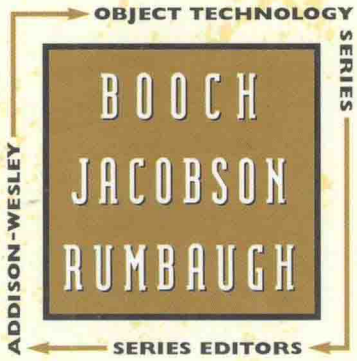
UML与面向对象设计影印丛书

UML实时系统开发

REAL-TIME UML SECOND EDITION

DEVELOPING EFFICIENT OBJECTS FOR EMBEDDED SYSTEMS

BRUCE POWEL DOUGLASS 编著



科学出版社
www.sciencep.com

UML与面向对象设计影印丛书

UML实时系统开发

Bruce Powel Douglass 编著

科学出版社

北京

内 容 简 介

嵌入式系统和实时系统的复杂程度日益提高，这要求系统设计方法更加成熟，可预见性更高。本书首先介绍了关于实时系统以及 UML 用于系统设计开发的基础知识，然后逐步讲解需求分析、对象结构及行为的定义、架构设计，还有细节设计，包括数据结构、操作、异常等。本书采用了大量的图表，让读者充分了解 UML 设计技巧，还提供了许多详细的设计实例，让读者掌握这些技巧在嵌入式系统设计中的应用。

本书实用性较强，可供嵌入式系统和实时系统设计开发人员阅读。

ISBN 0-201-65784-8

图字：01-2003-2547

图书在版编目(CIP)数据

UML 实时系统开发=Real-Time UML:Developing Efficient Objects for Embedded Sysgtems/（美）道格拉斯（Douglass,B.P.）编.—影印本. —北京：科学出版社，2003

ISBN 7-03-011403-5

Ⅰ.U… Ⅱ.道… 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 030822 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：22 1/2
印数：1—2 000 字数：427 000

定价：38.00 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

About the Author

Bruce was raised by wolves in the Oregon wilderness. He taught himself to read at age 3 and calculus before age 12. He dropped out of school when he was 14 and traveled around the US for a few years before entering the University of Oregon as a mathematics major. He eventually received his M.S. in exercise physiology from the University of Oregon and his Ph.D. in neurophysiology from the USD Medical School, where he developed a branch of mathematics called autocorrelative factor analysis for studying information processing in multicellular biological neural systems.

Bruce has worked as a software developer in real-time systems for almost 20 years and is a well-known speaker and author in the area of real-time embedded systems. He is on the Advisory Board of the *Embedded Systems* and *UML World* conferences, where he has taught courses in software estimation and scheduling, project management, object-oriented analysis and design, communications protocols, finite state machines, design patterns, and safety-critical systems design. He has developed and taught courses in real-time, object-oriented analysis and design for many years. He has authored articles for a number of journals and periodicals in the real-time domain.

Bruce is currently the Chief Evangelist[1] for i-Logix, a leading producer of tools for real-time systems development, and has worked with Rational and the other UML partners on the specification of the UML.

[1] Being a Chief Evangelist is much like being a Chief Scientist, except for the burning bushes.

He is one of the co-chairs of the Object Management Group's Real-Time Analysis and Design Working Group, which is currently examining the UML for possible future real-time extensions. He also consults, trains, and mentors a number of companies building large-scale, real-time, safety-critical systems. He is the author of four other books on software, including *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns* (Addison-Wesley, 1999) as well as a short textbook on table tennis.

Bruce enjoys classical music and has played classical guitar professionally. He has competed in several sports, including table tennis, bicycle racing, running, and full-contact Tae Kwon Do, although he currently only fights inanimate objects that don't hit back. He and his two sons contemplate epistemology in the frozen north. He can be reached at *bpd@ilogix.com*.

Foreword

Embedded computerized systems are here to stay. Reactive and real-time systems likewise. As this book aptly points out, one can see embedded systems everywhere; there are more computers hidden in the guts of things than there are conventional desktops or laptops.

