McGraw Hill Education 大学计算机教育国外著名教材系列（影印版）

PRACTICAL OBJECT-ORIENTED DESIGN WITH UML

SECOND EDITION

面向对象设计UML实践

（第2版）

Mark Priestley 著

McGraw Hill

清华大学出版社

大学计算机教育国外著名教材系列（影印版）

Practical Object-Oriented Design with UML

Second Edition

面向对象设计 UML 实践

（第 2 版）

Mark Priestley 著

清华大学出版社
北京

Mark Priestley
Practical Object-Oriented Design with UML, Second Edition
EISBN: 0-07-710393-9

图书在版编目(CIP)数据

面向对象设计 UML 实践(第 2 版)=Practical Object-Oriented Design with UML, Second Edition/(美)普里斯特(Priestley, M.)著. —影印本. —北京:清华大学出版社,2004.6
(大学计算机教育国外著名教材系列)
ISBN 7-302-08784-9

Ⅰ. 面…　Ⅱ. 普…　Ⅲ. 面向对象语言,UML—程序设计—高等学校—教材—英文　Ⅳ. TP312

中国版本图书馆 CIP 数据核字(2004)第 054012 号

出 版 者: 清华大学出版社		**地　　址**: 北京清华大学学研大厦	
http://www.tup.com.cn		**邮　　编**: 100084	
社 总 机: 010-62770175		**客户服务**: 010-62776969	

责任编辑: 龙啟铭
印 刷 者: 北京季蜂印刷有限公司
装 订 者: 三河市新茂装订有限公司
发 行 者: 新华书店总店北京发行所
开　　本: 185×230　**印张**: 23.5
版　　次: 2004 年 6 月第 1 版　2004 年 6 月第1 次印刷
书　　号: ISBN 7-302-08784-9/TP・6234
印　　数: 1~5000
定　　价: 39.00 元

出 版 说 明

进入 21 世纪，世界各国的经济、科技以及综合国力的竞争将更加激烈。竞争的中心无疑是对人才的竞争。谁拥有大量高素质的人才，谁就能在竞争中取得优势。高等教育，作为培养高素质人才的事业，必然受到高度重视。目前我国高等教育的教材更新较慢，为了加快教材的更新频率，教育部正在大力促进我国高校采用国外原版教材。

清华大学出版社从 1996 年开始，与国外著名出版公司合作，影印出版了“大学计算机教育丛书(影印版)”等一系列引进图书，受到国内读者的欢迎和支持。跨入 21 世纪，我们本着为我国高等教育教材建设服务的初衷，在已有的基础上，进一步扩大选题内容，改变图书开本尺寸，一如既往地请有关专家挑选适用于我国高校本科及研究生计算机教育的国外经典教材或著名教材，组成本套“大学计算机教育国外著名教材系列(影印版)”，以飨读者。深切期盼读者及时将使用本系列教材的效果和意见反馈给我们。更希望国内专家、教授积极向我们推荐国外计算机教育的优秀教材，以利我们把“大学计算机教育国外著名教材系列(影印版)”做得更好，更适合高校师生的需要。

清华大学出版社

影印版序

这本书很适合作为研究生或大学生面向对象方法课程的教材。从 2001 年到 2004 年，西北大学已有 4 届研究生使用了此书的第 1 版作为主要参考书。但第 1 版不足的是，它对软件开发过程和需求获取、分析的介绍显得过于单薄。第 2 版在这方面有了很大改进，主要是增加了第 3 章软件开发过程，并用餐馆预约系统取代了第 1 版的图编辑器作为开发案例。在软件开发过程一章中，通过对软件开发历史的回顾，不仅使读者对从瀑布模型到螺旋模型到迭代的增量开发有一个完整的认识，而且概括地介绍了目前在面向对象开发中影响很大的“统一(软件开发)过程”，使读者对“统一过程”有一个很好的了解。这些知识对于从事软件开发的人员是很必要的。餐馆预约系统案例的讲解是按照“统一过程”的核心工作流的思想安排的，在第 4～7 章分别讲述了如何用面向对象方法获取需求，分析、设计和实现这个系统。餐馆预约系统更贴近现实生活，更易于学生理解，用面向对象方法开发显得更自然，便于使读者更好地掌握面向对象方法的本质。

