

# **Software Engineering**

## **A Practitioner's Approach**

**Seventh Edition**



**Roger S. Pressman**

# Software Engineering

A PRACTITIONER'S APPROACH



# **Software Engineering**

**A PRACTITIONER'S APPROACH**

**SEVENTH EDITION**

**Roger S. Pressman, Ph.D.**



Boston Burr Ridge, IL Dubuque, IA New York San Francisco St. Louis  
Bangkok Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City  
Milan Montreal New Delhi Santiago Seoul Singapore Sydney Taipei Toronto



SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH, SEVENTH EDITION

Published by McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020. Copyright © 2010 by The McGraw-Hill Companies, Inc. All rights reserved. Previous editions © 2005, 2001, and 1997. 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 consent of The McGraw-Hill Companies, Inc., including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 DOC/DOC 0 9

ISBN 978-0-07-337597-7

MHID 0-07-337597-7

Global Publisher: *Raghothaman Srinivasan*

Director of Development: *Kristine Tibbetts*

Senior Marketing Manager: *Curt Reynolds*

Senior Managing Editor: *Faye M. Schilling*

Lead Production Supervisor: *Sandy Ludovissy*

Senior Media Project Manager: *Sandra M. Schnee*

Associate Design Coordinator: *Brenda A. Rohlves*

Cover Designer: *Studio Montage, St. Louis, Missouri*

(USE) Cover Image: © *The Studio Dog/Getty Images*

Compositor: *Macmillan Publishing Solutions*

Typeface: *8.5/13.5 Leawood*

Printer: *R. R. Donnelley Crawfordsville, IN*

**Library of Congress Cataloging-in-Publication Data**

Pressman, Roger S.

Software engineering : a practitioner's approach / Roger S. Pressman. — 7th ed.

p. cm.

Includes index.

ISBN 978-0-07-337597-7 — ISBN 0-07-337597-7 (hard copy : alk. paper)

1. Software engineering. I. Title.

QA76.758.P75 2010

005.1—dc22

2008048802

*In loving memory of my  
father who lived 94 years  
and taught me, above all,  
that honesty and integrity  
were the best guides for  
my journey through life.*

# ABOUT THE AUTHOR

**R**oger S. Pressman is an internationally recognized authority in software process improvement and software engineering technologies. For almost four decades, he has worked as a software engineer, a manager, a professor, an author, and a consultant, focusing on software engineering issues.

As an industry practitioner and manager, Dr. Pressman worked on the development of CAD/CAM systems for advanced engineering and manufacturing applications. He has also held positions with responsibility for scientific and systems programming.

After receiving a Ph.D. in engineering from the University of Connecticut, Dr. Pressman moved to academia where he became Bullard Associate Professor of Computer Engineering at the University of Bridgeport and director of the university's Computer-Aided Design and Manufacturing Center.

Dr. Pressman is currently president of R.S. Pressman & Associates, Inc., a consulting firm specializing in software engineering methods and training. He serves as principal consultant and has designed and developed *Essential Software Engineering*, a complete video curriculum in software engineering, and *Process Advisor*, a self-directed system for software process improvement. Both products are used by thousands of companies worldwide. More recently, he has worked in collaboration with *EdistaLearning* in India to develop comprehensive Internet-based training in software engineering.

Dr. Pressman has written many technical papers, is a regular contributor to industry periodicals, and is author of seven technical books. In addition to *Software Engineering: A Practitioner's Approach*, he has co-authored *Web Engineering* (McGraw-Hill), one of the first books to apply a tailored set of software engineering principles and practices to the development of Web-based systems and applications. He has also written the award-winning *A Manager's Guide to Software Engineering* (McGraw-Hill); *Making Software Engineering Happen* (Prentice Hall), the first book to address the critical management problems associated with software process improvement; and *Software Shock* (Dorset House), a treatment that focuses on software and its impact on business and society. Dr. Pressman has been on the editorial boards of a number of industry journals, and for many years, was editor of the "Manager" column in *IEEE Software*.

Dr. Pressman is a well-known speaker, keynoting a number of major industry conferences. He is a member of the IEEE, and Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu, and Pi Tau Sigma.

On the personal side, Dr. Pressman lives in South Florida with his wife, Barbara. An athlete for most of his life, he remains a serious tennis player (NTRP 4.5) and a single-digit handicap golfer. In his spare time, he has written two novels, *The Aymara Bridge* and *The Puppeteer*, and plans to begin work on another.

