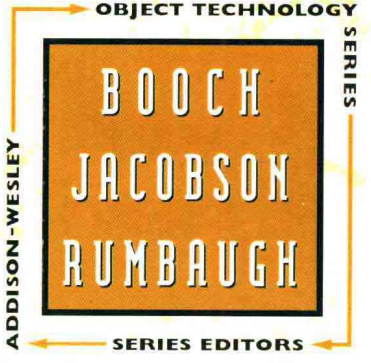
UML与面向对象设计影印丛书

用UML进行用况对象建模

APPLYING USE CASE DRIVEN OBJECT MODELING WIHT UML

AN ANNOTATED E-COMMERCE EXAMPLE

DOUG ROSENBERG
KENDALL SCOTT 编著



科学出版社
www.sciencep.com

UML与面向对象设计影印丛书

用UML进行用况对象建模

Doug Rosenberg
Kendall Scott 编著

科学出版社

北京

内 容 简 介

本书以一个电子商务系统为范例，详细介绍了其用况设计过程中的4个关键阶段：域建模、用况建模、稳定性分析、顺序图。对每个主题的介绍，都结合了一定的细节讨论、常见错误、练习题，易学易练。

本书适合面向对象系统设计人员阅读。

ISBN 0-201-73039-1

图字：01-2003-2544

图书在版编目(CIP)数据

用 UML 进行用况对象建模=Applying Use Case Driven Object Modeling With UML:An Annotated e-Commerce Example/（美）罗森堡（Rosenberg,D.），（美）斯科特（Scott,K.）著.—影印本. —北京：科学出版社，2003

ISBN 7-03-011406-X

Ⅰ.用… Ⅱ.①罗…②斯… Ⅲ.面向对象语言—UML—程序设计—英文 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2003）第 030812 号

策划编辑：李佩乾/责任编辑：李佩乾
责任印制：吕春珉/封面制作：东方人华平面设计室

科学出版社 出版
北京东黄城根北街16号
邮政编码:100717
http://www.sciencep.com

双青印刷厂 印刷
科学出版社发行 各地新华书店经销
*
2003年5月第 一 版 开本：787×960 1/16
2003年5月第一次印刷 印张：10 1/2
印数：1—2 000 字数：197 000

定价：25.00 元

（如有印装质量问题，我社负责调换<环伟>）

影印前言

随着计算机硬件性能的迅速提高和价格的持续下降，其应用范围也在不断扩大。交给计算机解决的问题也越来越难，越来越复杂。这就使得计算机软件变得越来越复杂和庞大。20 世纪 60 年代的软件危机使人们清醒地认识到按照工程化的方法组织软件开发的必要性。于是软件开发方法从 60 年代毫无工程性可言的手工作坊式开发，过渡到 70 年代结构化的分析设计方法、80 年代初的实体关系开发方法，直到面向对象的开发方法。

面向对象的软件开发方法是在结构化开发范型和实体关系开发范型的基础上发展而来的，它运用分类、封装、继承、消息等人类自然的思维机制，允许软件开发者处理更为复杂的问题域和其支持技术，在很大程度上缓解了软件危机。面向对象技术发端于程序设计语言，以后又向软件开发的早期阶段延伸，形成了面向对象的分析和设计。

20 世纪 80 年代末 90 年代初，先后出现了几十种面向对象的分析设计方法。其中，Booch, Coad/Yourdon、OMT 和 Jacobson 等方法得到了面向对象软件开发界的广泛认可。各种方法对许多面向对象的概念的理解不尽相同，即便概念相同，各自技术上的表示法也不同。通过 90 年代不同方法流派之间的争论，人们逐渐认识到不同的方法既有其容易解决的问题，又有其不容易解决的问题，彼此之间需要进行融合和借鉴；并且各种方法的表示也有很大的差异，不利于进一步的交流与协作。在这种情况下，统一建模语言(UML)于 90 年代中期应运而生。

