

CLASSICAL AND OBJECT-ORIENTED SOFTWARE ENGINEERING

► With UML and C++

STEPHEN R. SCHACH

传统和面向对象的软件工程
第4版

FOURTH EDITION



McGraw-Hill Book Co
世界图书出版公司

CLASSICAL AND OBJECT-ORIENTED SOFTWARE ENGINEERING

with UML and C++

FOURTH EDITION

Stephen R. Schach
Vanderbilt University

世界图书出版公司
McGraw-Hill Book Co

WCB/McGraw-Hill

A Division of The McGraw-Hill Companies

CLASSICAL AND OBJECT-ORIENTED SOFTWARE ENGINEERING WITH UML AND C++

Copyright © 1999 by The McGraw-Hill Companies, Inc. All rights reserved. Previous editions © 1990, 1993, and 1996, by Richard D. Irwin, a Times Mirror Higher Education Group, Inc., company. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

ISBN 0-07-290168-3

Copyright © 1999 by McGraw-Hill Companies, Inc. All Rights reserved. Jointly Published by Beijing World Publishing Corporation/McGraw-Hill. This edition may be sold in the People's Republic of China only. This book cannot be re-exported and is not for sale outside the People's Republic of China.

IE ISBN 0-07-116871-0

Library of Congress Cataloging-in-Publication Data

Schach, Stephen R.

Classical and object-oriented software engineering with UML and C++ /
Stephen R. Schach.—4th ed.

p. cm.

First-2nd eds. published under title: Software engineering.

Third ed. published under title: Classical and object-oriented
software engineering.

Includes bibliographical references and index.

ISBN 0-07-290168-3

1. Software engineering. 2. Object-oriented programming (Computer
science) I. Schach, Stephen R. Software engineering.

QA76.758.S318 1998

005.1—dc21

98-15171

<http://www.mhhe.com>

The following are registered trademarks:

Access	IMS/360	OS/VS2
ActiveX	Informix	Post-it
ADF	Iona	PowerBuilder
ADW	Java	Project
Aide-de-Camp	Lotus 1-2-3	PVCS
Analyst/Designer	Lucent Technologies	QAPartner
Apple	Macintosh	RAMIS-II
AT&T	MacProject	Rational
Bachman Product Set	Method/I	Rose
Battlemap	Microsoft	SoftBench
Borland	Motif	Software through Pictures
Bull	MS-DOS	Source Safe
CA-Tellaplan	MVS/360	SPARCstation
CCC	Natural	SQL
Coca-Cola	Netscape	Statemate
COM	<i>The New York Times</i>	Sun
DB2	Newton	Sun Microsystems
Demo II	Nomad	SunView
Emeraude	Object C	System Architect
Excel	ObjectBroker	Teamwork
Excelerator	Objective-C	UNIX
Ford	OLE	VAX
Foundation	OMTool	Visigenic
FoxBASE	1-800-FLOWERS	VM/370
Guide	OpenWindows	VMS
Hewlett-Packard	OpenDoc	<i>The Wall Street Journal</i>
Honeywell	Oracle	Windows 95
Hypercard	ORB Plus	Word
Hypertalk	ORBIX	X11
IBM	OS/360	XRunner
IEW	OS/370	

PREFACE

The title of this book, *Classical and Object-Oriented Software Engineering with UML and C++*, is somewhat surprising. After all, there is virtually unanimous agreement that the object-oriented paradigm is superior to the classical (structured) paradigm. It would seem obvious that an up-to-date software engineering textbook should describe only the object-oriented paradigm, and treat the classical paradigm at best as a historical footnote.

That is not the case. Despite the widespread enthusiasm for the object-oriented paradigm and the rapidly accumulating evidence of its superiority over the classical paradigm, it is nevertheless essential to include material on the classical paradigm. There are three reasons for this. First, it is impossible to appreciate why object-oriented technology is superior to classical technology without fully understanding the classical approach and how it differs from the object-oriented approach.