Wherever there are computers and computerized systems, there has to be software to drive them. And software doesn't grow on trees. People have to write it; people have to understand and analyze it; people have to use it; and people have to maintain and update it for change in future versions. It is this human aspect of programming that calls for modeling complex systems on levels of abstraction that are higher than that of "normal" programming languages. From this also comes the need for methodologies to guide software engineers and programmers in coping with the modeling process itself.

There is broad agreement that one of the things to strive for in devising a high-level modeling approach is good diagrammatics. All other things being equal, pictures are usually better understood than text or symbols. But we are not interested just in pictures or diagrams, since constructing complex software is not an exclusively human activity. We are interested in *languages* of diagrams, and these languages require computerized support for validation and analysis. Just as high-level programming languages require not only editors and version-control utilities, but also—and predominantly!—compilers and debugging tools, so do modeling languages require not only pretty graphics, document generation utilities, and project management aids, but also means for executing models, for synthesizing code, and for true verification.

This means that we need *visual formalisms* that come complete with a syntax to determine what is allowed and semantics to determine what the allowed things mean. Such formalisms should be as visual as possible (obviously, some things do not lend themselves to natural visualization) with the main emphasis placed on topological relationships between diagrammatic entities, and then, as next-best options, also geometry and metrics, and perhaps iconics, too.

Over the years, the main approaches to high-level modeling have been *structured analysis* (SA), and *object orientation* (OO). The two are about a decade apart in initial conception and evolution. SA, started in the late 1970s by DeMarco, Yourdon, and others, is based on "lifting" classical, procedural programming concepts up to the modeling level and doing it graphically. The result calls for modeling system structure by functional decomposition and flow of information, depicted by (hierarchical) data-flow diagrams. As to system behavior, the early and mid-1980s saw several methodology teams (such as Ward/Mellor, Hatley/Pirbhai, and the STATEMATE team from I-Logix) making detailed recommendations that enriched the basic SA model with means for capturing behavior based on state diagrams or the richer language of statecharts. Carefully defined behavioral modeling, we should add, is especially crucial for embedded, reactive, and real-time systems.

OO modeling (often under the name of *OO analysis and design*, or OOAD) started in the late 1980s, and, in a way, its history is very similar. The basic idea for system structure was to "lift" concepts from object-oriented programming up to the modeling level, and to do so graphically. Thus, the basic structural model for objects in Booch's method, in the OMT and ROOM methods, and in many others, deals with classes and instances, relationships and roles, operations and events, and aggregation and inheritance. Visuality is achieved by basing this model on an embellished and enriched form of entity-relationship diagrams. As to system behavior, most OO modeling approaches adopted the statecharts language for this (a decision that the undersigned cannot claim to be too upset about). A statechart is associated with each class, and its role is to describe the behavior of the instance objects. The subtle and complicated connections between structure and behavior—that is, between object models and statecharts—were treated by OO methodologists in a broad spectrum of degrees of detail, from vastly insufficient to adequate. The test, of course, is whether the

languages for structure and behavior and their interlinks are defined sufficiently to allow the "interpretation" and "compilation" of high-level models—that is, full model execution and code synthesis, and eventually—and hopefully—also full formal verification against requirements. This was achieved only in a couple of cases, namely, in the ObjecTime tool (based on the ROOM method of Selic, Gullekson, and Ward), and the Rhapsody tool (from i-Logix, based on work of Gery and the undersigned on Executable Object Modeling).