UML 的产生离不开三位面向对象的方法论专家 G. Booch、J. Rumbaugh 和 I. Jacobson 的通力合作。他们从多种方法中吸收了大量有用的建模概念，使 UML 的概念和表示法在规模上超过了以往任何一种方法，并且提供了允许用户对语言做进一步扩展的机制。UML 使不同厂商开发的系统模型能够基于共同的概念，使用相同的表示法，呈现彼此一致的模型风格。1997 年 11 月 UML 被 OMG 组织正式采纳为标准的建模语言，并在随后的几年中迅速地发展为事实上的建模语言国际标准。

UML 在语法和语义的定义方面也做了大量的工作。以往各种关于面向对象方法的著作通常是以比较简单的方式定义其建模概念，而以主要篇幅给出过程指导，论述如何运用这些概念来进行开发。UML 则以一种建模语言的姿态出现，使用语言学中的一些技术来定义。尽管真正从语言学的角度看它还有许多缺陷，但它在这方面所做的努力却是以往的各种建模方法无法比拟的。

从 UML 的早期版本开始，便受到了计算机产业界的重视，OMG 的采纳和大公司的支持把它推上了实际上的工业标准的地位，使它拥有越来越多的用户。它被广泛地用

于应用领域和多种类型的系统建模，如管理信息系统、通信与控制系统、嵌入式实时系统、分布式系统、系统软件等。近几年还被运用于软件再工程、质量管理、过程管理、配置管理等方面。而且它的应用不仅仅限于计算机软件，还可用于非软件系统，例如硬件设计、业务处理流程、企业或事业单位的结构与行为建模，等等。

在 UML 陆续发布的几个版本中，逐步修正了前一个版本中的缺陷和错误。即将发布的 UML2.0 版本将是对 UML 的又一次重大的改进。将来的 UML 将向着语言家族化、可执行化、精确化等理念迈进，为软件产业的工程化提供更有力的支撑。

本丛书收录了与面向对象技术和 UML 有关的 12 本书，反映了面向对象技术最新的发展趋势以及 UML 的新的研究动态。其中涉及对面向对象建模理论研究与实践的有这样几本书：《面向对象系统架构及设计》主要讨论了面向对象的基本概念、静态设计、永久对象、动态设计、设计模式以及体系结构等近几年来面向对象技术领域中的新的理论知识与方法；《用 UML 进行用况对象建模》主要介绍了面向对象的需求阶段、分析阶段、设计阶段中用况模型的建立方法与技术；《高级用况建模》介绍了在建立用况模型中需要注意的高级的问题与技术；《UML 面向对象设计基础》则侧重于经典的面向对象理论知识的阐述。

涉及 UML 在特定领域的运用的有这样几本：《UML 实时系统开发》讨论了进行实时系统开发时需要对 UML 进行扩展的技术；《用 UML 构建 Web 应用程序》讨论了运用 UML 进行 Web 应用建模所应该注意的技术与方法；《面向对象系统测试：模型、视图与工具》介绍了将 UML 应用于面向对象的测试领域所应掌握的方法与工具；《对象、构件、框架与 UML 应用》讨论了如何运用 UML 对面向对象的新技术——构件-框架技术建模的方法策略。《UML 与 Visual Basic 应用程序开发》主要讨论了从 UML 模型到 Visual Basic 程序的建模与映射方法。

介绍面向对象编程技术的有两本书：《COM 高手心经》和《ATL 技术内幕》，深入探讨了面向对象的编程新技术——COM 和 ATL 技术的使用技巧与技术内幕。

还有一本《Executable UML 技术内幕》，这本书介绍了可执行 UML 的理念与其支持技术，使得模型的验证与模拟以及代码的自动生成成为可能，也代表着将来软件开发的一种新的模式。