The second reason why both the classical and object-oriented paradigms are included is that technology transfer is a slow process. The vast majority of software organizations have not yet adopted the object-oriented paradigm. It is therefore likely that many of the students who use this book will be employed by organizations that still use classical software engineering techniques. Furthermore, even if an organization is now using the object-oriented approach for developing new software, existing software still has to be maintained, and this legacy software is not object-oriented. Thus, excluding classical material would not be fair to the students who use this text.

A third reason for including both paradigms is that a student who is employed at an organization that is considering the transition to object-oriented technology will be able to advise that organization regarding both the strengths and the weaknesses of the new paradigm. Thus, as in the previous edition, the classical and object-oriented approaches are compared, contrasted, and analyzed.

The Fourth Edition differs from the Third Edition in two ways. First, many new topics are introduced in this edition. Second, the material has been rearranged to support both one- and two-semester software engineering curricula: this is described in the next section.

With regard to new topics, Unified Modeling Language (UML) permeates this edition; this is reflected in the title of the book. In addition to utilizing UML for object-oriented analysis and object-oriented design, UML has also been used wherever there are diagrams depicting objects and their interrelationships. UML has become a de facto software engineering standard and this is reflected in the Fourth Edition.

Another new topic is design patterns. This material is part of a new chapter on reuse, portability, and interoperability. Other reuse topics in this chapter include software architecture and frameworks. Underlying all the reuse material is the importance of object reuse. The portability sections include material on Java. With regard to interoperability, there are sections on topics like OLE, COM, ActiveX, and CORBA.

There is also a new chapter on planning and estimating, especially for the object-oriented paradigm. The chapter therefore includes new material on feature points and COCOMO II.

The synchronize-and-stabilize life-cycle model used by Microsoft has been included in this edition. The associated team organization method is also described.

As in the previous edition, particular attention is also paid to object-oriented life-cycle models, object-oriented analysis, object-oriented design, management implications of the object-oriented paradigm, and to the testing and maintenance of object-oriented software. Metrics for objects are also included. In addition, there are many briefer references to objects, a paragraph or even just a sentence in length. The reason is that the object-oriented paradigm is not just concerned with how the various phases are performed, but rather permeates the way we think about software engineering. Object technology pervades this book.

The software process is still the concept that underlies the book as a whole. In order to control the process, we have to be able to measure what is happening to the project. Accordingly, the emphasis on metrics is retained. With regard to process improvement, material on SPICE has been added to the sections on the Capability Maturity Model (CMM) and ISO 9000.

Another topic that still is stressed is CASE. I also continue to emphasize the importance of maintenance and the need for complete and correct documentation at all times.

The software process is essentially language-independent and this is again reflected in the Fourth Edition. The few code examples are in C++. However, care has been taken to make this material accessible to readers with little or no knowledge of C++ by providing explanations of constructs that are specific to C++.

With regard to prerequisites, it is assumed that the reader is familiar with one high-level programming language such as Pascal, C, C++, Ada, BASIC, COBOL, FORTRAN, or Java. In addition, the reader is expected to have taken a course in data structures.

HOW THE FOURTH EDITION IS ORGANIZED

The Third Edition of this book was written for a one-semester, project-based software engineering course. The book accordingly consisted of two parts. Part 2 covered the life cycle, phase by phase; the aim was to provide the students with the knowledge and skills needed for the Term Project. Part 1 contained the theoretical material needed to understand Part 2. For example, Part 1 introduced the reader to CASE, metrics, and testing because each chapter of Part 2 contained a section on CASE tools for that phase, a section on metrics for that phase, and a section on testing during that phase. Part 1 was kept short to enable the instructor to start Part 2 relatively early in the semester. In this way, the class could begin developing the Term Project as soon as possible. The need to keep Part 1 brief meant that I had to include topics like reuse, portability, and team organization in Part 2. Thus, while the students were working on their term projects, they learned additional theoretical material.

However, there is a new trend in software engineering curricula. More and more computer science departments are realizing that the overwhelming prepon-

derance of their graduates find employment as software engineers. As a result, many colleges and universities have introduced a two-semester (or two-quarter) software engineering sequence. The first course is largely theoretical (but there is almost always a small project of some sort). The second course is a major team-based term project, usually a capstone project. When the Term Project is carried out in the second semester, there is no need for the instructor to rush to start Part 2.

