大学计算机教育丛书（影印版）

Use Case Maps for Object-Oriented Systems

用于面向对象系统开发的使用实例图

R. J. A. Buhr
R. S. Casselman

清华大学出版社 · PRENTICE HALL

Use Case Maps for Object-Oriented Systems

用于 面向对象系统开发的 使用实例图

R.J.A. Buhr
R.S. Casselman

清 华 大 学 出 版 社
Prentice-Hall International, Inc.

（京）新登字 158 号

D026/38 10

Use case maps for Object-Oriented Systems/R. J. A. Buhr, R. S. Casselman
©1996 by Prentice Hall, Inc.
Original edition published by Prentice Hall, Inc., a Simon & Schuster Company.
Prentice Hall 公司授权清华大学出版社在中国境内(不包括中国香港特别行政区、澳门地区和台湾地区)独家出版发行本书影印本。
本书任何部分之内容,未经出版者书面同意,不得用任何方式抄袭、节录或翻印。
本书封面贴有 Prentice Hall 激光防伪标签,无标签者不得销售。

北京市版权局著作权合同登记号：01-98-0075

图书在版编目(CIP)数据

用于面向对象系统开发的使用实例图：英文/(美)布赫(Buhr,R. J. A.),(美)卡斯尔曼(Casselman,R. S.)著. —影印版. —北京:清华大学出版社,1998.1
(大学计算机教育丛书)
ISBN 7-302-02796-X

Ⅰ. 用…　Ⅱ. ①布…②卡…　Ⅲ. 面向对象语言-程序设计　Ⅳ. TP312

中国版本图书馆 CIP 数据核字(98)第 02516 号

出版者：清华大学出版社(北京清华大学校内,邮编 100084)
http://www.tup.tsinghua.edu.cn
印刷者：清华大学印刷厂
发行者：新华书店总店北京科技发行所
开　本：850×1168 1/32　印张：10.25
版　次：1998 年 2 月第 1 版　1998 年 8 月第 3 次印刷
书　号：ISBN 7-302-02796-X/TP・1460
印　数：7001～12000
定　价：20.00 元

出版前言

我们的大学生、研究生毕业后，面临的将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，我们挑选了 7 本计算机科学方面最新版本的教材，进行影印出版。首批影印出版的 6 本书受到广大读者的热情欢迎，我们深受鼓舞，今后还将陆续推出新书。希望读者继续给予大力支持。Prentice Hall 公司和清华大学出版社这次合作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。

清华大学出版社

Prentice Hall 公司

1997.11

Foreword

*R*ay Buhr is one of the few people who has been able to combine sound software engineering practices with new theories and approaches. Ray has a credo: There is nothing as practical as a good theory. Whenever Ray presents something new we have learned to pay close attention, because his contributions to our discipline have, over the years, been adopted not only by the academic community but most importantly by practitioners of software development.

Now Ray Buhr together with Ron Casselman have given us a new contribution: use case maps. The authors have noticed how excellent software engineers have been working for years and observed how they intuitively reasoned about the behavior of systems and systems components. They have identified the need for a semi-formal tool (not mathematical) to express use case behavior at different levels of abstraction. This tool captures the way these superb designers work and makes that available to the rest of us.

Use cases have been accepted by many practitioners of modern software development. Requirements are statically expressed in a use case model in terms of actor and use case objects. The behavior of this model is described either in prose or by using interaction diagrams and state transition diagrams. Then each use case is mapped onto an object model. An object model is a static model of the design of the system. The mapping is described again in terms of interaction diagrams and state transition diagrams. This is basically what the development process looks like for those methods that have adopted use case thinking.

Use case maps fill a very important need. They fill the gap between verbal descriptions and detailed descriptions in terms of interaction diagrams. Interaction diagrams focus on the interaction between components, let these be actors, subsystems, objects, or the like. The responsibilities of the components are verbal annotations to the interaction diagrams. Use case maps allow us to reason about the responsibilities of the components without going into details about the messaging between the components. Using use case maps frees the designers from having to hold their own mental models of the dynamic behavior when they do detailed design. Instead the designers can focus their thinking on other higher level issues, such as the alternative object structures to realize the use cases. Therefore use case maps will help us to iterate over the use case model and the object models until we have identified the most feasible system. We will be able to build more well-structured systems—more robust and reusable systems.