第 2 版在其他方面也有一些改动，主要是结合教学实践对学生容易产生误解的概念作了更清楚的说明，使全书的结构更合理。例如，第 1 版的“类图”改为“类和对象图”，“状态图”改为“状态图和活动图”，将“构件图”单设为一章，去掉了最后一章出租汽车调度系统。但整个内容没有大的变动。第 2 版仍然保留着第 1 版的特点，在内容上，根据当前软件工程的发展趋势和教学的特点，精心选择了最重要的基本内容，体现了“少而精”的写作思想。在概念的引入、阐述上深入浅出，在突出设计和代码的联系方面更有独到之处。

全书语言准确严谨，表达规范，简洁明快，很适合作为面向对象设计英语授课的教材。

卞雷

西北大学计算机系教授

PREFACE

Mr Palomar's rule had gradually altered: now he needed a great variety of models, perhaps interchangeable, in a combining process, in order to find the one that would best fit a reality that, for its own part, was always made of many different realities, in time and in space.

Italo Calvino

This book aims to provide a practical and accessible introduction to object-oriented design. It assumes that readers have prior knowledge of an object-oriented programming language, ideally Java, and explains both the principles and application of UML. It is aimed principally at students in the later years of an undergraduate or Masters course in Computer Science or Software Engineering, although I hope that other audiences will find the book useful.

The overall strategy of the book is to emphasize the connections between design notations and code. There are many excellent books available which discuss systems analysis and design with UML, but fewer that pay detailed attention to the final product, the code of the system being developed. UML is at bottom a language for expressing the designs of object-oriented programs, however, and it seems natural to consider the notation and meaning of the language from this perspective. I have also, over the years, found this to be a good way of imparting to students a concrete sense of what the design notations actually mean.

The book has two main objectives related to this overall philosophy. The first is to provide a complete example of an object-oriented development using UML, starting with a statement of requirements and finishing with complete executable code which can be run, modified and extended.

This objective of course places limits on the size of the example that can be considered. To get round this, the book takes as its paradigm system architecture a typical stand-alone desktop application, supporting a graphical user interface and interfacing to a relational database. Within this framework, the text examines the development of some core functionality, and leaves extensions of the system to be worked out as exercises.

The second objective is to provide a tutorial introduction to those aspects of UML that are important in developing this kind of application. Much emphasis is placed on clearly explaining the constructs and notation of the design language, and demonstrating the close relationship between the design and the implementation of object-oriented programs. These issues are treated rather superficially in many books. If they are not clearly understood, however, it is difficult to make correct use of UML.

UML is a large and complex language, and when one is learning it there is a danger of being overwhelmed by details of the notation. In order to avoid this, the book uses a subset of UML that is sufficient to develop desktop applications. The most significant omissions are any coverage of concurrency, activity diagrams and anything other than a brief mention of deployment diagrams. These aspects of the language are obviously important for 'industrial-strength' applications of UML, but these lie somewhat outside the experience of the intended audience of this book.

STRUCTURE OF THE BOOK

Following an introductory chapter, Chapter 2 introduces the basic concepts of object modelling in the context of a simple programming example. Chapters 3 to 7 contain a more extended case study of the use of UML, while Chapters 8 to 12 present the most important UML notations systematically. These two sections are independent of each other, allowing different reading strategies as shown in Figure P.1. Chapter 13 discusses strategies for implementing UML designs and Chapter 14 provides a general discussion of some of the underlying principles of object-oriented design.