# CONTENTS AT A GLANCE

CHAPTER 1 Software and Software Engineering 1

---

**PART ONE THE SOFTWARE PROCESS 29**

CHAPTER 2 Process Models 30

CHAPTER 3 Agile Development 65

---

**PART TWO MODELING 95**

CHAPTER 4 Principles that Guide Practice 96

CHAPTER 5 Understanding Requirements 119

CHAPTER 6 Requirements Modeling: Scenarios, Information, and Analysis Classes 148

CHAPTER 7 Requirements Modeling: Flow, Behavior, Patterns, and WebApps 186

CHAPTER 8 Design Concepts 215

CHAPTER 9 Architectural Design 242

CHAPTER 10 Component-Level Design 276

CHAPTER 11 User Interface Design 312

CHAPTER 12 Pattern-Based Design 347

CHAPTER 13 WebApp Design 373

---

**PART THREE QUALITY MANAGEMENT 397**

CHAPTER 14 Quality Concepts 398

CHAPTER 15 Review Techniques 416

CHAPTER 16 Software Quality Assurance 432

CHAPTER 17 Software Testing Strategies 449

CHAPTER 18 Testing Conventional Applications 481

CHAPTER 19 Testing Object-Oriented Applications 511

CHAPTER 20 Testing Web Applications 529

CHAPTER 21 Formal Modeling and Verification 557

CHAPTER 22 Software Configuration Management 584

CHAPTER 23 Product Metrics 613

---

**PART FOUR MANAGING SOFTWARE PROJECTS 645**

CHAPTER 24 Project Management Concepts 646

CHAPTER 25 Process and Project Metrics 666

CHAPTER 26	Estimation for Software Projects	691
CHAPTER 27	Project Scheduling	721
CHAPTER 28	Risk Management	744
CHAPTER 29	Maintenance and Reengineering	761

**PART FIVE****ADVANCED TOPICS** 785

---

CHAPTER 30	Software Process Improvement	786
CHAPTER 31	Emerging Trends in Software Engineering	808
CHAPTER 32	Concluding Comments	833
APPENDIX 1	An Introduction to UML	841
APPENDIX 2	Object-Oriented Concepts	863
REFERENCES		871
INDEX		889

# TABLE OF CONTENTS

*Preface* xxv

---

## CHAPTER 1 SOFTWARE AND SOFTWARE ENGINEERING 1

---

1.1	The Nature of Software	3
1.1.1	Defining Software	4
1.1.2	Software Application Domains	7
1.1.3	Legacy Software	9
1.2	The Unique Nature of WebApps	10
1.3	Software Engineering	12
1.4	The Software Process	14
1.5	Software Engineering Practice	17
1.5.1	The Essence of Practice	17
1.5.2	General Principles	19
1.6	Software Myths	21
1.7	How It All Starts	24
1.8	Summary	25
PROBLEMS AND POINTS TO PONDER		25
FURTHER READINGS AND INFORMATION SOURCES		26

---

## PART ONE THE SOFTWARE PROCESS 29

---

## CHAPTER 2 PROCESS MODELS 30

---

2.1	A Generic Process Model	31
2.1.1	Defining a Framework Activity	32
2.1.2	Identifying a Task Set	34
2.1.3	Process Patterns	35
2.2	Process Assessment and Improvement	37
2.3	Prescriptive Process Models	38
2.3.1	The Waterfall Model	39
2.3.2	Incremental Process Models	41
2.3.3	Evolutionary Process Models	42
2.3.4	Concurrent Models	48
2.3.5	A Final Word on Evolutionary Processes	49
2.4	Specialized Process Models	50
2.4.1	Component-Based Development	50
2.4.2	The Formal Methods Model	51
2.4.3	Aspect-Oriented Software Development	52
2.5	The Unified Process	53
2.5.1	A Brief History	54
2.5.2	Phases of the Unified Process	54
2.6	Personal and Team Process Models	56
2.6.1	Personal Software Process (PSP)	57
2.6.2	Team Software Process (TSP)	58
2.7	Process Technology	59
2.8	Product and Process	60

