大学计算机教育丛书（影印版）

# SOFTWARE ARCHITECTURE

PERSPECTIVES ON AN EMERGING DISCIPLINE

MARY SHAW　DAVID GARLAN

# 软件体系结构

一门初露端倪学科的展望

清华大学出版社 · PRENTICE HALL

# SOFTWARE ARCHITECTURE

PERSPECTIVES ON AN EMERGING DISCIPLINE

# 软件体系结构

一门初露端倪学科的展望

Mary Shaw
*Carnegie Mellon University*
David Garlan
*Carnegie Mellon University*

清华大学出版社
**Prentice-Hall International,**

(京) 新登字 158 号

[illegible]

Software architecture: perspectives on an emerging discipline/Mary Shaw, David Garlan

**图书在版编目(CIP)数据**

软件体系结构:英文/肖(Shaw, M.), 加兰(Garlan, D.)著.—影印版.—北京:清华大学出版社, 1998.4

(大学计算机教育丛书)

ISBN 7-302-02832-X

Ⅰ.软… Ⅱ.①肖… ②加… Ⅲ.软件工程-英文 Ⅳ.TP311.5

中国版本图书馆 CIP 数据核字(98)第 01689

**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**: 8 3/8

**版　次**: 1998 年 5 月第 1 版　1998 年 9 月第 2 次印刷

**书　号**: ISBN 7-302-02832-X/TP・1489

**印　数**: 5001～10000

**定　价**: 15.00 元

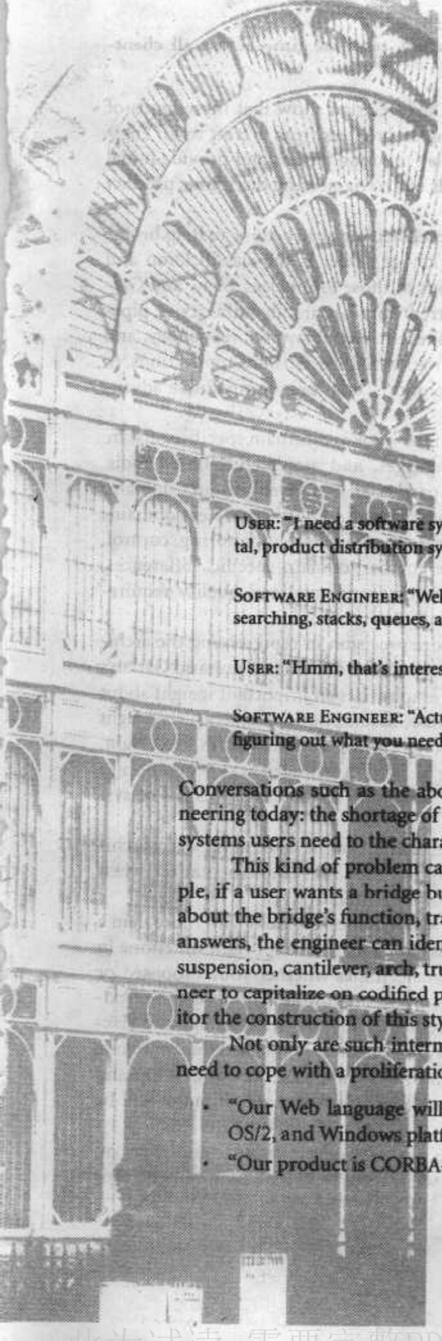
# 出版前言

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

**User:** "I need a software system that will help me manage my factory" (or my hospital, product distribution system, satellite system, etc.)

**Software Engineer:** "Well, let's see. I can put together components that do sorting, searching, stacks, queues, and so forth."

**User:** "Hmm, that's interesting. But how would those fit into my system?"

**Software Engineer:** "Actually, at a pretty low level. We'll have to spend some time figuring out what you need at a higher level, and how it would all fit together."

Conversations such as the above illustrate one of the biggest problems in software engineering today: the shortage of intermediate abstractions that connect the characteristics of systems users need to the characteristics of systems that software engineers can build.