In order to cater to both the one- and two-semester course sequences, I have rearranged the material of the previous edition and added to it. Part 1 now includes two more chapters, but the material of those two chapters is not a prerequisite for Part 2. First, Chapter 7 is entitled "Reusability, Portability, and Interoperability." The theme of this chapter is the need to develop reusable portable software that can run on a distributed heterogeneous architecture such as client-server.

Second, some instructors who adopted the Third Edition have told me that they were uncomfortable with a separate planning and estimating phase between the specification phase and the design phase. They agreed that accurate estimates of cost and duration are not possible until the specifications are complete (although sometimes we are required to produce estimates earlier in the life cycle). However, they felt that these planning and estimating activities did not merit a complete phase, particularly because they comprise only about 1 percent of the total software life cycle. Accordingly, I have dropped the separate planning phase and incorporated these activities at the end of the specifications phase. The various planning activities that are performed are described in Chapter 8, entitled "Planning and Estimating." This material, too, may be delayed in order to start Part 2. In addition to these two chapters, certain sections of other chapters (such as Section 2.12) may also be deferred and taught in parallel with Part 2. All material that can be postponed in this way is marked with ♦.

Thus, an instructor who is teaching a one-semester (or one-quarter) sequence using the Fourth Edition covers most of Chapters 1 through 6, and then starts Part 2 (Chapters 9 through 15). Chapters 7 and 8 can then be taught in parallel with Part 2. When teaching the two-semester sequence, the chapters of the book are taught in order; the class is now fully prepared for the semester-long team-based Term Project.

In order to ensure that the key software engineering techniques of Part 2 are truly understood, each is presented twice. First, whenever a technique is introduced, it is illustrated by means of the elevator problem. The elevator problem is the correct size for the reader to be able to see the technique applied to a complete problem, and it has enough subtleties to highlight both the strengths and weaknesses of the technique being taught. Then, at the end of each chapter there is a new continuing major Case Study. A detailed solution to the Case Study is presented. The solution to each phase of the Case Study is generally too large to appear in the chapter itself. Instead, only key points of the solution are presented in the chapter, and the complete material appears at the end of the book (Appendices C through I). The rapid prototype and detailed C++ implementations are available via the World-Wide Web at <http://www.mhhe.com/engcs/compisci/schach>.

THE PROBLEM SETS

As in the previous edition, there are four types of problems. First, at the end of each chapter there are a number of exercises intended to highlight key points. These exercises are self-contained; the technical information for all of the exercises can be found in this book.

Second, there is a major Term Project. It is designed to be solved by students working in teams of three, the smallest number of team members that cannot confer over a standard telephone. The Term Project comprises 15 separate components, each tied to the relevant chapter. For example, design is the topic of Chapter 12, so in that chapter the component of the Term Project is concerned with designing the software for the project. By breaking a large project into smaller, well-defined pieces, the instructor can monitor the progress of the class more closely. The structure of the Term Project is such that instructors may freely apply the 15 components to any other project they choose.

Because this book is written for use by graduate students as well as upper-class undergraduates, the third type of problem is based on research papers in the software engineering literature. In each chapter an important paper has been chosen; wherever possible, a paper related to object-oriented software engineering has been selected. The student is asked to read the paper and to answer a question relating to its contents. Of course, the instructor is free to assign any other research paper; the "For Further Reading" section at the end of each chapter includes a wide variety of relevant papers.

The fourth type of problem relates to the Case Study. This type of problem was introduced in the Third Edition in response to instructors who told me that they believe their students learn more by modifying an existing product than by developing a product from scratch. Many senior software engineers in the industry agreed with that viewpoint. Accordingly, each chapter in which the Case Study is presented has at least three problems that require the student to modify the Case Study in some way. For example, in one chapter the student is asked to redesign the Case Study using a different design technique than the one used for the Case Study. In another chapter, the student is asked what the effect would have been of performing the steps of the object-oriented analysis in a different order. To make it easy to modify the source code of the Case Study, it is readily available as described at the end of the previous section.

