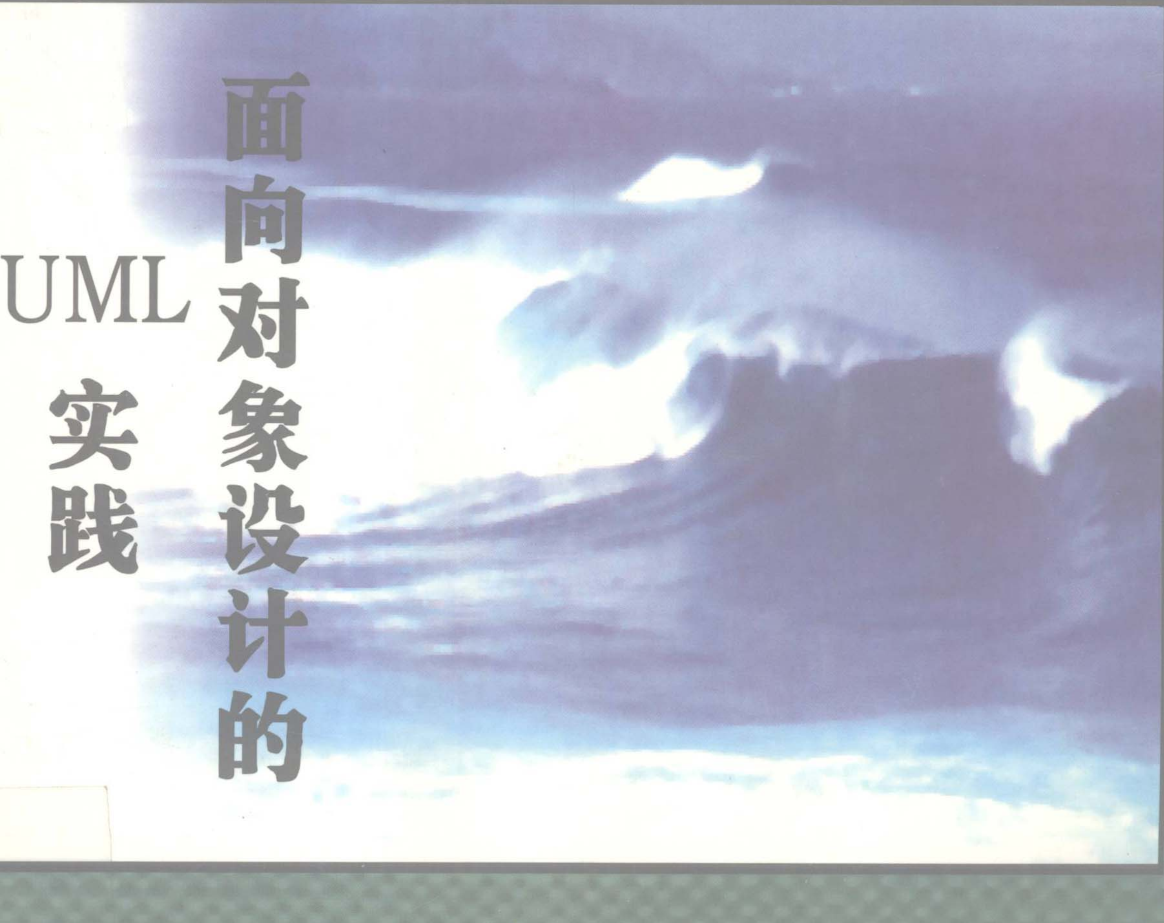
大学计算机教育丛书（影印版）

Practical Object-Oriented Design with UML

Mark Priestley



面向对象设计的 UML 实践

清华大学出版社
http://www.tup.tsinghua.edu.cn

McGraw-Hill
http://www.mhhe.com

Practical Object-Oriented Design with UML

面向对象设计的 UML 实践

Mark Priestley

University of Westminster

清 华 大 学 出 版 社
McGraw-Hill Companies, Inc.