This kind of problem can usually be handled in other engineering fields. For example, if a user wants a bridge built, a civil engineer can ask the user a number of questions about the bridge's function, traffic loads, setting, and environmental factors. Based on the answers, the engineer can identify the most appropriate architectural style for the bridge: suspension, cantilever, arch, truss, etc. This intermediate abstraction then enables the engineer to capitalize on codified principles and experience to specify, analyze, plan, and monitor the construction of this style of bridge, with high levels of efficiency and confidence.

Not only are such intermediate abstractions scarce in the software field, but also we need to cope with a proliferation of architectural claims such as:

- "Our Web language will enable your applications to operate on Unix, Macintosh, OS/2, and Windows platforms."
- "Our product is CORBA-compatible, and thus will fully interoperate."

- "Our system's balanced architecture ensures fast response time across all client-server configurations."

In Chapter 1 of this book, Mary Shaw and David Garlan show that these kinds of problems generally characterize a field that is trying to progress from a craft to an engineering discipline. In the remaining chapters, they lay the foundations and provide initial concepts and techniques for one of the critical needs of an engineering discipline: product architecting.

In particular, they provide several classes of intermediate abstractions to help bridge the gap between software needs and solutions. A key gap-filler is the classification and analysis of architectural styles for software, analogous to those for bridges. Shaw and Garlan provide definitions and discussions of major current software architectural styles: pipes and filters; data abstraction and object-orientation; event-based; layered; repository; and process control. They also apply the styles to some representative software applications, to show the differences among the resulting design solutions, and their comparative advantages and disadvantages.

Other key architectural gap-fillers provided in the book are domain specific software architectures (DSSAs), architecture definition languages, and architecture-based tools. DSSAs provide a set of intermediate abstractions particular to a given product domain, such as factory, hospital, product distribution, or satellite control domains. These domains may share some abstractions, such as functions for data acquisition, monitoring, control, and decision support. But they will have some further domain-specific differences, depending on characteristics of their typical users, environments, and quality requirements such as safety and information security.

Architecture definition languages provide more precision in representing the architecture of a system than do the usual software box-and-arrow drawings. Shaw and Garlan's treatment of architecture definition languages also emphasizes an important insight about software architecting: getting the connectors (interface assumptions, protocols, etc.) right is at least as important as getting the components (algorithms, data structures, etc.) right. Architecture definition languages also provide the basis for a stronger next generation of tools for defining a software architecture, and for reasoning about the properties of systems which would be built to that architecture.

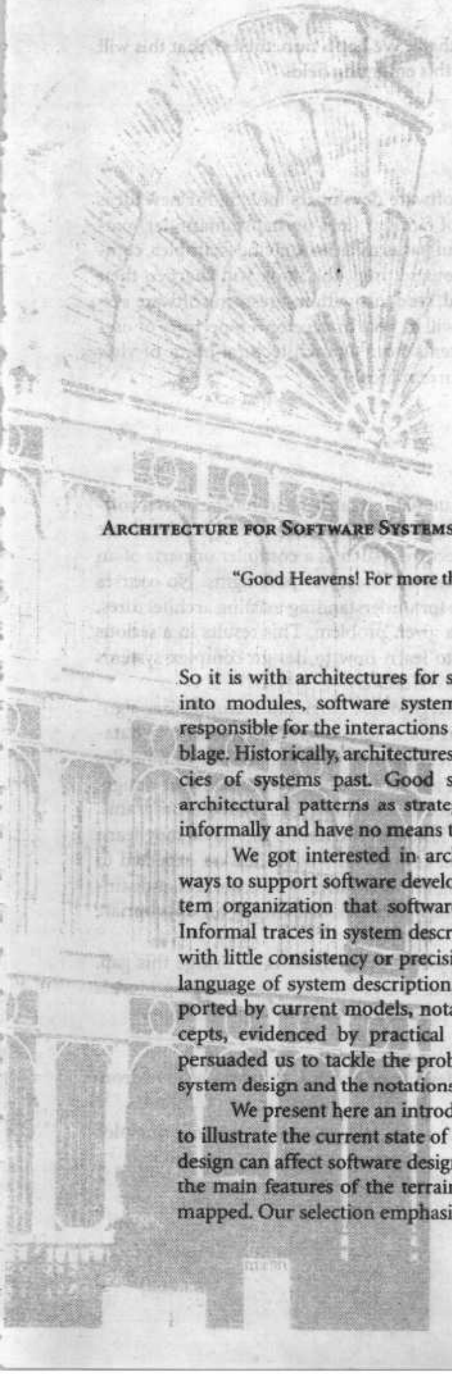
Architecture definition languages and tools enable this to be done at the early architecting stage, rather than finding out about these properties after implementation, when the cost and freedom to change the architecture is often prohibitive.