**TABLE OF CONTENTS**

2.9	Summary	61
PROBLEMS AND POINTS TO PONDER		62
FURTHER READINGS AND INFORMATION SOURCES		63

**CHAPTER 3 AGILE DEVELOPMENT 65**

---

3.1	What Is Agility?	67
3.2	Agility and the Cost of Change	67
3.3	What Is an Agile Process?	68
3.3.1	Agility Principles	69
3.3.2	The Politics of Agile Development	70
3.3.3	Human Factors	71
3.4	Extreme Programming (XP)	72
3.4.1	XP Values	72
3.4.2	The XP Process	73
3.4.3	Industrial XP	77
3.4.4	The XP Debate	78
3.5	Other Agile Process Models	80
3.5.1	Adaptive Software Development (ASD)	81
3.5.2	Scrum	82
3.5.3	Dynamic Systems Development Method (DSDM)	84
3.5.4	Crystal	85
3.5.5	Feature Driven Development (FDD)	86
3.5.6	Lean Software Development (LSD)	87
3.5.7	Agile Modeling (AM)	88
3.5.8	Agile Unified Process (AUP)	89
3.6	A Tool Set for the Agile Process	91
3.7	Summary	91
PROBLEMS AND POINTS TO PONDER		92
FURTHER READINGS AND INFORMATION SOURCES		93

**PART TWO MODELING 95**

---

**CHAPTER 4 PRINCIPLES THAT GUIDE PRACTICE 96**

---

4.1	Software Engineering Knowledge	97
4.2	Core Principles	98
4.2.1	Principles That Guide Process	98
4.2.2	Principles That Guide Practice	99
4.3	Principles That Guide Each Framework Activity	101
4.3.1	Communication Principles	101
4.3.2	Planning Principles	103
4.3.3	Modeling Principles	105
4.3.4	Construction Principles	111
4.3.5	Deployment Principles	113
4.4	Summary	115
PROBLEMS AND POINTS TO PONDER		116
FURTHER READINGS AND INFORMATION SOURCES		116

**CHAPTER 5 UNDERSTANDING REQUIREMENTS 119**

---

5.1	Requirements Engineering	120
5.2	Establishing the Groundwork	125
5.2.1	Identifying Stakeholders	125

5.2.2	Recognizing Multiple Viewpoints	126
5.2.3	Working toward Collaboration	126
5.2.4	Asking the First Questions	127
5.3	Eliciting Requirements	128
5.3.1	Collaborative Requirements Gathering	128
5.3.2	Quality Function Deployment	131
5.3.3	Usage Scenarios	132
5.3.4	Elicitation Work Products	133
5.4	Developing Use Cases	133
5.5	Building the Requirements Model	138
5.5.1	Elements of the Requirements Model	139
5.5.2	Analysis Patterns	142
5.6	Negotiating Requirements	142
5.7	Validating Requirements	144
5.8	Summary	145
PROBLEMS AND POINTS TO PONDER 145		
FURTHER READINGS AND INFORMATION SOURCES 146		

---

## CHAPTER 6 REQUIREMENTS MODELING: SCENARIOS, INFORMATION, AND ANALYSIS CLASSES 148

6.1	Requirements Analysis	149
6.1.1	Overall Objectives and Philosophy	150
6.1.2	Analysis Rules of Thumb	151
6.1.3	Domain Analysis	151
6.1.4	Requirements Modeling Approaches	153
6.2	Scenario-Based Modeling	154
6.2.1	Creating a Preliminary Use Case	155
6.2.2	Refining a Preliminary Use Case	158
6.2.3	Writing a Formal Use Case	159
6.3	UML Models That Supplement the Use Case	161
6.3.1	Developing an Activity Diagram	161
6.3.2	Swimlane Diagrams	162
6.4	Data Modeling Concepts	164
6.4.1	Data Objects	164
6.4.2	Data Attributes	164
6.4.3	Relationships	165
6.5	Class-Based Modeling	167
6.5.1	Identifying Analysis Classes	167
6.5.2	Specifying Attributes	171
6.5.3	Defining Operations	171
6.5.4	Class-Responsibility-Collaborator (CRC) Modeling	173
6.5.5	Associations and Dependencies	180
6.5.6	Analysis Packages	182
6.6	Summary	183
PROBLEMS AND POINTS TO PONDER 183		
FURTHER READINGS AND INFORMATION SOURCES 184		