总之，这套书所涉及的内容包含了对软件生命周期的全过程建模的方法与技术，同时也对近年来的热点领域建模技术、新型编程技术作了深入的介绍，有些内容已经涉及到了前沿领域。可以说，每一本都很经典。

有鉴于此，特向软件领域中不同程度的读者推荐这套书，供大家阅读、学习和研究。

北京大学计算机系　蒋严冰 博士

Preface

Theory, in Practice

In our first book, *Use Case Driven Object Modeling with UML*, we suggested that the difference between theory and practice was that in theory, there is no difference between theory and practice, but in practice, there is. In that book, we attempted to reduce OOAD modeling theory to a practical subset that was easy to learn and pretty much universally applicable, based on our experience in teaching this material to people working on hundreds of projects since about 1993.

Now, two years after hitting the shelves, that book is in its fifth printing. But even though our work has been favorably received, it seems like the job isn't all the way done yet. "We need to see more use case and UML modeling examples" is a phrase we've been hearing fairly often over the last couple of years. And, as we've used the first book as the backbone of training workshops where we apply the theory to real client projects, it has become clear that the process of reviewing the models is critically important and not well understood by many folks.

So, although we present a fairly extensive example in our first book, we convinced Addison-Wesley to let us produce this companion workbook, in which we dissect the design of an Internet bookstore, step-by-step, in great detail. This involves showing many common mistakes, and then showing the relevant pieces of the model with their mistakes corrected. We chose an Internet bookstore because it's relevant to many of today's projects in the Web-driven world, and because we've been teaching workshops using this example and, as a result, had a rich source of classroom UML models with real student mistakes in them.

We collected some of our favorite mistakes—that is, the kind of mistakes we saw getting repeated over and over again—and built this workbook around those models. And then we added three new chapters about reviews—one on requirements reviews, one on preliminary design reviews, and one on critical design reviews.

What really makes this book unique, though, is the fact that you, the reader, get to correct the mistakes.

The Premise

After we give you an overview of the ICONIX process in Chapter 1, four of the seven subsequent chapters address the four key phases of the process in some detail. The format of each of these chapters is as follows:

- The first part describes the essence of domain modeling (Chapter 2), use case modeling (Chapter 3), robustness analysis (Chapter 5), or sequence diagrams (Chapter 7), and places the material in the context of the "big picture" of the process. In each of these chapters, you'll work through pieces of the Internet bookstore example, and then you'll see an overview diagram at the end of the chapter that brings the relevant pieces together. We present fragments of ten different use cases in Chapter 3; we carry five of these forward through preliminary design and detailed design in Chapters 5 and 7, respectively. (The fragments of class diagrams that appear in Chapter 2 also trace into the use case text and to full class diagrams that appear in Chapters 5 and 7.)
- The next section describes the key elements of the given phase. Each of these sections is basically a condensed version of an associated chapter in *Use Case Driven Object Modeling with UML*, with some new information added within each chapter.
- The following section describes the top 10 mistakes that our students tend to make during workshops in which we teach the process. We've added five new Top 10 lists in this book: Top 10 robustness analysis errors, Top 10 sequence diagramming errors, and Top 10 mistakes to avoid for each of the three "review" chapters.
- The final section presents a set of five exercises for you to work, to test your knowledge of the material in the chapter.

The following aspects are common to each set of exercises:

- There's a red box, with a white label, at the top of each right-hand page. For the domain modeling and use case exercises, this label takes the form Exercise X; for the robustness analysis and sequence diagram exercises, the label takes the form of a use case name. (We'll explain the significance of this soon.)
- There are three or four mistakes on each right-hand page. Each mistake has a "Top 10" logo next to it that indicates which rule is being violated.
- The left-hand page on the flip side of each "red" page has a black box, with a white label, at the top. Corrections to the errors presented on the associated "bad" page are explicitly indicated; explanations of the mistakes appear at the bottom of the page.

Your task is to write corrections on each "bad" exercise page *before* you flip it over to see the "good" exercise diagram.