Thus, software architectures provide the software engineering field with more than a set of gap-filling abstractions. They provide the basis for the most important milestone in the software life cycle process: the milestone that determines whether your proposed or default architecture has the strength to cope with current and future workloads; the flexibility to adapt to changing technology and requirements; and the affordability and risk-freedom to be developed within its planned budget and schedule. If you pass this milestone successfully, you have a confident basis for committing major resources to develop and sustain the software system. If not, the de facto architecture you marry in haste will be there for you to repent at leisure.

My favorite chapter in the book is Chapter 5, which begins to provide guidelines on how to determine an architecture which best fits a set of software system requirements. For a class of user interface software, it establishes a "design space" of functional dimensions (required portability, customizability, external event handling, basic user interface mode, etc.) and structural dimensions (abstraction level of the application program interface, control thread mechanism, communication mechanisms, etc.). It then provides guidelines for matching structural dimension choices to functional dimension characteristics, and for reconciling structural design choices with each other. This provides the beginning of an engineering discipline which can be taught to students and applied across increasing ranges of software projects.

Another good feature of the book is its guidance on organizing and teaching a course on software architecture, based on several years' experience in teaching such a course at Carnegie Mellon University. At USC, we are beginning to offer a course on software architecture for our MS program in software engineering, and are finding that the book provides a good set of organizing concepts and material for the course. A final bit of expectations management: this book is a first cut at codifying a just-emerging field. It has some rough spots, and it doesn't provide all the answers. It won't provide you with fully mature industry-consensus architecting languages and terminology; surefire cookbook architecting solutions; or tools that automate the analysis of complex tradeoffs among functionality, performance, cost, and various desired software qualities. On the other hand, it provides the best general framework and set of techniques for dealing with software architectures that is available today. And it conveys the excitement of being able to look at the software field in new ways, and of experiencing a new branch of software engineering in the process of being developed and applied.

*BARRY BOEHM*
*TRW Professor of Software Engineering, USC*



# PREFACE

## Architecture for Software Systems

"Good Heavens! For more than forty years I have been speaking prose without knowing it!"

Molière, *Le Bourgeois Gentilhomme*, Act II, sc. iv

So it is with architectures for software systems. Ever since the first program was divided into modules, software systems have had architectures, and programmers have been responsible for the interactions among the modules and the global properties of the assemblage. Historically, architectures have been implicit—accidents of implementation, or legacies of systems past. Good software developers have often adopted one or several architectural patterns as strategies for system organization, but they use these patterns informally and have no means to make them explicit in the resulting system.

We got interested in architectures for software systems while investigating better ways to support software development. We were struck by the evidence of patterns for system organization that software developers use purposefully but nearly unconsciously. Informal traces in system descriptions reveal a substantial folklore of system design, used with little consistency or precision. Here—waiting to be exposed and organized—is a rich language of system description. Its vocabulary includes constructs and patterns not supported by current models, notations, or tools. The clear utility of the architectural concepts, evidenced by practical use even in the absence of crisp definitions or tools, persuaded us to tackle the problem of closing the gap between the useful abstractions of system design and the notations and tools. This book is one of the results.

We present here an introduction to the field of software architecture. Our purpose is to illustrate the current state of the discipline and examine the ways in which architectural design can affect software design. Naturally, a short volume such as this can only highlight the main features of the terrain; indeed, the terrain is even now in the process of being mapped. Our selection emphasizes informal descriptions, touching lightly on formal notations and specifications, and on tools to support them. We hope, nonetheless, that this will serve to illuminate the nature and significance of this emerging field.