(京)新登字 158 号

图书在版编目(CIP)数据

面向对象设计的 UML 实践: 英文/(美)普里斯特(Priestley, M.)编著. –影印本. –北京: 清华大学出版社,2000.11
(大学计算机教育丛书)
ISBN 7-302-04098-2

Ⅰ. 面… Ⅱ. 普… 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(2000)第 75978 号

出版者: 清华大学出版社(北京清华大学学研大厦,邮编 100084)
http://www.tup.tsinghua.edu.cn
印刷者: 清华大学印刷厂
发行者: 新华书店总店北京发行所
开 本: 787×960 1/16 印张: 22.25
版 次: 2000 年 12 月第 1 版 2001 年 3 月第 2 次印刷
书 号: ISBN 7-302-04098-2/TP·2415
印 数: 3001~6000
定 价: 39.00 元

出版者的话

今天，我们的大学生、研究生和教学、科研工作者，面临的是一个国际化的信息时代。他们将需要随时查阅大量的外文资料；会有更多的机会参加国际性学术交流活动；接待外国学者；走上国际会议的讲坛。作为科技工作者，他们不仅应有与国外同行进行口头和书面交流的能力，更为重要的是，他们必须具备极强的查阅外文资料获取信息的能力。有鉴于此，在原国家教委所颁布的“大学英语教学大纲”中有一条规定：专业阅读应作为必修课程开设。同时，在大纲中还规定了这门课程的学时和教学要求。有些高校除开设“专业阅读”课之外，还在某些专业课拟进行英语授课。但教、学双方都苦于没有一定数量的合适的英文原版教材作为教学参考书。为满足这方面的需要，我们陆续精选了一批国外计算机科学方面最新版本的著名教材，进行影印出版。我社获得国外著名出版公司和原著作者的授权将国际先进水平的教材引入我国高等学校，为师生们提供了教学用书，相信会对高校教材改革产生积极的影响。

我们欢迎高校师生将使用影印版教材的效果、意见反馈给我们，更欢迎国内专家、教授积极向我社推荐国外优秀计算机教育教材，以利我们将《大学计算机教育丛书（影印版）》做得更好，更适合高校师生的需要。

清华大学出版社
《大学计算机教育丛书（影印版）》项目组
1999.6

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 is a revised edition of my earlier book *Practical Object-Oriented Design*. It shares the same aims as the earlier book, namely to provide a practical introduction to design which will be of use to people with experience of programming who want to learn how to express the design of object-oriented programs more abstractly.

The most significant change from the earlier book is that the notation used is now that of UML, the Unified Modeling Language. UML is to a large extent an evolutionary development of the OMT language used in the earlier book, so this change has not necessitated great changes in the structure and content of the book.

As with the earlier book, 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. Unlike questions of methodology and process, these issues are treated rather superficially in many books. If they are not clearly understood, however, it is difficult to make meaningful use of a notation such as UML.

In addition, the book addresses a number of pragmatic issues which are often omitted from design books, such as the integration of a design with an existing framework, the use of patterns in design, dealing with persistent data, and the physical design of object-oriented programs.

UML is a much larger and more complex language than OMT, and when 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 express all the significant features of medium-sized object-oriented programs. The most significant omissions are any coverage of concurrency, activity diagrams, and anything other than a brief mention of component and 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.

The Java language is used for programming examples. In order to ensure maximum portability, the use of the language has been kept as straightforward as possible and the Java 1.0 event model is used in preference to later, more complex, models. For the benefit of readers who prefer to use C++, the book's web site will provide alternative versions of those sections of the book which are specific to the Java language.

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 5 contain a more extended example of the use of UML in designing a diagram editor application, while chapters 6 to 8 present the most important UML notations systematically.

Chapters 2 to 5 introduce many features of UML in the context of extended examples before a more systematic presentation of the language is given in chapters 6 to 8. However, the diagram editor case study in chapters 3 to 5 is not referred to in the text of chapters 6 to 8, so readers who prefer to cover the language systematically before looking at its use can move directly from chapter 2 to chapter 6, returning later to chapters 3 to 5.