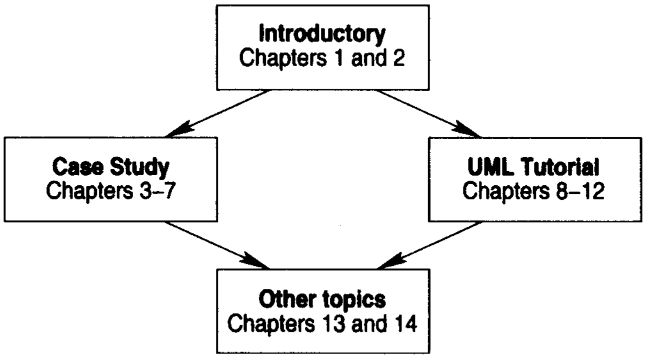

Figure P.1 Chapter dependencies

CHANGES IN THE SECOND EDITION

The most significant change in the second edition is that the diagram editor example has been replaced by a new case study based on a simple booking system for a restaurant. This provides an application with more 'real-world' context than the diagram editor, and is one which many students find easier to relate to. It also allows concepts of different architectural layers to be introduced more naturally than before, and these topics are now covered explicitly in Chapters 4 to 7.

Although the focus of the book is on language rather than process, it is impossible to divorce the two entirely in any practical approach. In the new Chapter 3, the book now includes an explicit discussion of some issues in the development of software processes and provides an outline sketch of the Unified Process.

The remaining chapters are very much the same as in the previous edition, though minor changes have been made throughout the book, both in content and presentation. To make room for the new chapter and case study, some material has had to be omitted from this edition, most noticeably the second case study. All the material that has been omitted, including the diagram editor example, will be made available from book's website.

FURTHER RESOURCES

A web page for the book provides access to the source code for the examples used in the book, solutions to all exercises and material from the first edition. It can be found at the following URL:

```
http://www.mcgraw-hill.co.uk/textbooks/priestley
```

An instructor's manual, including slides, the diagrams used in the book and additional exercises, is available to *bona fide* academics who have adopted the book for classroom use. Information on how to obtain the manual can be found on the publisher's website.

ACKNOWLEDGEMENTS

In the preparation of this new edition, my most significant debt is to my students who have made use of the earlier editions, and also sat through earlier presentations of the restaurant booking system. I am indebted to Michael Richards for the initial idea for this case study.

CONTENTS

1

INTRODUCTION TO UML

According to its designers, UML, the Unified Modeling Language, is 'a general-purpose visual modeling language that is used to specify, visualize, construct and document the architecture of a software system'. This chapter explains how models are used in the software development process, and the role of a language such as UML. The high-level structure of UML is described, together with an informal account of its semantics and the relationship between design notations and code.

1.1 MODELS AND MODELLING

The use of models in the development of software is extremely widespread. This section explains two characteristic uses of models, to describe real-world applications and also the software systems that implement them, and then discusses the relationships between these two types of model.

Models of software

Software is often developed in the following manner. Once it has been determined that a new system is to be built, an informal description is written stating what the software should do. This description, sometimes known as a *requirements specification*, is often prepared in consultation with the future users of the system, and can serve as the basis for a formal contract between the user and the supplier of the software.

The completed requirements specification is then passed to the programmer or project team responsible for writing the software; they go away and in relative isolation produce a program based on the specification. With luck, the resulting program will be produced on time, within budget, and will satisfy the needs of the people for whom the original proposal was produced, but in many cases, sadly, this does not happen.

The failure of many software projects has led people to study the way in which software is developed, in an attempt to understand why projects fail. As a result, many suggestions have been made about how to improve the software development process. These often take the form of *process models* describing a number of activities involved in development and the order in which they should be carried out.

Process models can be depicted graphically. For example, Figure 1.1 shows a very simple process, where code is written directly from the system requirements with no intervening steps. As well as showing the processes involved, as rectangles with rounded corners, this diagram shows the artefacts produced at each stage in the process. When two stages in a process are carried out in sequence, the output of one stage is often used as the input to the next, as indicated by the dashed arrows.