To summarize: Chapter 2 presents classes used by the ten sample use cases. Chapter 3 presents fragments from all of those use cases. Chapters 5 and 7 present diagrams connected with five of the use cases. The idea is that you'll move from a partial understanding of the use cases through to sequence diagrams that present full text, and some of the associated elements of the detailed design, for each use case.

What about the other three chapters, you ask?

- Chapter 4 describes how to perform requirements review, which involves trying to ensure that the use cases and the domain model work together to address the customers' functional requirements.

- Chapter 6 describes how to perform preliminary design review (PDR), which involves trying to ensure that robustness diagrams exist for all use cases (and are consistent with those use cases), the domain model has a fairly rich set of attributes that correspond well with whatever prototypes are in place (and all of the objects needed by the use cases are represented in that model), and the development team is ready to move to detailed design.
- Chapter 8 describes how to perform critical design review (CDR), which involves trying to ensure that the "how" of detailed design, as shown on sequence diagrams, matches up well with the "what" that the use cases specify, and that the detailed design is of sufficient depth to facilitate a relatively small and seamless leap into code.

All three of these review chapters offer overviews, details, and top 10 lists, but we don't make you work any more exercises. What these reviews have in common is the goal of ensuring consistency of the various parts of the model, as expressed on the "good" exercise diagrams.

The Appendix contains a report that summarizes the model for the bookstore; you can download the full model from **http://www.iconixsw.com/WorkbookExample.html**. The Appendix contains all of the diagrams that appear in the body of the book, but the full model includes design details for the other five use cases. This allows you to go through these use cases as further exercises, and then compare your results to ours; we highly recommend that you do this.

Cool premise, isn't it? We're not aware of another book like this one, and we're hoping you'll find it useful in your efforts to apply use case driven object modeling with UML.

Acknowledgments

Doug would like to thank his intrepid crew at ICONIX, especially Andrea Lee for her work on the script for the Inside the ICONIX Process CD, which we borrowed heavily from for Chapter 1, along with Chris Starczak, Jeff Kantor, and Erin Arnold. Doug would also like to thank Kendall for (finally) agreeing that yes, this *would* make the book better, and yes, we *do* have time to add that, and yes, the fact that R comes before S *does* mean that Mr. Rosenberg has more votes than Mr. Scott. [Co-author's note to self: Get name legally changed to Scott Kendall before the next book comes out. *That'll* teach him.]

Doug and Kendall would like to thank Paul Becker and all the fine folks at Addison-Wesley (including Ross Venables, who's no longer there but who got this project off the ground) who somehow managed to compress the production schedule to compensate for the delays in the writing schedule (which are all Kendall's fault). We'd also like to thank the reviewers of the manuscript, especially Mark Woodbury, whose incisive comments about "defragmenting" the example gave us the push we needed to get it the point where we think it's really, really cool as opposed to just really cool. And, we'd like to thank Greg Wilson, who reviewed our first book for *Dr. Dobbs' Journal*, liked it, and suggested that we write a companion workbook. Specifically, he said: "The second criticism of this book is one that I thought I'd never make: It is simply too short. Having finally found a useful, readable, and practical description of a design-centered development methodology, I really wanted a dozen or more examples of each

point to work through. If the authors were to produce a companion workbook, I can promise them that they'd have at least one buyer."

Finally, Kendall would like to thank Doug for raising the art of snarkiness to a level that makes Kendall look like a paragon of good cheer in comparison to Doug.

Doug Rosenberg
Santa Monica, California
May 2001
dougr@iconixsw.com
http://www.iconixsw.com

Kendall Scott
Harrison, Tennessee
May 2001
kendall@usecasedriven.com
http://www.usecasedriven.com

Contents

Chapter 1

Introduction

The ICONIX process sits somewhere in between the very large Rational Unified Process (RUP) and the very small eXtreme programming approach (XP). The ICONIX process is use case driven, like the RUP, but without a lot of the overhead that the RUP brings to the table. It's also relatively small and tight, like XP, but it doesn't discard analysis and design like XP does. This process also makes streamlined use of the Unified Modeling Language (UML) while keeping a sharp focus on the traceability of requirements. And, the process stays true to Ivar Jacobson's original vision of what "use case driven" means, in that it results in concrete, specific, readily understandable use cases that a project team can actually use to drive the development effort.