In a remarkable departure from the similarity in evolution between the SA and OO paradigms for system modeling, the last four to five years have seen OO methodologists working together. They have compared notes, debated the issues, and finally cooperated in formulating a general Unified Modeling Language, or UML for short, in the hope of bringing together the best of the various OO modeling approaches. This sweeping effort, which in its teamwork is reminiscent of the Algol60 and Ada efforts, is taking place under the auspices of the Object Management Group, and was led by by Grady Booch (of the Booch method), Jim Rumbaugh (codeveloper of the OMT method), and Ivar Jacobson (czar of use-cases). Version 0.8 of the UML was released in 1996 and was rather open-ended, vague, and not nearly as well-defined as one might have expected. For about a year, the UML team went into overdrive, with a lot of help from methodologists and language designers from various companies, and version 1.0, whose defining documents were released in early 1997, was much tighter and more solid. Since then there have been a number of revisions. In 1997 the UML was adopted as a standard by the Object Management Group (OMG), and with more work there is a good chance that it will become not just an officially approved, if somewhat dryly documented, standard, but the main modeling mechanism of choice for the software that is constructed according to the object-oriented doctrine. And this is no small matter, as more and more software engineers are now claiming that more kinds of software are best developed in an OO fashion.

For capturing system structure, the UML indeed adopts a diagrammatic language for classes and objects that is based on the entity-relationship approach. For early-stage behavioral analysis, it recommends use cases and utilizes sequence diagrams (often called message sequence charts or MSCs), and for the full constructive specification of behavior it adopts statecharts, as modified in the aforementioned executable object modeling work.

Bruce Douglass' book does an excellent job of dishing out engineering wisdom to people who have to construct complex software—especially real-time, embedded, reactive software. Moreover, he does this using UML as the main underlying vehicle, a fact which, given the recent standardization of the UML and its fast-spreading usage, makes the book valuable to anyone whose daily worry is the expeditious and smooth development of such systems.

Moreover, Bruce's book is clear and very well written, and it gives the reader the confidence boost that stems from the fact that the author is not writing from the ivy-clouded heights of an academic institution or the religiously tainted vantage point of a professional methodologist, but that he has extensive experience in engineering the very kinds of systems the book deals with. This stark difference might be termed "the grand duality of system behavior." We are far from having a good algorithmic understanding of this duality. While statecharts seem adequate for the intraobject specification, sequence diagrams can specify use cases pretty well, but they are far too weak to serve the general role of full interobject specification (for example, they cannot specify "anti-scenarios"—ones that are forbidden). There have been recent proposals to extend sequence diagrams so that they can be used to capture more, but the jury is not in on those yet. Also, we are far from having a good algorithmic understanding of the duality between the two modes of modeling. We don't know yet how to effectively derive one view from the other, or even how to efficiently test whether descriptions presented in the two are mutually consistent.

The recent wave of popularity that the UML is enjoying will bring with it a true flood of books, papers, reports, seminars, and tools, describing, utilizing, and elaborating upon the UML, or purporting to do so. Readers will have to be extra careful in finding the really worthy trees in this messy forest. I have no doubt that Bruce's book will remain one of those.

As to the UML itself, one must remember that right now UML is a little *too* massive. We understand well only parts of it; the definition of other parts has yet to be carried out in sufficient depth to make crystal clear their relationships with the constructive core of UML (the class diagrams and the statecharts). For example, use-cases and their associated sequence and collaboration diagrams are invaluable to users and requirements engineers trying to work out the system's desired behavior in terms of scenarios. In the use-case world, we describe a single

scenario (or a single cluster of closely related scenarios) for all relevant objects—*inter-object behavior* we might call it. In contrast, a statechart describes all the behavior for a single object—which is *intra-object behavior.*

Other serious challenges remain, for which only the surface has been scratched. Examples include true formal verification of object-oriented software modeled using the high-level means afforded by the UML, automatic eye-pleasing and structure-enhancing layout of UML diagrams, satisfactory ways of dealing with hybrid systems that involve discrete, as well as continuous, parts, and much more.

As a general means for dealing with complex software, object-orientation is also here to stay. Perhaps this is true of the UML too, although my personal feeling is that in the wake of the initial excitement about a standard for modeling software the UML will have to be made smaller and tighter. Otherwise, it will become too cumbersome and multifaceted to be really useful. I think it will gradually shrink, leaving only three or four types of diagrams that are really needed and are useful. The rest will probably become obsolete and will eventually disapper.