Use case maps are one of the most important contributions to our understanding of use cases. I highly recommend this book to people who already have learned the basics of use cases from my own books or papers and who want to take a step further. I also recommend this book to those people who have learned about basic object-orientation from, for instance, Grady Booch, Derek Coleman, Brian Henderson-Sellers, Steve Mellor, Jim Odell, Jim Rumbugh, and Rebecca Wirfs-Brock. This is a more advanced book and reading it will provide you with new insights in the very difficult discipline of developing software.

Ivar Jacobson

Preface

Our justification for adding yet another book to an overcrowded field is that it offers something new: *use case maps*. In this preface we try to give you a quick glimpse of why you should be interested in them.

Use cases [18] are structured prose descriptions of interaction scenarios between a system to be designed and users of the system. Use cases explain preconditions, postconditions, and the scenario itself (or possibly a set of scenarios that are so closely related that they can be better described as a theme with variations). The presumption is that a well chosen set of critical scenarios is a good starting point for design and, ultimately, for testing the implementation. Use case maps provide a visual notation for use cases and also a means of extending them into high-level design. However, understanding use case maps does not depend on familiarity with use cases. They are a new abstraction in their own right. To understand the nature of this abstraction, why it is new and why we think it is useful, we need to fill in some background.

Ever increasing demands on software for more of everything—functionality, flexibility, reusability, extensibility, performance, robustness, distributed operation—seem to ensure that software complexity keeps pushing the limits of both tools and human intellect. Tools—new languages, compilers, operating systems, application packages, CASE tools—provide more productive capacity (or try to) but the problems always seem to outpace them. This historical problem with software is showing no signs of going away.

Systems—sets of collaborating components that jointly achieve some overall purpose—are at the heart of all modern applications of computers, as the following examples indicate. Object-oriented programs in execution, even considered purely in software terms, are systems (of collaborating objects). They are also components of larger systems: the application environments in which they run. Computer communication networks are systems that are part software, part physical. Applications distributed over such networks are systems that are also part software, part physical, for example, they contain collaborating processes, objects, clients, and servers, to name but a few of many possibilities. Computers embedded as components in automobiles, aircraft, nuclear power plants, and medical instruments are systems. Because, by definition, the operation of a system is decentralized among its collaborating components, the purposeful behaviour of a system as a whole is difficult to visualize. This is particularly true with software, for a number of reasons. Software is nonphysical, the code we see in its source files may express the components of the running system only in a somewhat indirect fashion, and the conceptual world of programming languages is typically weak on system concepts (for example, the same object oriented programs that we said above are systems, do not look like systems when you read the code, just sets of class definitions).

Use case maps provide a notation to aid humans in expressing and reasoning about large-grained behaviour patterns in systems.

One of the most difficult problems with systems is understanding and expressing the large-grained behaviour patterns that will be jointly achieved by the components of a system while the system is running. To understand this term, think of a stimulus like a mouse click or an interrupt from a communications device that triggers some chain of causally related responsibilities performed in software (or in a mix of software and intervening hardware in a distributed system). One way of looking at this is from outside the system, for example, a mouse click causes a file icon to open on a screen (the system as a whole has the responsibility of making this happen). Another way to view this is from inside, for example, the responsibilities of handling the click are decentralized in a set of objects in a running program and the click propagates through this set by means of a causally connected sequence of interobject collaborations. This causes objects along the way to perform responsibilities in relation to the click. This ultimately results in one of the objects causing the file icon to open on the screen. Either way, the chain of causally related responsibilities is a "large-grained behaviour pattern".

Use case maps provide notations for indicating intended coupling between large-grained behaviour patterns.

An important design issue with large-grained behaviour patterns is that several may be in progress through a system at the same time, and may be coupled to each other. Patterns that are independent may simply be interleaved, but this is not what we mean by "coupled". Coupling may be intended or not. An example of an intended coupling is the following: a pattern triggered by a mouse click requests data that will, in the normal course of events, come from a physically remote part of the system through some large-grained behaviour pattern initiated there; the original mouse-initiated pattern must wait part way through for the remotely initiated one. On the other hand, interpattern coupling may be unintended, and cause errors. In other words there may be conflicts between patterns.