Chapters 9 to 13 are more or less independent of each other and can be read in any order. Chapter 9 covers the use of constraints with UML, and the OCL language. Chapter 10 presents systematic techniques for the implementation of designs, building on some basic material presented in Chapter 5. Chapter 11 covers some miscellaneous issues including the relationship between logical and physical design, and the impact of non-functional requirements on a design, and Chapter 12 discusses some important principles of object-oriented design, and the popular area of design patterns. A case study is presented in Chapter 13, and it is planned that further case studies will be available from the book's web site.

FURTHER RESOURCES

A web page for this book has been set up, providing access to the source code for the case studies used in the book, solutions to selected exercises, additional case studies and other related material. It can be found at the following URL:

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

A instructor's manual, including suggested solutions to all exercises, is available to *bona fide* academics. Information on how to obtain the manual can be found on the publisher's web site.

ACKNOWLEDGEMENTS

In the preparation of this book, my most significant debt is to the readers of the earlier book who have taken the trouble to communicate to me their opinions. I have much appreciated this steady trickle of, on the whole, positive feedback. It is in the nature of publishing that the only concrete way I have of expressing my gratitude is to hope that you will all immediately buy this new book, and find it just as useful.

The enthusiasm of Elizabeth Robinson for this new edition was instrumental in motivating me to complete the book, and I would also like to thank the anonymous reviewers of the manuscript for their helpful comments. Thanks also to Alison, for providing some breaks that neither of us really needed, and a big thank you to Rosemary, Felix and Alasdair.

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 programs

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 this is sadly not the case.

In an attempt to address some of these problems, much effort has been devoted to analysing the process by which software is developed, and many methods have been proposed suggesting how it could be done better. Processes can be illustrated graphically; for example, the diagram in Figure 1.1 depicts the rudimentary process outlined in the previous paragraph.

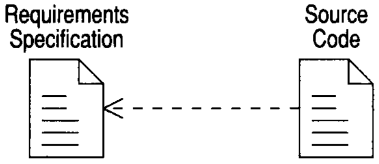


Figure 1.1 A primitive model of software development

In this diagram the icons represent the two different documents involved in the development of the system, namely the original specification and the source code itself. The dashed arrow states that the code depends on the requirements specification in the sense that the functionality provided by the system should be that specified in the requirements. In many real developments the situation is less clear-cut than this: it is a common experience to find that writing code or seeing a prototype system running changes one's view of what a proposed system should do.

The description of the desired system from which a development process starts can take many forms. Very often a written specification forms the starting point of the development. Such a specification might be either a very informal outline of the required system, or a highly detailed and structured functional specification. In small developments the initial system description might 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 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 aspects of software engineering 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 has been done above. 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. The programmer is no longer writing code based on the requirements specification alone, but has first of all produced a structure chart showing how the overall functionality of the program is split into a number of subroutines, and illustrating the calling relationship between the subroutines.

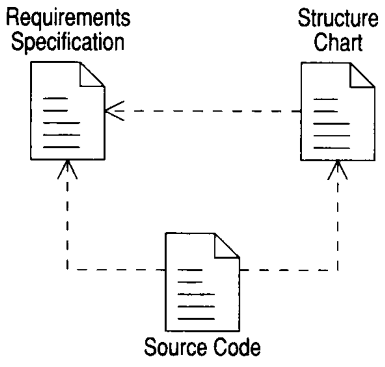


Figure 1.2 A more complex software development process

Figure 1.2 shows that the structure chart depends on the information contained in the requirements specification, and both the specification and the structure chart are used to write the final code. The programmer might be using the structure chart to clarify the overall architecture of the program, and referring 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 and ignoring large amounts of low-level detail. 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 subroutine calling structure 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 could almost be 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.