OO is a powerful and wise way to think about systems and to program them, and will for a long time to come be part and parcel of the body of knowledge required by any self-respecting software engineer. This book will greatly help in that. On the other hand, OO doesn't solve *all* problems, and by extension neither does UML. There is still much work to be done. In fact, it is probably no great exaggeration to say that there is a lot more that we don't know and can't do yet in this business than what we do and can. Still, what we have is tremendously more than we would have hoped for just a few years ago, and for this we should be thankful and humble.

Professor David Harel
Dean, Faculty of Mathematics and Computer Science
The Weizmann Institute of Science
Rehovot, Israel
July 1999

Preface to the Second Edition

I have been both pleased and gratified by the success of the first edition of *Real-Time UML: Developing Efficient Objects for Embedded Systems.* I think the popularity of the first edition is due to both its timeliness and the appropriateness of object technology (in general) and the UML (in particular) to the development of real-time and embedded systems. At the time of the publication of the first edition, it was clear that the UML would be a major force in the development of object-oriented systems. However, even its strongest supporters have been surprised by the rapidity and near totality of its acceptance by developers. As one methodologist supporting a different modeling approach expressed to me, "I ignored the UML and then got hit with a freight train." The UML is wildly successful in the Darwinian sense of the term, as well in its technical superiority, and has become the most dominant life form in the object ecosphere.

As embedded systems gain in complexity, the old hack-and-ship approaches fail utterly and completely and, occasionally, spectacularly. The complexity of today's systems is driving developers to construct models of the system from different viewpoints in order to understand and plan the various system aspects. These views include the physical, or deployment, view, and the logical, or essential, view. Both views must support structural and behavioral aspects. This is what the UML is about, and this is why it has been so successful.

Audience

The book is oriented toward the practicing professional software developer and the computer science major in the junior or senior year. This book could also serve as an undergraduate- or graduate-level text, but the focus is on practical development rather than a theoretical introduction. Very few equations will be found in this book, but more theoretical and mathematical approaches are referenced where appropriate. The book assumes a reasonable proficiency in at least one programming language and at least a cursory exposure to the fundamental concepts of both object orientation and real-time systems.

Goals

The goals for the first edition remain goals for this edition, as well. This book is still meant to be an easy-to-read introduction to the UML and the application of its notation and semantics to the development of real-time and embedded systems. At the time of this writing, it is one of two books on the UML and real-time systems. I am also the author of the other, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns* (Addison-Wesley, 1999). *Doing Hard Time* is a more in-depth look at the fundamentals and vagaries of real-time systems, with emphasis on analysis of object schedulability, the use of behavioral patterns in the construction of statechart models, and how to use real-time frameworks effectively. It is a deeper exploration of real-time systems, which happens to use the UML to express these concepts. In contrast, *Real-Time UML* is primarily about the UML and secondarily about capturing the requirements, structure, and behavior of real-time systems using the UML.

In addition to these original goals for the first edition, the second edition adds two more: (1) to bring the book in conformance with the recent changes in the UML standard, and (2) to enhance the book's effectiveness based on feedback from the first edition.

The UML has undergone a couple of revisions since its original acceptance by the OMG. The first revision, 1.2, is almost exclusively

editorial, with no significant modification. The UML revision 1.3, on the other hand, is a significant improvement in a variety of ways. For example, the «uses» stereotype of generalization of use cases has now been replaced with the «includes» stereotype of dependency, which makes a great deal more sense.

Similarly, the notion of an action in UML 1.1 relied heavily on the use of "uninterpreted text" to capture its details. The UML 1.3 has elaborated the metamodel to encompass a number of different kinds of actions, making behavioral modeling more complete. The action semantics metamodel and how it relates to object messaging, is discussed in Chapters 2 and 4.

There have been a number of changes to the statechart model in the 1.3 revision, as well. The first edition of *Real-Time UML* devoted a lot of space to statecharts, and this second edition expends even more effort in the coverage of behavioral modeling with statecharts. Much of this space is used for the new features of statecharts—synch pseudostates, stub states, and so on. This resulted in a significant rewrite of Chapter 4, which deals with object behavioral modeling.