The approach we follow takes the best of three methodologies that came into existence in the early 1990s. These methodologies were developed by the folks that now call themselves the "three amigos": Ivar Jacobson, Jim Rumbaugh, and Grady Booch. We use a subset of the UML, based on Doug's analysis of the three individual methodologies.

There's a quote in Chapter 32 of *The Unified Modeling Language User Guide*, written by the amigos, that says, "You can model 80 percent of most problems by using about 20 percent of the UML." However, nowhere in this book do the authors tell you which 20 percent that might be. Our subset of the UML focuses on the core set of notations that you'll need to do most of your modeling work. Within this workbook we also explain how you can use other elements of the UML and where to add them as needed.

One of our favorite quotes is, "The difference between theory and practice is that in theory, there is no difference between theory and practice, but in practice, there is." In practice, there never seems to be enough time to do modeling, analysis, and design. There's always pressure from management to jump to code, to start coding prematurely because progress on software projects tends to get measured by how much code exists. Our approach is a minimalist, streamlined approach that focuses on that area that lies in between use cases and code. Its emphasis is on what needs to happen at that point in the life cycle where you're starting out: you have a start on some use cases, and now you need to do a good analysis and design.

Our goal has been to identify a minimal yet sufficient subset of the UML (and of modeling in general) that seems generally to be necessary in order to do a good job on your software project. We've been refining our definition of "minimal yet sufficient" in this context for eight or nine years now. The approach we're telling you about in this workbook is one that has been used on hundreds of projects and has been proven to work reliably across a wide range of industries and project situations.

A Walk (Backwards) through the ICONIX Process

Figure 1-1 shows the key question that the ICONIX process aims to answer.

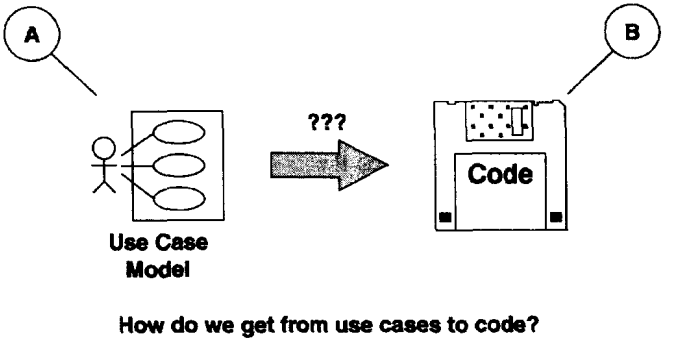


Figure 1-1: *Use Cases to Code*

What we're going to illustrate is how to get from point A to point B directly, in the shortest possible time. (Actually, we're not going to go all the way to code, but we'll take you close enough so you can taste it.) You can think of point A as representing this thought: "I have an idea of what my system has to do, and I have a start on some use cases," and point B as representing some completed, tested, debugged code that actually does what the use cases said it needed to do. In other words, the code implements the required behavior, as defined by the use cases. This book focuses on how we can get you from the fuzzy, nebulous area of "I think I want it to do something like this" to making those descriptions unambiguous, complete, and rigorous, so you can produce a good, solid architecture, a robust software design, then (by extension) nice clean code that actually implements the behavior that your users want.

We're going to work backwards from code and explain the steps to our goal. We'll explain why we think the set of steps we're going to teach is the minimal set of steps you need, yet is sufficient for most cases in closing the gap between use cases and code. Figure 1-2 shows the three assumptions we're going to make to start things off: that we've done some prototyping; that we have made some idea of what our user interface might look like; and that we might have some start in identifying the scenarios or use cases in our system.