Use case maps bridge a modeling gap between requirements and design.

Large-grained behaviour patterns seem to belong to *both* requirements and high-level design. Expressing them only with prose use cases at the requirements level leaves a big gap between requirements and design. Expressing them during design without use case maps requires making commitments to realization details. Use case maps fill a gap in the suite of design models by providing a way of representing large-grained behaviour patterns as first-class abstractions above the level of realization details. It is true that people design systems successfully without use case maps by holding the patterns in their minds, but the mental models are often lost afterwards. There is a chicken-and-egg problem: The patterns won't happen in the actual system until all the details are resolved, but designers need to think about the patterns in a high-level way to make high-level decisions about how to realize them, before the details are resolved.

Use case maps offer something new in relation to architecture. They provide a behavioural framework for making architectural decisions at a high level of design, and also for characterizing behaviour at the architectural level once the architecture is decided.

The high-level structural form of a system, above the level of the details of its components, is often called its architecture. Architecture is hard to define in the abstract, but companies that make products with software in them have a view of what it is. Architecture is defined by answers to high-level questions such as the following: How many components of what types should there be in the running system? How should components be clustered into large-grained units like layers or peer subsystems? What types of structures should connect components into collaborating teams to handle higher-level responsibilities, for example, structures like pipelines, rings, or networks? What additional structures are needed to monitor for failures and to recover from them? Should component structures be fixed or dynamic? How should responsibilities be allocated among the components of the structures? Which structures are likely to give the best performance along critical paths through them, or be the most robust in the presence of failures? Which structures are likely to give the best flexibility, reusability and extensibility? What structures are needed to build families of products that may use different mixes of components? And so forth. Such questions are at the same level of abstraction as large-grained behaviour patterns and need to be related to them. However, while we have had adequate means in the past for representing the structures of architecture, there has been a missing link up till now: a good way of expressing the large-grained behaviour patterns of architecture.

Use case maps provide a new technique for capturing large-grained behaviour patterns as concrete work products that may be saved, manipulated, extended, and reused to guide implementation, maintenance, and evolution.

People making high-level decisions need to think in a high-level way. People making detailed decisions to implement high-level ones need to understand the high-level thinking in order to get the details right. People making changes to details for maintenance or evolution purposes need to understand the high level-thinking in order not to make changes that inadvertently damage the big picture. Capturing the high-level thinking in concrete form helps with all of these things.

Use case maps bring real time and object-oriented issues together under a common conceptual umbrella.

Object-oriented programming is important as a technique for improving reusability and extensibility of software. As explained further in Chapter 1, properties of systems such as concurrency and robust operation in the presence of failure are often suggested by using the term *real time* as a modifier of the term *system*. The ever increasing demands on software for more of everything identified above are creating a need for a broader view of object oriented programs as systems in their own right and as components of real time systems. A problem is that this broader view is hard to see at the level of object-oriented programming languages. The combination of real time issues, object oriented issues, and more-of-everything issues creates interesting challenges that have motivated this book.

Use case maps add to the repertoire of patterns available to the designer.

Patterns are currently the focus of much interest in the object oriented community. Use case maps provide a new kind of pattern that adds to the repertoire of patterns available for both object-oriented and real time applications.

CONTENT OF THE BOOK

This book stands alone as a comprehensive text on use case maps and their applications to high-level design of systems. It also shows how to use the maps in a coordinated way with other standard requirements/design models for object-oriented and real time systems (for example, prose use cases, class relationship diagrams, collaboration graphs, interaction sequence diagrams, and so on). However, readers should not look to this book to provide a comprehensive, step-by-step, life-cycle method that covers all aspects of design and development, or to provide a tutorial on basic object-oriented and real time concepts.

ASSUMED PREREQUISITES

Prescribing prerequisites for this book is difficult because it covers so much ground at a high level of abstraction.

A blanket set of prerequisites for reading it end to end and proceeding directly to applying it across the range of object-oriented and real time implementation techniques would be the following: general knowledge of the basic issues and principles of object-oriented programming (with classes and objects) and real time programming (with interrupts and concurrent processes), with some implementation experience in both areas. This preparation is necessary because this is—deliberately—a relatively short book and it would take a very long one to tell readers without this background how to fill in all the details for the wide range of implementation technologies that might be used for these kinds of systems.