## AUDIENCE

The book serves two groups. First, professional software developers looking for new ideas about system organization will find discussions of familiar (and perhaps unfamiliar) patterns for system organization. By identifying useful patterns clearly, giving examples, comparing them, and evaluating their utility in various settings, the book will sharpen their understanding and broaden their options. Second, students with interests in software system organization will find fresh ideas here. They will be able to develop a repertoire of useful techniques that allows them to approach systems from an architectural point of view and that goes beyond the single-mindedness of current fads.

## EDUCATION IN SOFTWARE ARCHITECTURE

Software architectures now receive little or no systematic treatment in most existing software engineering curricula, either undergraduate or graduate. At best, students are exposed to one or two specific application architectures (such as a compiler or parts of an operating system) and may hear about a few other architectural paradigms. No courses seriously attempt to develop comprehensive skills for understanding existing architectures, developing new ones, or selecting one to match a given problem. This results in a serious gap in current curricula: students are expected to learn how to design complex systems without the requisite intellectual tools for doing so effectively.

The software component of the typical undergraduate curriculum emphasizes algorithms and data structures. Although courses on compilers, operating systems, or databases are usually offered, there is no systematic treatment of the organization of modules into systems, or of the concepts and techniques at an architectural level of software design. Thus, system issues are seriously underrepresented in current undergraduate programs. Further, students now face a large gap between lower-level courses, in which they learn programming techniques, and upper-level project courses, in which they are expected to design more significant systems. Without knowing the alternatives and criteria that distinguish good architectural choices, the already challenging task of defining an appropriate architecture becomes formidable.

We have developed a course, *Architectures for Software Systems*, to bridge this gap Largely using the materials of this book, the course brings together the emerging models for software architectures and the best of current practice.

Specifically, the course does the following:

- Teaches how to understand and evaluate designs of existing software systems from an architectural perspective
- Provides the intellectual building blocks for designing new systems in principled ways, using well-understood architectural paradigms

- Shows how formal notations and models can be used to characterize and reason about a system design
- Presents concrete examples of actual system architectures that can serve as models for new designs

This book can be used, together with supplemental readings, as a text for a such a course. (Chapter 9 describes the course in more detail.) Equally well—and perhaps more practical for many—the book can be used as a supplemental text for courses in software engineering or software design.

## ACKNOWLEDGMENTS

This volume integrates the results of several years of research that depended critically on the contributions of our collaborators. Most significantly, our collaborators on results incorporated here are godparents to the current work. We particularly appreciate the willingness of Tom Lane, Toru Asada, Roy F. Swonger, Nadine Bounds, Paul Duerig, and Marco Schumacher to allow us to include their papers on design guidance [Lan90a, ASBD92, S+94]. Thanks go to our co-authors on other work: The formalization of instrumentation systems represents joint work with Norman Delisle and others from Tektronix. The formalization of implicit invocation systems represents collaborative work with David Notkin and Kevin Sullivan. Our collection of sample architectural problems was developed in collaboration with Rob Allen, Daniel Klein, John Ockerbloom, Curtis Scott, and Marco Schumacher. The formalization of software architecture was joint work with Rob Allen and Gregory Abowd. The implementation of implicit invocation was done in collaboration with Curtis Scott. The UniCon language and tool were developed jointly with Rob DeLine, Daniel Klein, Theodore Ross, David Young, and Gregory Zelesnik. The Aesop System was developed with the help of Rob Allen, John Ockerbloom, Ralph Melton, and Bob Monroe as well as numerous undergraduates. The report on initial experience with a course reports a joint offering with Chris Okaski, Curtis Scott, and Roy Swonger.