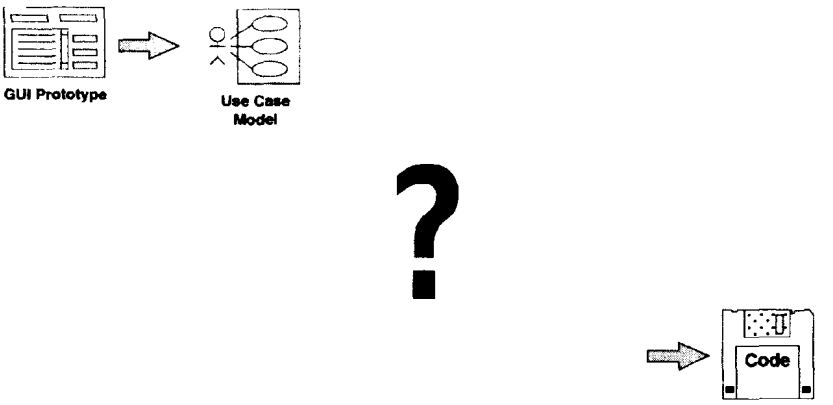


Figure 1-2: *Starting Off*

This puts us at the point where we're about to launch into analysis and design. What we want to find out is how we can get from this starting point to code. When we begin, there's only a big question mark—we have some nebulous, fuzzy ideas of what our system has to do, and we need to close this gap before we start coding.

In object-oriented systems, the structure of our code is defined by classes. So, before we write code, we'd like to know what our software classes are going to be. To do this, we need one or more class diagrams that show the classes in the system. On each of these classes, we need a complete set of attributes, which are the data members contained in the classes, and operations, which define what the software functions are. In other words, we need to have all our software functions identified, and we need to make sure we have the data those functions require to do their job.

We'll need to show how those classes encapsulate that data and those functions. We show how our classes are organized and how they relate to each other on class diagrams. We'll use the UML class diagram as the vehicle to display this information. Ultimately, what we want to get to is a set of very detailed design-level class diagrams. By **design-level**, we mean a level of detail where the class diagram is very much a template for the actual code of the system—it shows exactly how your code is going to be organized.

Figure 1-3 shows that class diagrams are the step before code, and there is a design-level diagram that maps one-to-one from classes on your diagram to classes in your source code. But there's still a gap. Instead of going from use cases to code, now we need to get from use cases to design-level class diagrams.

One of the hardest things to do in object-oriented software development is **behavior allocation**, which involves making decisions for every software function that you're going to build. For each function, you have to decide which class in your software design should be the class that contains it. We need to allocate all the behavior of the system—every software function needs to be allocated into the set of classes that we're designing.

