables are out there. Table 1.2 divides the table into two sections: conditions above the double line and consequences below it. Each column shows how a particular combination of conditions leads to a particular set of consequences.

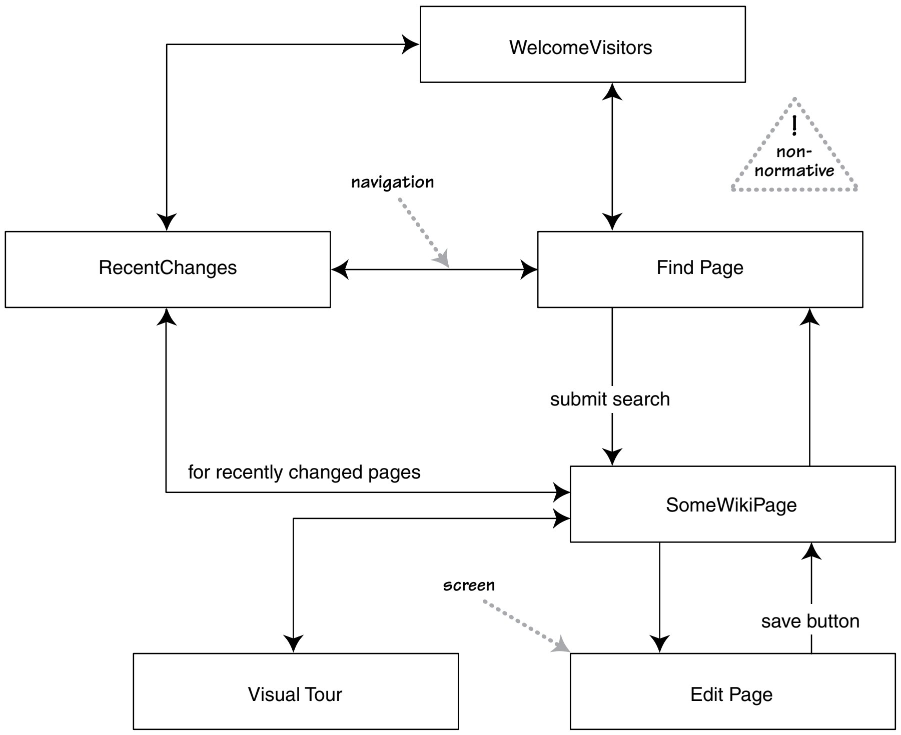

Figure 1.3 *An informal screen flow diagram for part of the wiki (http://c2.com/cgi/wiki)*

Table 1.2 *A Decision Table*

Premium customer	X	X	Y	Y	N	N
Priority order	Y	N	Y	N	Y	N
International order	Y	Y	N	N	N	N
Fee	$150	$100	$70	$50	$80	$60
Alert rep	•	•	•			

You'll run into various kinds of these things in various books. Don't hesitate to try out techniques that seem appropriate for your project. If they work well, use them. If not, discard them. (This is, of course, the same advice as for UML diagrams.)

Where to Start with the UML

Nobody, not even the creators of the UML, understand or use all of it. Most people use a small subset of the UML and work with that. You have to find the subset of the UML that works for you and your colleagues.

If you are starting out, I suggest that you concentrate first on the basic forms of class diagrams and sequence diagrams. These are the most common and, in my view, the most useful diagram types.

Once you've got the hang of those, you can start using some of the more advanced class diagram notation and take a look at the other diagrams types. Experiment with the diagrams and see how helpful they are to you. Don't be afraid to drop any that don't seem to be useful to your work.

Where to Find Out More

This book is not a complete and definitive reference to the UML, let alone OO analysis and design. A lot of words are out there and a lot of worthwhile things to read. As I discuss the individual topics, I also mention other books you should go to for more in-depth information there. Here are some general books on the UML and object-oriented design.

As with all book recommendations, you may need to check which version of the UML they are written for. As of June 2003, no published book uses UML 2.0, which is hardly surprising, as the ink is barely dry on the standard. The books I suggest are good books, but I can't tell whether or when they will be updated to the UML 2 standard.

If you are new to objects, I recommend my current favorite introductory book: [Larman]. The author's strong responsibility-driven approach to design is worth following.

For the conclusive word on the UML, you should look to the official standards documents; but remember, they are written for consenting methodologists in the privacy of their own cubicles. For a much more digestible version of the standard, take a look at [Rumbaugh, UML Reference].

For more detailed advice on object-oriented design, you'll learn many good things from [Martin].

I also suggest that you read books on patterns for material that will take you beyond the basics. Now that the methods war is over, patterns (page 27) are where most of the interesting material about analysis and design appears.

This page intentionally left blank

Index