The *Instructor's Solution Manual*, available from McGraw-Hill, contains detailed solutions to all the exercises, as well as to the Term Project. In addition, the *Instructor's Solution Manual* contains transparency masters for all the figures in this book. The transparency masters can also be downloaded from www.mhhe.com/engcs/compsci/schach.

MHHE 10

ACKNOWLEDGMENTS

I am indebted to those who reviewed this edition, including:

Thaddeus R. Crews, Jr., Western Kentucky University
Eduardo B. Fernandez, Florida Atlantic University
Michael Godfrey, Cornell University
Thomas B. Horton, Florida Atlantic University
Gail Kaiser, Columbia University
Laxmikant V. Kale, University of Illinois
Chung Lee, California State Polytechnic University at Pomona
Susan Mengel, Texas Tech University
David S. Rosenblum, University of California at Irvine
Shmuel Rotenstreich, George Washington University
Wendel Scarbrough, Azusa Pacific University
Gerald B. Sheble, Iowa State

I am particularly grateful to two of the reviewers. Thad Crews made many creative pedagogic suggestions. As a consequence, it is easier to teach from this book and to learn from it. Laxmikant Kale pointed out a number of weaknesses. I am grateful to him for his meticulous reading of the entire manuscript.

I should like to thank three individuals who have also made contributions to earlier books. First, Jeff Gray has once again made numerous insightful suggestions. In particular, I am grateful for his many ideas regarding Chapter 7. Also, he is once again a coauthor of the *Instructor's Solution Manual*. Second, my son David has made a number of helpful contributions to the book and is also a coauthor of the *Instructor's Solution Manual*. Third, I thank Saveen Reddy for drawing my attention to the quotation from Marcus Aurelius that appears in the last Just in Case You Wanted to Know box.

With regard to my publishers, McGraw-Hill, I am especially grateful to executive editor Betsy Jones, sponsoring editor Brad Kosirog, and project manager Paula Buschman.

Finally, as always, I thank my family for their continual support. When I started writing books, my limited free time had to be shared between my family and my current book project. Now that my children are assisting with my books, writing has become a family activity. For the seventh time, it is my privilege to dedicate this book to my wife, Sharon, and my children, David and Lauren, with love.

Stephen R. Schach

BRIEF CONTENTS

Preface vii

PART 1

Introduction to the Software
Life Cycle 1

CHAPTER 1
Scope of Software Engineering 3

CHAPTER 2
The Software Process 30

CHAPTER 3
Software Life-Cycle Models 64

CHAPTER 4
Teams and the Tools
of Their Trade 90

CHAPTER 5
Testing 134

CHAPTER 6
Introduction to Objects 171

CHAPTER 7
Reusability, Portability,
and Interoperability 217

CHAPTER 8
Planning and Estimating 262

PART 2

The Phases of the Software
Life Cycle 299

CHAPTER 9
Requirements Phase 301

CHAPTER 10
Specification Phase 329

CHAPTER 11
Object-Oriented Analysis
Phase 376

CHAPTER 12
Design Phase 403

CHAPTER 13
Implementation Phase 437

CHAPTER 14
Implementation and
Integration Phase 479

CHAPTER 15
Maintenance Phase 502

APPENDIX A
Air Gourmet 523

APPENDIX B
Software Engineering
Resources 526

CONTENTS

Preface vii

PART 1

Introduction to the Software Life Cycle 1

CHAPTER 1 Scope of Software Engineering 3

- 1.1 Historical Aspects 5
- 1.2 Economic Aspects 7
- 1.3 Maintenance Aspects 8
- 1.4 Specification and Design Aspects 12
- 1.5 Team Programming Aspects 15
- 1.6 The Object-Oriented Paradigm 16
- 1.7 Terminology 21
- Chapter Review 23
- For Further Reading 24
- Problems 25
- References 26

CHAPTER 2 The Software Process 30

- 2.1 Client, Developer, and User 32
- 2.2 Requirements Phase 33
 - 2.2.1 Requirements Phase Testing 34
- 2.3 Specification Phase 35
 - 2.3.1 Specification Phase Testing 37
- 2.4 Design Phase 37
 - 2.4.1 Design Phase Testing 39
- 2.5 Implementation Phase 39
 - 2.5.1 Implementation Phase Testing 39
- 2.6 Integration Phase 40
 - 2.6.1 Integration Phase Testing 40
- 2.7 Maintenance Phase 41
 - 2.7.1 Maintenance Phase Testing 41
- 2.8 Retirement 42