However, the principles of the book are accessible to people with less background than this. Much of the book is new material introduced from first principles in a tutorial fashion. As such, much of it can be read and understood at an overview level by relative novices (for example, second or third year undergraduate university students in computer science or engineering). At the end of this preface is a guided tour of the book for readers with different backgrounds. Readers with only object-oriented background or only real time systems background can both read a large fraction of the book without additional preparation.

Although implementation experience would be required to translate this book's ideas into practice, and some programming examples are presented in the form of fragments of code in C++ and Smalltalk for those who want a sense of the path to implementation, the bulk of the book is not in any way dependent on knowledge of the specifics of particular implementation technologies for software.

The material in this book is intended for a wide audience, including, for example, real-time programmers, distributed-system programmers, object-oriented programmers, computer scientists, software engineers, electrical engineers with some software specialization, software designers, and software architects.

This book does not aim to provide a tutorial on basic principle of object-orientation or real time systems. We refer the reader to other books such as Jacobson [18], Selic [27], Coleman [13], to name a few, for background. We particularly recommend Jacobson for treatment of the basics of object orientation and the use case approach (as distinct from the *use case map* approach) and Selic for a system perspective that combines object-oriented and real time concerns.

METHODS THAT THESE TECHNIQUES SUPPLEMENT

The techniques of this book supplement a number of other object-oriented and real time design methods, as follows (in the discussion below we refer to the approach of this book as UCM, standing for use case maps):

- OOSE, by Jacobson et al [18]: UCM is philosophically compatible with OOSE in the sense that both focus on system design rather than programming. UCM is complementary to OOSE in the sense that it provides a high-level-design bridge across the rather large gap between OOSE's use cases and its detailed design approach with interaction sequence diagrams.
- ROOM, by Selic, Ward, Gulekson [27]: UCM is philosophically compatible with and complementary with ROOM in the same sense as above. UCM provides a high-level-design front end that complements the cooperating-state-machines approach of ROOM.
- OMT, by Rumbaugh et. al. [26]: UCM provides a high-level design front end that complements OMT in much the same way as above. OMT is more focused on class than system design. A strong point is its clean, practical notation for representing class relationships. Otherwise, OMT contains many elements that are superficially similar to OOSE and ROOM, namely scenario diagrams and state machines, but is weaker than either from a system design and modelling perspective.
- OOD, by Booch [3]: Booch has become a de-facto standard on fundamentals. We agree with many of Booch's observations on the design process. However, we find

his notations reflect programming issues more than system issues. UCM focuses on system issues first and brings in programming issues as details later. As such it can provide a high-level system-design front end to detailed design of object-oriented programs with Booch's approach.

- FUSION, by Coleman et al [13]: The Fusion method combines and extends the use case approach of OOSE with class-based design as in OMT and OOD. UCM can be used to supplement this approach by providing a transition from use case modeling to interaction-style diagrams.
- CRC or Responsibility Driven Design, by Wirfs-Brock, Wilkerson, Wiener [32]: This approach does not have a systems perspective but does use a front end modeling approach that is compatible with UCM.
- DP (Design Patterns), by Gamma, Helm, Johnson, Vlissides [14]: UCM adds a new type of high-level reusable pattern expressed with use case maps that complements DP patterns. UCM and DP patterns together cover a very wide range of problems and issues.
- SDWA (System Design With Ada [6]) and PVTSD (Practical Visual Techniques in System Design [7]), by R.J.A. Buhr: SDWA and PVTSD provide concepts, notations, and IPC patterns for real time systems, that have become standard in the Ada community. UCM provides techniques that supplement SDWA and PVTSD without invalidating their basic notions. UCM provides a better model of large-grained behaviour patterns than PVTSD's event scenarios. It also provides a more compact detailed-design notation than either of them.

RELATION TO SYSTEM MODELING TECHNIQUES

The UCM approach comes at system modeling from an entirely different angle than, for example, state machine models or Petri net models, to name just two of many types of executable or mathematical models of systems.