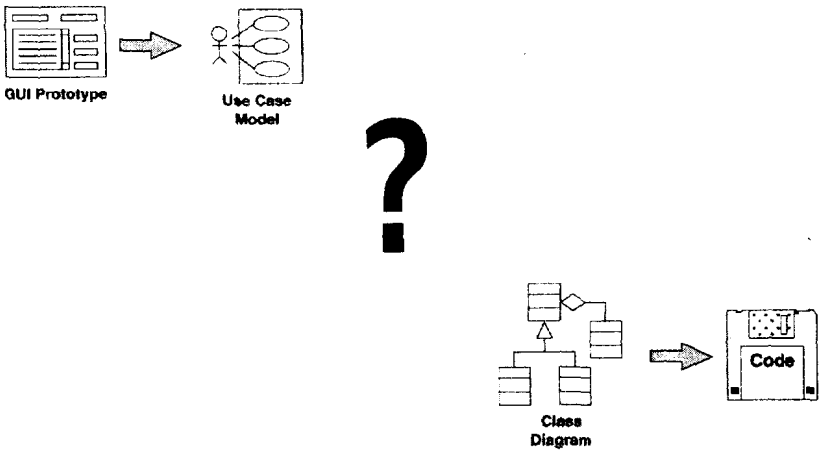


Figure 1-3: *Class Diagrams Map Out the Structure of the Code*

One UML diagram that's extremely useful in this area is the sequence diagram. This diagram is an ideal vehicle to help you make these behavior allocation decisions. Sequence diagrams are done on a per-scenario basis: for every scenario in our system, we'll draw a sequence diagram that shows us which object is responsible for which function in our code. The sequence diagram shows how runtime object instances communicate by passing messages. Each message invokes a software function on the object that receives the message. This is why it's an ideal diagram for visualizing behavior allocation.

Figure 1-4 shows that the gap between use cases and code is getting smaller as we continue to work backwards. Now, we need to get from use cases to sequence diagrams.

We'll make our decisions about allocating behavior to our classes as we draw the sequence diagrams. That's going to put the operations on the software classes. When you use a visual modeling tool such as Rational Rose or GDPro, as you draw the message arrows on the sequence diagrams, you're actually physically assigning operations to the classes on the class diagrams. The tool enforces the fact that behavior allocation happens from the sequence diagram. As you're drawing the sequence diagram, the classes on the class diagram get populated with operations.

So, the trick is to get from use cases to sequence diagrams. This is a non-trivial problem in most cases because the use cases present a requirements-level view of the system, and the sequence diagram is a very detailed design view. This is where our approach is different from the other approaches on the market today. Most approaches talk about use cases and sequence diagrams but don't address how to get across the gap between the fuzzy use cases and a code-like level of detail on the sequence diagrams. Getting across this gap between *what* and *how* is the central aspect of the ICONIX process.

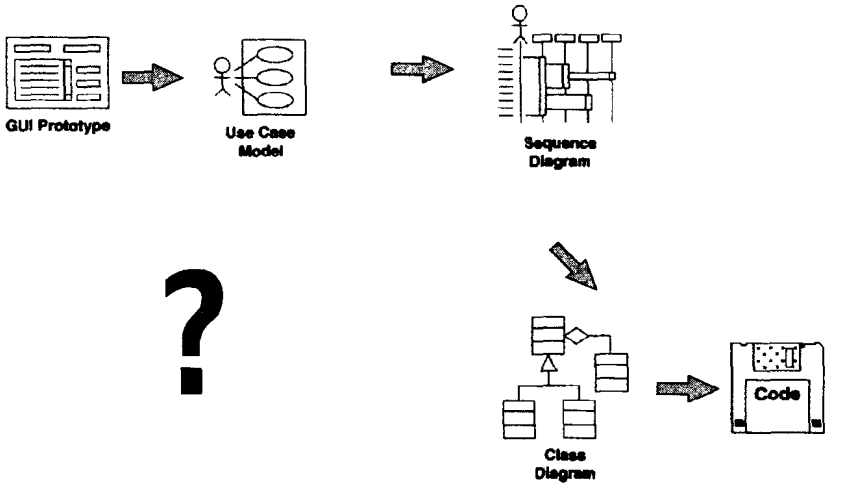


Figure 1-4: *Sequence Diagrams Help Us Allocate Operations (Behavior) to Classes*

What we're going to do now is close the gap between the fuzzy, nebulous use case and the very detailed and precise sequence diagram with another kind of diagram called a **robustness diagram**. The robustness diagram sits in the gap between requirements and detailed design; it will help make getting from the use cases to the sequence diagrams easier.

If you've been looking at UML literature, the robustness diagram was originally only partially included in the UML. It originated in Ivar Jacobson's work and got included in the UML standard as an appendage. This has to do with the history and the sequence of how Booch, Rumbaugh, and Jacobson got together and merged their methodologies, as opposed to the relative importance of the diagram in modeling.