2.9 Problems with Software Production: Essence and Accidents 43

- 2.9.1 Complexity 44
- 2.9.2 Conformity 46
- 2.9.3 Changeability 47
- 2.9.4 Invisibility 48
- 2.9.5 No Silver Bullet? 49
- 2.10 Improving the Software Process 50
- 2.11 Capability Maturity Models 50
- 2.12 ISO 9000 54
- 2.13 SPICE 55
- 2.14 Costs and Benefits of Software Process Improvement 56
- Chapter Review 57
- For Further Reading 58
- Problems 59
- References 60

CHAPTER 3 Software Life-Cycle Models 64

- 3.1 Build-and-Fix Model 64
- 3.2 Waterfall Model 65
 - 3.2.1 Analysis of the Waterfall Model 68
- 3.3 Rapid Prototyping Model 70
 - 3.3.1 Integrating the Waterfall and Rapid Prototyping Models 72
- 3.4 Incremental Model 72
 - 3.4.1 Analysis of the Incremental Model 74
- 3.5 Synchronize-and-Stabilize Model 77
- 3.6 Spiral Model 77
 - 3.6.1 Analysis of the Spiral Model 82
- 3.7 Object-Oriented Life-Cycle Models 83
- 3.8 Comparison of Life-Cycle Models 84
- Chapter Review 86
- For Further Reading 86
- Problems 87
- References 88

CHAPTER 4 Teams and the Tools of Their Trade 90

- 4.1 Team Organization 90
- 4.2 Democratic Team Approach 92
 - 4.2.1 Analysis of the Democratic Team Approach 93
- 4.3 Classical Chief Programmer Team Approach 94
 - 4.3.1 The *New York Times* Project 96
 - 4.3.2 Impracticality of the Classical Chief Programmer Team Approach 96
- 4.4 Beyond Chief Programmer and Democratic Teams 97
- 4.5 Synchronize-and-Stabilize Teams 101
- 4.6 Stepwise Refinement 102
 - 4.6.1 Stepwise Refinement Example 103
- 4.7 Cost-Benefit Analysis 109
- 4.8 Software Metrics 110
- 4.9 CASE 112
- 4.10 Taxonomy of CASE 113
- 4.11 Scope of CASE 114
- 4.12 Software Versions 119
 - 4.12.1 Revisions 119
 - 4.12.2 Variations 120
- 4.13 Configuration Control 120
 - 4.13.1 Configuration Control during Product Maintenance 123
 - 4.13.2 Baselines 124
 - 4.13.3 Configuration Control during Product Development 124
- 4.14 Build Tools 125
- 4.15 Productivity Gains with CASE Technology 126
- Chapter Review 128
- For Further Reading 128
- Problems 129
- References 131

CHAPTER 5 Testing 134

- 5.1 Quality Issues 135
 - 5.1.1 Software Quality Assurance 135
 - 5.1.2 Managerial Independence 136

- 5.2 Nonexecution-Based Testing 137
 - 5.2.1 Walkthroughs 137
 - 5.2.2 Managing Walkthroughs 138
 - 5.2.3 Inspections 139
 - 5.2.4 Comparison of Inspections and Walkthroughs 141
 - 5.2.5 Strengths and Weaknesses of Reviews 142
 - 5.2.6 Metrics for Inspections 142
- 5.3 Execution-Based Testing 143
- 5.4 What Should Be Tested? 143
 - 5.4.1 Utility 144
 - 5.4.2 Reliability 145
 - 5.4.3 Robustness 145
 - 5.4.4 Performance 146
 - 5.4.5 Correctness 146
- 5.5 Testing versus Correctness Proofs 148
 - 5.5.1 Example of a Correctness Proof 148
 - 5.5.2 Correctness Proof Case Study 152
 - 5.5.3 Correctness Proofs and Software Engineering 154
- 5.6 Who Should Perform Execution-Based Testing? 157
- 5.7 Testing Distributed Software 158
- 5.8 Testing Real-Time Software 160
- 5.9 When Testing Stops 162
- Chapter Review 163
- For Further Reading 163
- Problems 165
- References 166