---

## CHAPTER 7 REQUIREMENTS MODELING: FLOW, BEHAVIOR, PATTERNS, AND WEBAPPS 186

7.1	Requirements Modeling Strategies	186
7.2	Flow-Oriented Modeling	187

7.2.1	Creating a Data Flow Model	188
7.2.2	Creating a Control Flow Model	191
7.2.3	The Control Specification	191
7.2.4	The Process Specification	192
7.3	Creating a Behavioral Model	195
7.3.1	Identifying Events with the Use Case	195
7.3.2	State Representations	196
7.4	Patterns for Requirements Modeling	199
7.4.1	Discovering Analysis Patterns	200
7.4.2	A Requirements Pattern Example: Actuator-Sensor	200
7.5	Requirements Modeling for WebApps	205
7.5.1	How Much Analysis Is Enough?	205
7.5.2	Requirements Modeling Input	206
7.5.3	Requirements Modeling Output	207
7.5.4	Content Model for WebApps	207
7.5.5	Interaction Model for WebApps	209
7.5.6	Functional Model for WebApps	210
7.5.7	Configuration Models for WebApps	211
7.5.8	Navigation Modeling	212
7.6	Summary	213
PROBLEMS AND POINTS TO PONDER 213		
FURTHER READINGS AND INFORMATION SOURCES 214		

## CHAPTER 8 DESIGN CONCEPTS 215

---

8.1	Design within the Context of Software Engineering	216
8.2	The Design Process	219
8.2.1	Software Quality Guidelines and Attributes	219
8.2.2	The Evolution of Software Design	221
8.3	Design Concepts	222
8.3.1	Abstraction	223
8.3.2	Architecture	223
8.3.3	Patterns	224
8.3.4	Separation of Concerns	225
8.3.5	Modularity	225
8.3.6	Information Hiding	226
8.3.7	Functional Independence	227
8.3.8	Refinement	228
8.3.9	Aspects	228
8.3.10	Refactoring	229
8.3.11	Object-Oriented Design Concepts	230
8.3.12	Design Classes	230
8.4	The Design Model	233
8.4.1	Data Design Elements	234
8.4.2	Architectural Design Elements	234
8.4.3	Interface Design Elements	235
8.4.4	Component-Level Design Elements	237
8.4.5	Deployment-Level Design Elements	237
8.5	Summary	239
PROBLEMS AND POINTS TO PONDER 240		
FURTHER READINGS AND INFORMATION SOURCES 240		

**CHAPTER 9 ARCHITECTURAL DESIGN 242**

---

9.1	Software Architecture 243
9.1.1	What Is Architecture? 243
9.1.2	Why Is Architecture Important? 245
9.1.3	Architectural Descriptions 245
9.1.4	Architectural Decisions 246
9.2	Architectural Genres 246
9.3	Architectural Styles 249
9.3.1	A Brief Taxonomy of Architectural Styles 250
9.3.2	Architectural Patterns 253
9.3.3	Organization and Refinement 255
9.4	Architectural Design 255
9.4.1	Representing the System in Context 256
9.4.2	Defining Archetypes 257
9.4.3	Refining the Architecture into Components 258
9.4.4	Describing Instantiations of the System 260
9.5	Assessing Alternative Architectural Designs 261
9.5.1	An Architecture Trade-Off Analysis Method 262
9.5.2	Architectural Complexity 263
9.5.3	Architectural Description Languages 264
9.6	Architectural Mapping Using Data Flow 265
9.6.1	Transform Mapping 265
9.6.2	Refining the Architectural Design 272
9.7	Summary 273
	PROBLEMS AND POINTS TO PONDER 274
	FURTHER READINGS AND INFORMATION SOURCES 274

**CHAPTER 10 COMPONENT-LEVEL DESIGN 276**

---

10.1	What Is a Component? 277
10.1.1	An Object-Oriented View 277
10.1.2	The Traditional View 279
10.1.3	A Process-Related View 281
10.2	Designing Class-Based Components 282
10.2.1	Basic Design Principles 282
10.2.2	Component-Level Design Guidelines 285
10.2.3	Cohesion 286
10.2.4	Coupling 288
10.3	Conducting Component-Level Design 290
10.4	Component-Level Design for WebApps 296
10.4.1	Content Design at the Component Level 297
10.4.2	Functional Design at the Component Level 297
10.5	Designing Traditional Components 298
10.5.1	Graphical Design Notation 299
10.5.2	Tabular Design Notation 300
10.5.3	Program Design Language 301
10.6	Component-Based Development 303
10.6.1	Domain Engineering 303
10.6.2	Component Qualification, Adaptation, and Composition 304
10.6.3	Analysis and Design for Reuse 306
10.6.4	Classifying and Retrieving Components 307