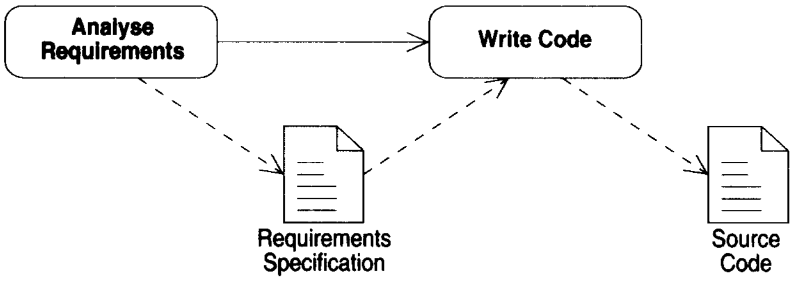

Figure 1.1 A primitive model of software development

The requirements specification produced at the start of a development can take many forms. A written specification may be either a very informal outline of the required system or a highly detailed and structured functional description. In small developments the initial system description may not even be written down, but only exist as the programmer's informal understanding of what is required. In yet other cases a prototype system may have been developed in conjunction with the future users, and this could then form the basis of subsequent development work. In the discussion above all these possibilities are included in the general term 'requirements specification', but this should not be taken to imply that only a written document can serve as a starting point for subsequent development.

It should also be noted that Figure 1.1 does not depict the whole of the software life cycle. In this book, the term 'software development' is used in rather a narrow sense, to cover only the design and implementation of a software system, and many other important components of the life cycle are ignored. A complete project plan would also cater for crucial activities such as project management, requirements analysis, quality assurance and maintenance.

When a small and simple program is being written by a single programmer, there is little need to structure the development process any more than in Figure 1.1. Experienced programmers can keep the data and subroutine structures of such a program clear in their minds while writing it, and if the behaviour of the program is not what is expected they can make any necessary changes directly to the code. In certain situations this is an entirely appropriate way of working.

With larger programs, however, and particularly if more than one person is involved in the development, it is usually necessary to introduce more structure into the process. Software development is no longer treated as a single unstructured activity, but is instead broken up into a number of subtasks, each of which usually involves the production of some intermediate piece of documentation.

Figure 1.2 illustrates a software development process which is slightly more complex than the one shown in Figure 1.1. In this case, the programmer is no longer writing code based on the requirements specification alone, but first of all produces a structure chart showing how the overall functionality of the program is split into a number of modules, or subroutines, and illustrating the calling relationship between these subroutines.

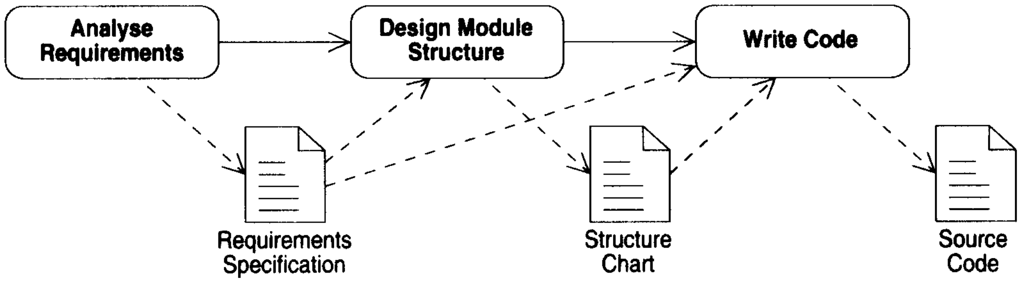

Figure 1.2 A more complex software development process

This process model shows that the structure chart is based on the information contained in the requirements specification, and that both the specification and the structure chart are used when writing the final code. The programmer might use the structure chart to clarify the overall architecture of the program, and refer to the specification when coding individual subroutines to check up on specific details of the required functionality.

The intermediate descriptions or documents that are produced in the course of developing a piece of software are known as *models*. The structure chart mentioned in Figure 1.2 is an example of a model in this sense. A model gives an abstract view of a system, highlighting certain important aspects of its design, such as the subroutines and their relationships, and ignoring large amounts of low-level detail, such as the coding of each routine. As a result, models are much easier to understand than the complete code of the system and are often used to illustrate aspects of a system's overall structure or architecture. An example of the kind of structure that is meant is provided by the relationships between subroutines documented in the structure chart above.

As larger and more complex systems are developed and as the number of people involved in the development team increases, more formality needs to be introduced into the process. One aspect of this increased complexity is that a wider range of models is used in the course of a development. Indeed, software design has sometimes been defined as the construction of a series of models describing important aspects of the system in more and more detail, until sufficient understanding of the requirements is gained to enable coding to begin.