CHAPTER 6 Introduction to Objects 171

- 6.1 What Is a Module? 171
- 6.2 Cohesion 175
 - 6.2.1 Coincidental Cohesion 175
 - 6.2.2 Logical Cohesion 176
 - 6.2.3 Temporal Cohesion 178
 - 6.2.4 Procedural Cohesion 178
 - 6.2.5 Communicational Cohesion 178
 - 6.2.6 Informational Cohesion 179
 - 6.2.7 Functional Cohesion 180
 - 6.2.8 Cohesion Example 180
- 6.3 Coupling 181
 - 6.3.1 Content Coupling 182
 - 6.3.2 Common Coupling 182

6.3.3	Control Coupling	184
6.3.4	Stamp Coupling	185
6.3.5	Data Coupling	186
6.3.6	Coupling Example	186
6.3.7	The Importance of Coupling	188
6.4	Data Encapsulation	189
6.4.1	Data Encapsulation and Product Development	192
6.4.2	Data Encapsulation and Product Maintenance	194
6.5	Abstract Data Types	198
6.6	Information Hiding	201
6.7	Objects	203
6.8	Inheritance, Polymorphism, and Dynamic Binding	207
6.9	Cohesion and Coupling of Objects	209
	Chapter Review	210
	For Further Reading	210
	Problems	211
	References	213

CHAPTER 7

Reusability, Portability, and Interoperability 217

7.1	Reuse Concepts	217
7.2	Impediments to Reuse	219
7.3	Reuse Case Studies	220
7.3.1	Raytheon Missile Systems Division	220
7.3.2	Toshiba Software Factory	222
7.3.3	NASA Software	223
7.3.4	GTE Data Services	224
7.3.5	Hewlett-Packard	224
7.3.6	European Space Agency	225
7.4	Objects and Productivity	226
7.5	Reuse during the Design and Implementation Phases	228
7.5.1	Module Reuse	228
7.5.2	Application Frameworks	229
7.5.3	Design Patterns	230
7.5.4	Software Architecture	235
7.6	Reuse and Maintenance	235
7.7	Portability	236
7.7.1	Hardware Incompatibilities	237
7.7.2	Operating System Incompatibilities	238

7.7.3	Numerical Software Incompatibilities	239
7.7.4	Compiler Incompatibilities	239
7.8	Why Portability?	245
7.9	Techniques for Achieving Portability	246
7.9.1	Portable System Software	246
7.9.2	Portable Application Software	247
7.9.3	Portable Data	248
7.10	Interoperability	249
7.10.1	OLE, COM, and ActiveX	250
7.10.2	CORBA	251
7.10.3	Comparing OLE/COM and CORBA	251
7.11	Future Trends in Interoperability	252
	Chapter Review	252
	For Further Reading	253
	Problems	254
	References	256

CHAPTER 8

Planning and Estimating 262

8.1	Planning and the Software Process	262
8.2	Estimating Duration and Cost	264
8.2.1	Metrics for the Size of a Product	265
8.2.2	Techniques of Cost Estimation	270
8.2.3	Intermediate COCOMO	273
8.2.4	COCOMO II	276
8.2.5	Tracking Duration and Cost Estimates	277
8.3	Components of a Software Project Management Plan	278
8.4	Software Project Management Plan Framework	280
8.5	IEEE Software Project Management Plan	281
8.6	Planning of Testing	284
8.7	Planning of Object-Oriented Projects	285
8.8	Training Requirements	286
8.9	Documentation Standards	287
8.10	CASE Tools for Planning and Estimating	288

- 8.11 Testing the Software Project Management Plan 291
- Chapter Review 291
- For Further Reading 291
- Problems 293
- References 294