10.7	Summary	309
PROBLEMS AND POINTS TO PONDER		310
FURTHER READINGS AND INFORMATION SOURCES		311

---

**CHAPTER 11 USER INTERFACE DESIGN 312**

11.1	The Golden Rules	313
11.1.1	Place the User in Control	313
11.1.2	Reduce the User's Memory Load	314
11.1.3	Make the Interface Consistent	316
11.2	User Interface Analysis and Design	317
11.2.1	Interface Analysis and Design Models	317
11.2.2	The Process	319
11.3	Interface Analysis	320
11.3.1	User Analysis	321
11.3.2	Task Analysis and Modeling	322
11.3.3	Analysis of Display Content	327
11.3.4	Analysis of the Work Environment	328
11.4	Interface Design Steps	328
11.4.1	Applying Interface Design Steps	329
11.4.2	User Interface Design Patterns	330
11.4.3	Design Issues	331
11.5	WebApp Interface Design	335
11.5.1	Interface Design Principles and Guidelines	336
11.5.2	Interface Design Workflow for WebApps	340
11.6	Design Evaluation	342
11.7	Summary	344
PROBLEMS AND POINTS TO PONDER		345
FURTHER READINGS AND INFORMATION SOURCES		346

---

**CHAPTER 12 PATTERN-BASED DESIGN 347**

12.1	Design Patterns	348
12.1.1	Kinds of Patterns	349
12.1.2	Frameworks	352
12.1.3	Describing a Pattern	352
12.1.4	Pattern Languages and Repositories	353
12.2	Pattern-Based Software Design	354
12.2.1	Pattern-Based Design in Context	354
12.2.2	Thinking in Patterns	356
12.2.3	Design Tasks	357
12.2.4	Building a Pattern-Organizing Table	358
12.2.5	Common Design Mistakes	359
12.3	Architectural Patterns	360
12.4	Component-Level Design Patterns	362
12.5	User Interface Design Patterns	364
12.6	WebApp Design Patterns	368
12.6.1	Design Focus	368
12.6.2	Design Granularity	369
12.7	Summary	370
PROBLEMS AND POINTS TO PONDER		371
FURTHER READING AND INFORMATION SOURCES		372

**CHAPTER 13 WEBAPP DESIGN 373**

---

13.1	WebApp Design Quality	374
13.2	Design Goals	377
13.3	A Design Pyramid for WebApps	378
13.4	WebApp Interface Design	378
13.5	Aesthetic Design	380
13.5.1	Layout Issues	380
13.5.2	Graphic Design Issues	381
13.6	Content Design	382
13.6.1	Content Objects	382
13.6.2	Content Design Issues	382
13.7	Architecture Design	383
13.7.1	Content Architecture	384
13.7.2	WebApp Architecture	386
13.8	Navigation Design	388
13.8.1	Navigation Semantics	388
13.8.2	Navigation Syntax	389
13.9	Component-Level Design	390
13.10	Object-Oriented Hypermedia Design Method (OOHDM)	390
13.10.1	Conceptual Design for OOHDM	391
13.10.2	Navigational Design for OOHDM	391
13.10.3	Abstract Interface Design and Implementation	392
13.11	Summary	393
PROBLEMS AND POINTS TO PONDER 394		
FURTHER READINGS AND INFORMATION SOURCES 395		

**PART THREE QUALITY MANAGEMENT 397**

---

**CHAPTER 14 QUALITY CONCEPTS 398**

---

14.1	What Is Quality?	399
14.2	Software Quality	400
14.2.1	Garvin's Quality Dimensions	401
14.2.2	McCall's Quality Factors	402
14.2.3	ISO 9126 Quality Factors	403
14.2.4	Targeted Quality Factors	404
14.2.5	The Transition to a Quantitative View	405
14.3	The Software Quality Dilemma	406
14.3.1	"Good Enough" Software	406
14.3.2	The Cost of Quality	407
14.3.3	Risks	409
14.3.4	Negligence and Liability	410
14.3.5	Quality and Security	410
14.3.6	The Impact of Management Actions	411
14.4	Achieving Software Quality	412
14.4.1	Software Engineering Methods	412
14.4.2	Project Management Techniques	412
14.4.3	Quality Control	412
14.4.4	Quality Assurance	413
14.5	Summary	413
PROBLEMS AND POINTS TO PONDER 414		
FURTHER READINGS AND INFORMATION SOURCES 414		

**CHAPTER 15 REVIEW TECHNIQUES 416**

---

15.1	Cost Impact of Software Defects	417
15.2	Defect Amplification and Removal	418
15.3	Review Metrics and Their Use	420
15.3.1	Analyzing Metrics	420
15.3.2	Cost Effectiveness of Reviews	421
15.4	Reviews: A Formality Spectrum	423
15.5	Informal Reviews	424
15.6	Formal Technical Reviews	426
15.6.1	The Review Meeting	426
15.6.2	Review Reporting and Record Keeping	427
15.6.3	Review Guidelines	427
15.6.4	Sample-Driven Reviews	429
15.7	Summary	430
PROBLEMS AND POINTS TO PONDER 431		
FURTHER READINGS AND INFORMATION SOURCES 431		