These, of course, are not the only ones who helped. Many colleagues have contributed to the development of the ideas by helping with examples, challenging our ideas, insisting that we be rigorous, providing constructive comments on drafts, and in untold other ways. So, thanks again to James Alstad, Pepe Galmes, Lorin Grubb, Chris Okasaki, Curtis Scott, and Roy Swonger for their help in developing our course on this material; David Notkin, Kevin Sullivan, and Gail Kaiser for their contributions to understanding event-based systems; Rob Allen for help in developing a rigorous understanding of the pipe-and-filter style; the oscilloscope development team at Tektronix for their part in demonstrating the value of domain-specific architectural styles in an industrial context; Eldon Shaw for fostering Mary's appreciation for engineering; Roy Weil for providing engineering sensibilities and tolerating the vagaries of authorhood; Angle Jordan for arranging the opportunity to study the history of engineering; Bill Wulf and Ralph London for the long-term collaboration that taught us the need to choose different architectures for different problems; colleagues at Fisher Controls, for the opportunity to learn about control software; Marc Graham, for arranging discussions with most of the DARPA Domain-Specific Software Architecture groups; Will Tracz, for presenting an architecture for avionics whose essential core was essentially a feedback loop, thereby provoking another look at process control; Allen Newell, for inspiring the analysis of shared information systems; David Steier for discussions about Soar/IBDE; Daniel Jackson and Jeannette Wing for their help in clarifying the benefits and limitations of formal approaches to software architecture; Lynette Garlan for letting David come to Pittsburgh to pursue research in software architecture; David Notkin, Kevin Sullivan, and Rob Allen for their collaborative efforts in developing a scientific basis for implicit invocation; Raj Rajkumar, for helping us incorporate real-time analysis in UniCon; Robert DeLine, Daniel Klein, Fuchun Jiang, and Gregory Zelesnik, for contributions to the UniCon implementation; Michael Baumann, Chanakya C. Damarla, Steven Fink, Doron Gan, Andrew Kompanek, Curtis Scott, Ralph Melton, Bob Monroe, Brian Solganick, Peter Su, and Steve Zdancewic for contributions to the Aesop implementation; Gregory Abowd, Rob Allen, Mario Barbacci, Rob DeLine, Stu Feldman, Marc Graham, Kevin Jeffay, Dan Klein, Eliot Moss, John Ockerbloom, Reid Simmons, Pamela Zave, José Galmes, and Greg Zelesnik for participating as guest lecturers in our course.

We also thank Barry Boehm, Rob DeLine, Larry Druffel, Frank Friedman, Norm Gibbs, Bill Griswold, Ralph Johnson, Nancy Mead, Eliot Moss, Allen Newell, David Notkin, Gene Rollins, Robert Schwanke, Dilip Soni, Will Tracz, Roy Weil, Jeannette Wing, members of CMU's Software Architecture Reading Group, and numerous anonymous referees for their constructive comments on drafts of various parts of the work.

Research can't be done without support, and we appreciate the interest and support of numerous government, academic, and industrial sponsors. The work was funded variously by the Department of Defense Advanced Research Project Agency under grant MDA972-92-J-1002; the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330; the U.S. Federal Government under Contract Number F19628-90-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a Federally Funded Research and Development Center; Mobay Corporation; National Science Foundation Grants CCR-9109469, CCR-9112880, and CCR-9357792; Siemens Corporate Research; Digital Equipment Corporation's Graduate Engineering Education Program (for support for collaborators); and the Carnegie Mellon University School of Computer Science and Software Engineering Institute (which is sponsored by the U.S. Department of Defense).

The views and conclusions here are our own. They should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government, the U.S. Department of Defense, the National Science Foundation, Wright Laboratory, Siemens Corporation, Mobay Corporation, or Carnegie Mellon University.

Much of the material in this book is derived from work we have published in other forums. We have relied principally on the following.

- Chapter 1: "Prospects for an engineering discipline of software" [Sha90].
- Chapter 2: "An introduction to software architecture" [GS93b], "Beyond Objects: A software design paradigm based on process control" [Sha95].
- Chapter 3: "An introduction to software architecture" [GS93b], "Beyond objects: A software design paradigm based on process control" [Sha95], "Candidate model problems in software architecture" [S+94].

- Chapter 4: "Software architectures for shared information systems" [Sha93b].
- Chapter 5: "Studying software architecture through design spaces and rules" [Lan90a], "The quantified design space—A tool for the quantitative analysis of Designs" [ASBD92].
- Chapter 6: "Formal approaches to software architecture" [Gar93].
- Chapter 7: "Characteristics of higher-level languages for software architecture" [SG94], "Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status" [Sha93a], "Adding implicit invocation to traditional programming languages" [GS93a].
- Chapter 8: "Abstractions for software architecture and tools to support them" [SDK+95], "Exploiting style in architectural design environments" [GAO94], "Beyond definition/use: architectural interconnection" [AG94a].
- Chapter 9: "Experience with a course on architectures for software systems" [GSO+92].