One use of techniques like state machines or Petri nets is to express behaviour requirements for systems in a precise and relatively complete way, viewing the system as a black box. Among other things, this enables complex requirements to be checked by automatic techniques, to help spot mistakes before they are built into implementations. The problem with such approaches from a design perspective is that they do not provide a progressive path to resolving the high level design issues raised above. They tend to build a wall between requirements and design. The wall is not important if implementations can be generated directly from models and maintained and evolved by changing the models, but this is still out of reach for the kinds of systems addressed by this book.

A different approach is to incorporate techniques like state machines or Petri nets into an executable system model, to specify the internal logic of the components. Such models exist and can even be used to generate implementation code, but they are at a detailed level of design, not the high level we are seeking in this book.

EXPERIENCE WITH THE APPROACH

The approach of this book came partly out of cooperative research and development projects with industry starting around 1990, partly out of experience teaching university undergraduate and graduate design classes starting around 1988, and partly from interactions with industry on the application of the techniques that resulted in refining them.

The approach has been thoroughly exercised in the classroom. It has been presented in approximately the form of this book to students in a series of undergraduate and graduate courses and several short courses to industry, during the period 1991 to 1995. A graduate course "Object-Oriented Design of Real Time and Distributed Systems" centering around this material has been offered several times, including to industry. A one-day training course on "Designing with Timethreads" (another way of saying "Designing with Use Case Maps") has been offered a number of times to industry. Some of the material on real-time systems has been used for several years in an undergraduate course on design and programming of real-time systems in a Computer System Engineering program and will appear in somewhat different form in another textbook that has evolved out of that course. This classroom experience smoothed the rough edges in the presentation of the ideas. However, it has also done more than that. Because the total audience has included a large number of experienced people from industry, the material has had a high level of "reality checking" that has resulted in a strong focus on practical applicability.

The ideas are relatively new and have not had time to have supporting tools developed or to have become established in widespread way in industry. So far, we can say that the techniques have met with enthusiastic response from many students, have acquired some champions in industry, and are being used on some practical projects, for example, during design reviews. The issue of tools is a chicken-and-egg one. Widespread use depends to some extent on tools, but the expense of developing tools is not justified until there is widespread use. If enough readers of this book judge the techniques are useful, experience suggests tools will follow.

OUTLINE OF THE BOOK

Chapter 1: Object Oriented and Real Time Come Together. This chapter sets the stage for the rest of the book. It starts from the premise that application push and design pull are driving object-oriented and real time issues together. Application push is the ever increasing demand on software for more of everything—functionality, flexibility, reusability, extensibility, performance, robustness, distributed operation. Design pull is the need for better design techniques to deal with the issues that this raises. This chapter associates the terms "object oriented" and "real time" with software implementation practices that are sufficiently different in detail that bringing the areas together is difficult with current design models. It identifies use case maps as a new kind of design model that helps to bring them together, and thus helps to satisfy both application push and design pull.

Chapter 2: The Behavioural Fabric of Systems. This chapter develops the idea of use case maps as concrete expressions of an abstract idea called the behavioural fabric. The behavioural fabric is the view we have in our minds of large-grained behaviour patterns of systems with which we are familiar. The notation is introduced and some examples presented taken from the physical and software worlds, including communicating fax machines and the model-view-controller paradigm of Smalltalk.
Chapter 3: Basic Use Case Map Model. This is a self-contained tutorial on the basics of the use case map model that explains the notation and provides rules and guidelines for creating legal maps, interpreting the maps in behaviour terms, binding the maps to components during design, and working with maps at different scales in a coordinated way. It defers issues of concurrent paths in maps to Chapter 7.
Chapter 4: A Context for Designing with Use Case Maps. This chapter describes a context in which design with use case maps may take place in a coordinated manner with other design models. The context positions design models in relation to four levels of design abstraction (requirements, high-level design, detailed design, implementation) and three basic domains of separable concerns within the levels (operation, manufacturing, assembly). The context includes notations for component types to cover the range of issues identified in Chapter 1.
Chapter 5: A Simple Example. A simple producer-consumer example is used to take a tour through the suite of models of Chapter 4. A very important feature of this chapter is that it shows in detail how to deal with the difficult problem of bringing dynamically changing software run-time structures into the high-level design picture. For concreteness, C++ code examples are provided.
Chapter 6: Case Study: A Conventional Object-Oriented Application from an Unconventional Perspective. Here the focus is on showing how to work with the use case maps to help with the design of a representative object-oriented application, a graphical user interface system called BGETool. A set of use case maps for this application is constructed using maps for the model-view-controller paradigm of Smalltalk as starting points. The case study uses all the design models in a coordinated way at all levels of design and implementation (some code is also provided).
Chapter 7: Advanced Use Case Map Model. This chapter extends the use case map model of Chapter 3 to include concurrent scenarios that may proceed at unpredictable rates relative to each other, may influence each other, may conflict with each other, and may fail before completion. It presents design at this level as an activity that positions components along paths to imply appropriate solution properties.
Chapter 8: Case Study: High-Level Design of a Real Time, Distributed System. This chapter focuses on issues in high-level design of a simple distributed application that is intended to be implemented using real time techniques, namely concur-