Recent consulting experience in fields ranging from advanced medical imaging to the next generation of intelligent, autonomous spacecraft, in addition to reader feedback from the first edition, is reflected in this second edition. For example, numerous consulting efforts have convinced me that many developers have a great deal of difficulty understanding and applying use cases to capture requirements for real-time and embedded systems. To address this need, I developed a one-day course called *Effective Use Cases*, which I have given at NASA and elsewhere. Principles that have proven their effectiveness in the field are captured here, in Chapter 2. Similarly, the techniques and strategies that have worked well for capturing object models or state behavior, have wound up expressed in this book, as well.

Another change in this book is the elaboration of an effective process for using the UML in product development. I call this process Rapid Object-Oriented Process for Embedded Systems (ROPES). The most common questions I have been asked since publication of the first book have been about the successful deployment of the UML in project teams developing real-time and embedded systems. Thus, Chapter 1 explains this process and identifies the work activities and artifacts produced during different parts of the iterative lifecycle. In fact, the

ROPES process forms the basis for the organization of the book itself, from Chapter 2 through 7.[2]

Despite the goals of the UML in terms of providing a standard, there has been some fractionalization as vendors try to differentiate themselves in the marketplace. While progress will naturally involve vendors providing new and potentially valuable model constructs above and beyond those provided by the UML, several vendors have claimed that their new features will be part of some new yet-to-be-announced UML for Real-Time. Interestingly, some of these vendors don't even participate in the OMG, while others provide mutually incompatible "enhancements." By spreading this FUD (fear, uncertainty, and doubt) among the developer community, I feel these vendors have done a great disservice to their constituency. Developers should understand both the benefits *and* risks of using single-source modeling concepts. These features may make the system easier to model (although, in many cases, these so-called enhancements fail in that regard), but they also lock the product development to a single vendor's tool. Another risk is the inability to use model interchange between tools when the models no longer adhere to the UML standard. This can greatly decrease the benefits to the developer of using the UML. In an effort to dispel some of the FUD, I've added Appendix B to outline what it means to make changes to the standard, why *no* single vendor can claim it owns the UML standard (it is, after all, owned by the OMG), and what changes are likely to be made to the UML over the next several years.

Finally, I would suggest that interested readers visit the I-Logix Web site, *www.ilogix.com*. There you will find a number of papers on related topics, written by myself and others, as well as the UML specifications, tool descriptions, and links to relevant sites.

Bruce Powel Douglass, Ph.D.
Spring, 1999

[2] More information on the ROPES process can be had from the I-Logix web site, *www.ilogix.com*, as well in another book, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns* (Addison-Wesley, 1999).

Preface to the First Edition

Goals

Real-Time UML: Developing Efficient Objects for Embedded Systems is an introduction to object-oriented analysis and design for hard real-time systems using the Unified Modified Language (UML). UML is a third-generation modeling language that rigorously defines the semantics of the object metamodel and provides a notation for capturing and communicating object structure and behavior. Many methodologists—including Grady Booch (Booch Method), Jim Rumbaugh (Object Modeling Technique [OMT]), Ivar Jacobson (Object-Oriented Software Engineering [OOSE]), and David Harel (Statecharts)—collaborated to achieve UML. Many more participated, myself included, in the specification of the UML, and we believe that it is the leading edge in modeling for complex systems.

There are very few books on the use of objects in real-time systems and even fewer on UML. Virtually all object-oriented books focus primarily on business or database application domains and do not mention real-time aspects at all. On the other hand, texts on real-time systems have largely ignored object-oriented methods. For the most part, they fall into two primary camps: those that bypass methodological considerations altogether and focus solely on "bare metal" programming, and those that are highly theoretical, with little advice for actually implementing workable systems. *Real-Time UML: Developing Efficient Objects for Embedded Systems* is meant to be a concise and timely bridge for these technologies, presenting the development of deployable real-time systems using the object semantics and notation of the