Across the top of a sequence diagram is a set of objects that are going to be participating in a given scenario. One of the things we have to do before we can get to a sequence diagram is to have a first guess as to which objects will be participating in that scenario. It also helps if we have a guess as to what software functions we'll be performing in the scenario. While we do the sequence diagram, we'll be thinking about mapping the set of functions that will accomplish the desired behavior onto that set of objects that participate in the scenario.

It helps a great deal to have a good idea about the objects that we'll need and the functions that those objects will need to perform. When you do it the second time, it's a lot more accurate than when you take a first guess at it. The process that we're following, which is essentially Ivar Jacobson's process as described in his Objectory work, is a process that incorporates a first guess, or preliminary design, the results of which appear on what we call a robustness diagram. We refine that first guess into a detailed design on the sequence diagram. So, we'll do a sequence diagram for each scenario that we're going to build.

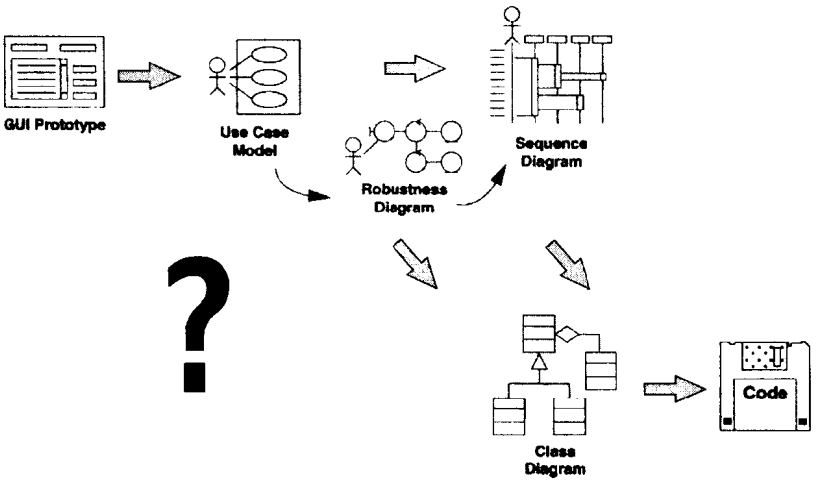


Figure 1-5: *Robustness Diagrams Close the Gap Between Requirements and Detailed Design*

Figure 1-5 shows that we're adding a diagram to our subset of UML. The robustness diagram was described in the original UML specs, but its definition was in an extra document called *Objectory Process-Specific Extensions*. What we've found over the past ten years is that it's very difficult to get from use cases to sequence diagrams without this technique. Using the robustness diagram helps avoid the common problem of project teams thrashing around with use cases and not really getting anywhere towards their software design. If you incorporate this step, it will make this process and your project much easier. We didn't invent robustness analysis, but we're trying to make sure it doesn't get forgotten. Robustness analysis has proven to be an invaluable aid in getting across the gap between requirements and design.

Robustness analysis sits right in the gap between what the system has to do and how it's actually going to accomplish this task. While we're crossing this gap, there are actually several different activities that are going on concurrently. First, we're going to be discovering objects that we forgot when we took our first guess at what objects we had in the system. We can also add the attributes onto our classes as we trace data flow on the robustness diagrams. Another important thing we'll do is update and refine the text of the use case as we work through this diagram.

We still have a question mark, though. That question mark relates to the comment we just made about discovering the objects that we forgot when we took our first guess. This implies that we're going to take a first guess at some point.

There's a magic phrase that we use to help teach people how to write use cases successfully: *Describe system usage in the context of the object model.* The first thing this means is that we're not talking, in this book, about writing fuzzy, abstract and vague, ambiguous use cases that don't have enough detail in them from which to produce a software design. We're going to teach you to write use cases that are very explicit, precise, and unambiguous. We have a very specific goal in mind when discussing use cases: we want to drive the software design from them. Many books on use cases take a different perspective, using use cases as more of