rent processes, timers, and interrupt service routines. It covers the following topics: discovering processes from maps, factoring maps to give smaller maps for subsystems, using maps as invariants for making design trade-offs, and working with maps at different scales in a coordinated manner. The example is a computer communications problem called the MTU (Message Transfer Utility) that, although superficially simple, exemplifies many of the characteristics that make designing real time systems difficult.

Chapter 9: Detailed Design Notation. This chapter provides a general collaboration graph notation that is both particularly simple and particularly widely applicable to a range of object-oriented and real time implementation techniques. It is positioned here to set the stage for detailed developments in the following two chapters. However, except for an overview section, the focus of the chapter is rather detailed and most of it is not needed to understand the essence of the following chapters.

Chapter 10: Case Study: Rounding Out the Real Time, Distributed System Example. This chapter rounds out the case study, not only in its own terms, but also in object-oriented terms. It shows how to make the transition from use case maps to collaboration graphs with processes in them. It shows how to examine a difficult detailed issue (dynamic buffering) in a high-level way using use case maps and how to bring the results back into the detailed domain. It shows how to bring in a class hierarchy that includes all the components in the maps and collaboration graphs, including processes, slots, teams, and fixed objects. It illustrates the idea of making use case maps a common denominator for evaluating trade-offs between real time and object-oriented issues.

Chapter 11: Patterns. This chapter draws together various patterns threads in this book and the literature. It does so by sketching some elements that might go into a patterns handbook that covers a wide range of concerns, including patterns in use case maps (called path patterns), IPC patterns, layering patterns, object interaction patterns, construction patterns for objects to fill slots, and construction patterns for processes and teams. The patterns are illustrated by examples drawn from the MTU and BGETool case studies.

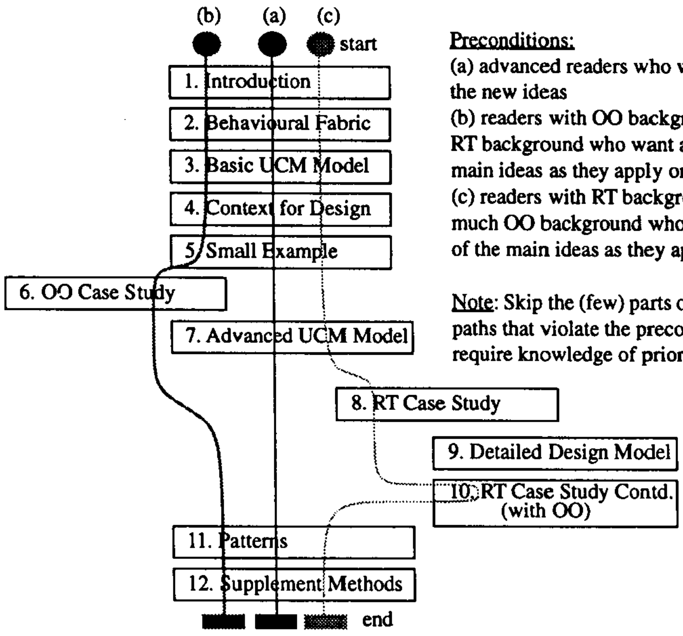
Chapter 12: Supplementing Familiar Design Methods. This chapter recaps the design models and the context for design used in this book, summarizes the reasons for including use case maps in any suite of design models, and suggests how use case maps may be used to supplement existing design methods.

Appendix A: Notation Summary. Reference cards for the new notations are provided, as well as a summary of some standard notations that we have borrowed from elsewhere.