The use of models is therefore central to software design, and provides two important benefits which help to deal with the complexity involved in developing almost any significant piece of software. Firstly, models provide succinct descriptions of important aspects of a system that may be too complex to be grasped as a whole. Secondly, models provide a valuable means of communication, both between different members of the development team, and also between the team and outsiders such as the client. This book describes the models that are used in object-oriented design and gives illustrations of their use.

Models of applications

Models are also used in software development to help in understanding the application area being addressed by a system, before the stages of system design and coding are reached. Such models are sometimes referred to as *analysis models* as opposed to the *design models* discussed above. The two types of model can be differentiated by the fact that unlike design models, analysis models do not make any explicit reference to the proposed software system or its design, but aim instead to capture certain aspects and properties of the 'real world'.

In general terms, analysis and design models fulfil the same needs and provide the same sorts of benefit. Both software systems and the real-world systems that they are supporting or interacting with tend to be highly complex and very detailed. In order to manage this complexity, descriptions of systems need to emphasize structure rather than detail, and to provide an abstract view of the system. The exact nature of this abstract view will depend on the purposes for which it is produced, and in general several such views, or models, will be needed to give an adequate overall view of a system.

Characteristically, analysis models describe the data handled in an application and the various processes by which it is manipulated. In traditional analysis methods, these models are expressed using diagrams such as logical data models and data flow diagrams. It is worth noticing that the use of analysis models to describe business processes predates and is independent of the computerization of such processes. For example, organization charts and diagrams illustrating particular production processes have been used for a long time in commerce and industry.

Relationship between analysis and design models

It is likely that both analysis and design models, as defined above, will be produced in the course of the development of any significant software system. This raises the question of what relationship exists between them.

The process of system development has traditionally been divided into a number of phases. An analysis phase, culminating in the production of a set of analysis models, is followed by a design phase, which leads to the production of a set of design models. In this scenario, the analysis models are intended to form the input to the design phase, which has the task of creating structures which will support the properties and requirements stated in the analysis models.

One problem with this division of labour is that very different types of language and notation have often been used for the production of analysis and design models. This leads to a process of translation when moving from one phase to the next. Information contained in the analysis models must be reformulated in the notation required for the design models.

Clearly, there is a danger that this process will be both error-prone and wasteful. Why, it has been asked, go to the trouble of creating analysis models if they are going to be replaced by design models for the remainder of the development process? Also, given that notational differences exist between the two types of model, it can be difficult to be certain that all the information contained in an analysis model has been accurately extracted and represented in the design notation.

One promise of object-oriented technology has been to remove these problems by using the same kinds of model and modelling concepts for both analysis and design. In principle, the idea is that this will remove any sharp distinction between analysis and design models. Clearly, design models will contain low-level details that are not present in analysis models, but the hope is that the basic structure of the analysis model will be preserved and be directly recognizable in the design model. Apart from anything else, this might be expected to remove the problems associated with the transfer between analysis and design notations.

A consequence of using the same modelling concepts for analysis and design is to blur the distinction between these two phases. The original motivation behind this move was the hope that software development could be treated as a seamless process: analysis would identify relevant objects in the real-world system, and these would be directly represented in software. In this view, design is basically a question of adding specific implementation details to the underlying analysis model, which would be preserved unchanged throughout the development process. The plausibility of this view will be considered in more detail in Section 2.10 once the object model has been discussed in more detail.

The purpose of this book is to explain the modelling concepts used by object-oriented methods, and to show how models can be expressed in the notation defined by UML. The focus of the book is on design and the use of design models in the development of software, but the same modelling concepts apply equally well to the production of analysis models. Analysis is a skill distinct from design, and there is much to learn about the techniques for carrying it out effectively, but the resulting analysis models can be perfectly well expressed using the notation presented in this book.

1.2 METHODOLOGIES

Software development, then, is not simply a case of sitting down at a terminal and typing in the program code. In most cases the complexity of the problem to be solved requires that intermediate development steps are taken, and that a number of abstract models of the program structure are produced. These points apply equally well to developments involving only one programmer and to conventional team developments.