PART 2

The Phases of the Software Life Cycle 299

CHAPTER 9 Requirements Phase 301

- 9.1 Requirements Analysis Techniques 302
- 9.2 Rapid Prototyping 303
- 9.3 Human Factors 305
- 9.4 Rapid Prototyping as a Specification Technique 307
- 9.5 Reusing the Rapid Prototype 309
- 9.6 Other Uses of Rapid Prototyping 311
- 9.7 Management Implications of the Rapid Prototyping Model 312
- 9.8 Experiences with Rapid Prototyping 313
- 9.9 Joint Application Design (JAD) 315
- 9.10 Comparison of Requirements Analysis Techniques 315
- 9.11 Testing during the Requirements Phase 316
- 9.12 CASE Tools for the Requirements Phase 316
- 9.13 Metrics for the Requirements Phase 318
- 9.14 Osbert Oglesby Case Study: Requirements Phase 318
- 9.15 Osbert Oglesby Case Study: Rapid Prototype 321
- 9.16 Object-Oriented Requirements? 324
- Chapter Review 324
- For Further Reading 325
- Problems 326
- References 327

CHAPTER 10 Specification Phase 329

- 10.1 The Specification Document 329
- 10.2 Informal Specifications 331
 - 10.2.1 Case Study: Text Processing 332
- 10.3 Structured Systems Analysis 333
 - 10.3.1 Sally's Software Shop 333
- 10.4 Other Semiformal Techniques 341
- 10.5 Entity-Relationship Modeling 342
- 10.6 Finite State Machines 344
 - 10.6.1 Elevator Problem: Finite State Machines 346
- 10.7 Petri Nets 351
 - 10.7.1 Elevator Problem: Petri Nets 355
- 10.8 Z 357
 - 10.8.1 Elevator Problem: Z 358
 - 10.8.2 Analysis of Z 360
- 10.9 Other Formal Techniques 362
- 10.10 Comparison of Specification Techniques 363
- 10.11 Testing during the Specification Phase 364
- 10.12 CASE Tools for the Specification Phase 365
- 10.13 Metrics for the Specification Phase 366
- 10.14 Osbert Oglesby Case Study: Structured Systems Analysis 366
- 10.15 Osbert Oglesby Case Study: Software Project Management Plan 367
- Chapter Review 367
- For Further Reading 368
- Problems 369
- References 372

CHAPTER 11 Object-Oriented Analysis Phase 376

- 11.1 Object-Oriented versus Structured Paradigm 376
- 11.2 Object-Oriented Analysis 378

- 11.3 Elevator Problem: Object-Oriented Analysis 380
- 11.4 Use-Case Modeling 381
- 11.5 Class Modeling 382
 - 11.5.1 Noun Extraction 382
 - 11.5.2 CRC Cards 385
- 11.6 Dynamic Modeling 387
- 11.7 Testing during the Object-Oriented Analysis Phase 388
- 11.8 CASE Tools for the Object-Oriented Analysis Phase 391
- 11.9 Osbert Oglesby Case Study: Object-Oriented Analysis 393
- 11.10 Osbert Oglesby Case Study: Software Project Management Plan 398
- Chapter Review 399
- For Further Reading 399
- Problems 400
- References 401

CHAPTER 12 Design Phase 403

- 12.1 Design and Abstraction 403
- 12.2 Action-Oriented Design 404
- 12.3 Data Flow Analysis 405
 - 12.3.1 Data Flow Analysis Example 406
 - 12.3.2 Extensions 411
- 12.4 Transaction Analysis 411
- 12.5 Data-Oriented Design 413
- 12.6 Object-Oriented Design 414
- 12.7 Elevator Problem: Object-Oriented Design 414
- 12.8 Formal Techniques for Detailed Design 420
- 12.9 Real-Time Design Techniques 422
- 12.10 Testing during the Design Phase 423
- 12.11 CASE Tools for the Design Phase 424
- 12.12 Metrics for the Design Phase 425
- 12.13 Osbert Oglesby Case Study: Object-Oriented Design 426
- Chapter Review 431
- For Further Reading 431