Appendix B: Some Coding Examples. Several code examples are given for pieces of the case studies presented in the body of the text. The examples are in the C++ programming language. We have chosen C++ because it is a popular language, not because the ideas of this text are in any way tied to a specific language. There is enough explanation of the code that non-C++ programmers should be able to read and understand it.

A TOUR OF THE BOOK WITH USE CASE MAPS

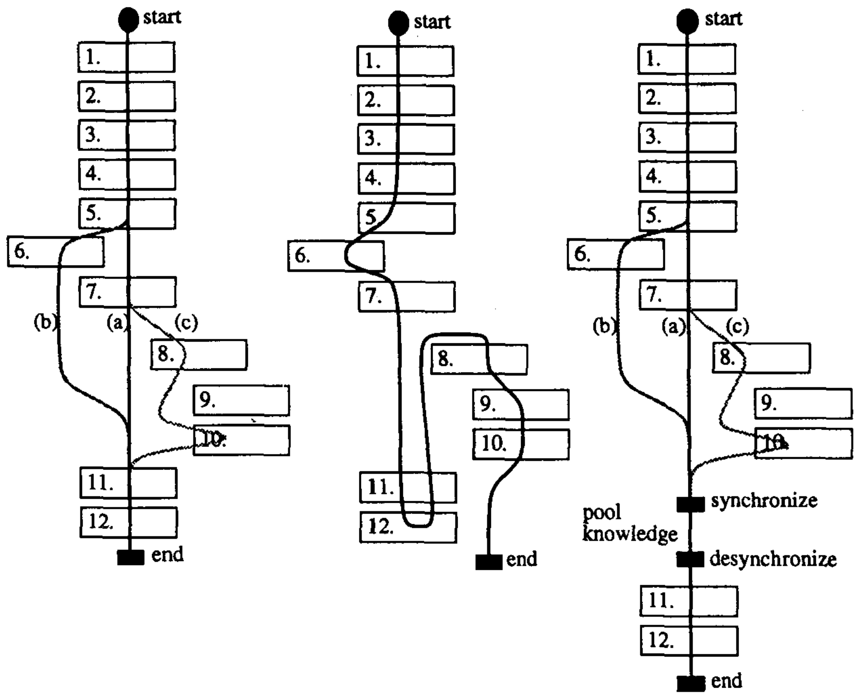
We provide a simple use case map below to suggest three selective patterns of reading this book. A path superimposed on a chapter box means *read the chapter.* It goes without saying that there is a fourth pattern that does not need a map—read the whole book from start to finish in the order in which it appears.



Preconditions:
(a) advanced readers who want only to skim the new ideas
(b) readers with OO background but not much RT background who want an overview of the main ideas as they apply only to OO
(c) readers with RT background but without much OO background who want an overview of the main ideas as they apply only to RT

Note: Skip the (few) parts of chapters along the paths that violate the preconditions or that require knowledge of prior skipped parts.

To give both a fuller sense of use case maps and some additional insight into other patterns for reading the book, we show a number of composite use case maps below as they would be actually drawn using the techniques of the book. On the left is the pattern above, redrawn to superimpose the shared parts of the paths. In the middle is a workable, out-of-order reading pattern for the whole book (this is not the only one). On the right is a concurrent study pattern that might be followed by a team of people who all read along the same path to begin with (1-5) and then follow different paths until (a) has finished 7, (b) has finished 6, and (c) has finished 8 and 10, whereupon they synchronize to pool their knowledge, and then desynchronize to continue independently through 11-12. This map looks like the one on the left, except for the synchronization and desynchronization bars. The difference is that the map on the left makes no assumptions about how many readers are following the different paths, whereas the one on the right assumes that at least one reader is following each path—otherwise no path could complete.



APPLICATIONS

There are two case studies in this book that were chosen to be representative of issues that are encountered at the two extremes of the range of implementation technologies for systems, namely *object oriented* and *real time*. One is a GUI program (Chapter 6) and the other is a small computer communications system (Chapter 8 and Chapter 10). To fit within the scope of a short book, they are relatively small-scale and self-contained problems that can be carried all the way through from high-level design to fragmentary implementation. We have tried to use the case studies as means to the end of illustrating