We also wish to thank the authors and publishers of several of the figures for granting permission to reprint them here.

Figure 1.1 is reproduced with permission of the McGraw-Hill Companies from *Computer Structures* by G. Bell and A. Newell, McGraw-Hill 1971.

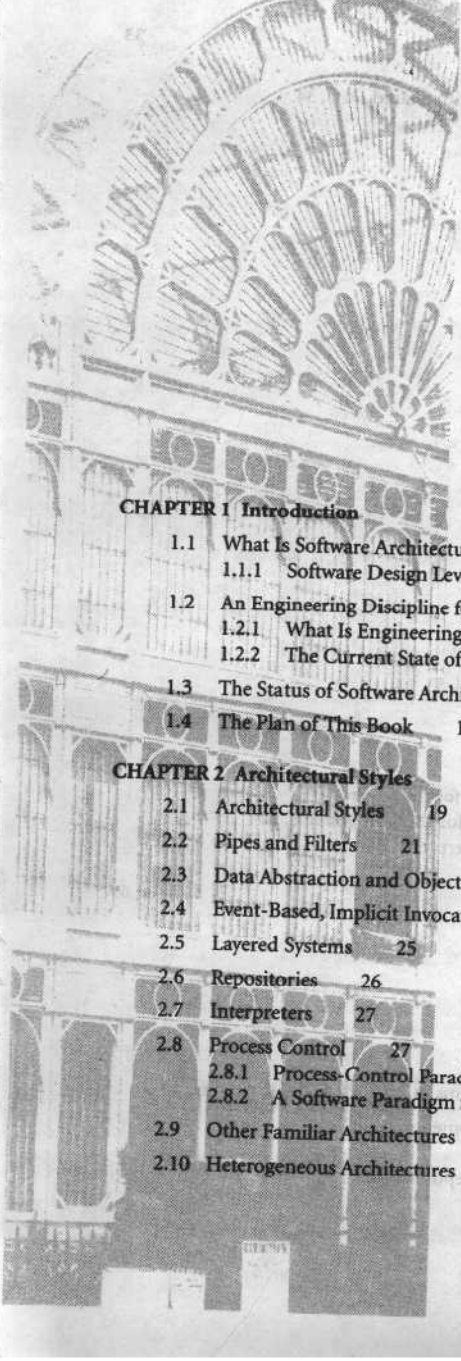
Figures 3.16, 3.17, 4.8, 4.9, 4.11, 4.18 are reprinted with permission of IEEE from Grady Booch's "Object-oriented development" which appeared in *IEEE Transactions on Software Engineering* in February 1986, Won Kim and Jungysun Seo's "Classifying schematic and data heterogeneity in multidatabase systems" which appeared in *IEEE Computer* in December 1991, Rafi Ahmet et al.'s "Pegasus heterogeneous multidatabase system" which appeared in *IEEE Computer* in December 1991, Minder Chen and Ronald Norman's "Framework for integrated CASE" which appeared in *IEEE Software* in March 1992, and Gio Wiederhold's "Mediators in the architecture of future information systems" which appeared in *IEEE Computer* in March 1992; all © IEEE.

Figures 3.22 and 3.23 are reprinted from PROVOX product literature with permission from Fisher-Rosemount Systems, Inc.

Figure 3.25 is reprinted with permission of the Association for Computing Machinery from Frederick Hayes-Roth's "Rule-based systems" which appeared in *Communications of the ACM* in September 1985.

Figure 3.27 is reprinted with permission of the American Association for Artificial Intelligence from H. Penny Nii's "Blackboard systems" which appeared in *AI Magazine* volume 7 numbers 3 and 4, (c) American Association for Artificial Intelligence.

Figures 4.1, 4.2, 4.4, and 4.5 are reprinted by permission of John Wiley and Sons from Laurence J. Best's *Application Architecture* © 1990 John Wiley and Sons.



# CONTENTS