- Problems 432
- References 433

CHAPTER 13 Implementation Phase 437

- 13.1 Choice of Programming Language 437
- 13.2 Fourth Generation Languages 440
- 13.3 Good Programming Practice 443
- 13.4 Coding Standards 449
- 13.5 Module Reuse 450
- 13.6 Module Test Case Selection 451
 - 13.6.1 Testing to Specifications versus Testing to Code 451
 - 13.6.2 Feasibility of Testing to Specifications 451
 - 13.6.3 Feasibility of Testing to Code 452
- 13.7 Black-Box Module-Testing Techniques 455
 - 13.7.1 Equivalence Testing and Boundary Value Analysis 455
 - 13.7.2 Functional Testing 456
- 13.8 Glass-Box Module-Testing Techniques 457
 - 13.8.1 Structural Testing: Statement, Branch, and Path Coverage 458
 - 13.8.2 Complexity Metrics 459
- 13.9 Code Walkthroughs and Inspections 462
- 13.10 Comparison of Module-Testing Techniques 462
- 13.11 Cleanroom 463
- 13.12 Potential Problems When Testing Objects 464
- 13.13 Management Aspects of Module Testing 467
- 13.14 When to Rewrite Rather than Debug a Module 467
- 13.15 CASE Tools for the Implementation Phase 469
- 13.16 Osbert Oglesby Case Study: Black-Box Test Cases 469
- Chapter Review 471
- For Further Reading 471
- Problems 472
- References 474

CHAPTER 14 **Implementation and** **Integration Phase 479**

- 14.1 Implementation and Integration 479
 - 14.1.1 Top-Down Implementation and Integration 480
 - 14.1.2 Bottom-Up Implementation and Integration 482
 - 14.1.3 Sandwich Implementation and Integration 483
 - 14.1.4 Implementation and Integration of Object-Oriented Products 485
 - 14.1.5 Management Issues during the Implementation and Integration Phase 485
- 14.2 Testing during the Implementation and Integration Phase 486
- 14.3 Integration Testing of Graphical User Interfaces 486
- 14.4 Product Testing 487
- 14.5 Acceptance Testing 488
- 14.6 CASE Tools for the Implementation and Integration Phase 489
- 14.7 CASE Tools for the Complete Software Process 490
- 14.8 Integrated Environments 490
 - 14.8.1 Process Integration 490
 - 14.8.2 Tool Integration 491
 - 14.8.3 Other Forms of Integration 494
- 14.9 Environments for Business Applications 494
- 14.10 Public Tool Infrastructures 495
- 14.11 Potential Problems with Environments 495
- 14.12 Metrics for the Implementation and Integration Phase 496
- 14.13 Osbert Oglesby Case Study: Implementation and Integration Phase 497
- Chapter Review 498
- For Further Reading 498
- Problems 499
- References 500

CHAPTER 15 **Maintenance Phase 502**

- 15.1 Why Maintenance Is Necessary 502
- 15.2 What Is Required of Maintenance Programmers 503
- 15.3 Maintenance Case Study 505
- 15.4 Management of Maintenance 507
 - 15.4.1 Fault Reports 507
 - 15.4.2 Authorizing Changes to the Product 508
 - 15.4.3 Ensuring Maintainability 509
 - 15.4.4 Problem of Repeated Maintenance 509
- 15.5 Maintenance of Object-Oriented Software 510
- 15.6 Maintenance Skills versus Development Skills 514
- 15.7 Reverse Engineering 514
- 15.8 Testing during the Maintenance Phase 515
- 15.9 CASE Tools for the Maintenance Phase 516
- 15.10 Metrics for the Maintenance Phase 517
- 15.11 Osbert Oglesby Case Study: Maintenance 517
- Chapter Review 518
- For Further Reading 519
- Problems 519
- References 520

APPENDIX A **Air Gourmet 523**

APPENDIX B **Software Engineering** **Resources 526**

APPENDIX C **Osbert Oglesby Case Study:** **Rapid Prototype 529**

APPENDIX D Osbert Oglesby Case Study: Structured Systems Analysis 530	APPENDIX H Osbert Oglesby Case Study: Black-Box Test Cases 559
APPENDIX E Osbert Oglesby Case Study: Object-Oriented Analysis 534	APPENDIX I Osbert Oglesby Case Study: Complete Source Code 563
APPENDIX F Osbert Oglesby Case Study: Software Project Management Plan 535	Bibliography 564
APPENDIX G Osbert Oglesby Case Study: Design 540	Author Index 597
	Subject Index 603