**CHAPTER 16 SOFTWARE QUALITY ASSURANCE 432**

---

16.1	Background Issues	433
16.2	Elements of Software Quality Assurance	434
16.3	SQA Tasks, Goals, and Metrics	436
16.3.1	SQA Tasks	436
16.3.2	Goals, Attributes, and Metrics	437
16.4	Formal Approaches to SQA	438
16.5	Statistical Software Quality Assurance	439
16.5.1	A Generic Example	439
16.5.2	Six Sigma for Software Engineering	441
16.6	Software Reliability	442
16.6.1	Measures of Reliability and Availability	442
16.6.2	Software Safety	443
16.7	The ISO 9000 Quality Standards	444
16.8	The SQA Plan	445
16.9	Summary	446
PROBLEMS AND POINTS TO PONDER 447		
FURTHER READINGS AND INFORMATION SOURCES 447		

**CHAPTER 17 SOFTWARE TESTING STRATEGIES 449**

---

17.1	A Strategic Approach to Software Testing	450
17.1.1	Verification and Validation	450
17.1.2	Organizing for Software Testing	451
17.1.3	Software Testing Strategy—The Big Picture	452
17.1.4	Criteria for Completion of Testing	455
17.2	Strategic Issues	455
17.3	Test Strategies for Conventional Software	456
17.3.1	Unit Testing	456
17.3.2	Integration Testing	459
17.4	Test Strategies for Object-Oriented Software	465
17.4.1	Unit Testing in the OO Context	466
17.4.2	Integration Testing in the OO Context	466
17.5	Test Strategies for WebApps	467
17.6	Validation Testing	467

17.6.1	Validation-Test Criteria	468
17.6.2	Configuration Review	468
17.6.3	Alpha and Beta Testing	468
17.7	System Testing	470
17.7.1	Recovery Testing	470
17.7.2	Security Testing	470
17.7.3	Stress Testing	471
17.7.4	Performance Testing	471
17.7.5	Deployment Testing	472
17.8	The Art of Debugging	473
17.8.1	The Debugging Process	473
17.8.2	Psychological Considerations	474
17.8.3	Debugging Strategies	475
17.8.4	Correcting the Error	477
17.9	Summary	478
PROBLEMS AND POINTS TO PONDER 478		
FURTHER READINGS AND INFORMATION SOURCES 479		

**CHAPTER 18 TESTING CONVENTIONAL APPLICATIONS 481**

---

18.1	Software Testing Fundamentals	482
18.2	Internal and External Views of Testing	484
18.3	White-Box Testing	485
18.4	Basis Path Testing	485
18.4.1	Flow Graph Notation	485
18.4.2	Independent Program Paths	487
18.4.3	Deriving Test Cases	489
18.4.4	Graph Matrices	491
18.5	Control Structure Testing	492
18.5.1	Condition Testing	492
18.5.2	Data Flow Testing	493
18.5.3	Loop Testing	493
18.6	Black-Box Testing	495
18.6.1	Graph-Based Testing Methods	495
18.6.2	Equivalence Partitioning	497
18.6.3	Boundary Value Analysis	498
18.6.4	Orthogonal Array Testing	499
18.7	Model-Based Testing	502
18.8	Testing for Specialized Environments, Architectures, and Applications	503
18.8.1	Testing GUIs	503
18.8.2	Testing of Client-Server Architectures	503
18.8.3	Testing Documentation and Help Facilities	505
18.8.4	Testing for Real-Time Systems	506
18.9	Patterns for Software Testing	507
18.10	Summary	508
PROBLEMS AND POINTS TO PONDER 509		
FURTHER READINGS AND INFORMATION SOURCES 510		

**CHAPTER 19 TESTING OBJECT-ORIENTED APPLICATIONS 511**

---

19.1	Broadening the View of Testing	512
19.2	Testing OOA and OOD Models	513

19.2.1	Correctness of OOA and OOD Models	513
19.2.2	Consistency of Object-Oriented Models	514
19.3	Object-Oriented Testing Strategies	516
19.3.1	Unit Testing in the OO Context	516
19.3.2	Integration Testing in the OO Context	516
19.3.3	Validation Testing in an OO Context	517
19.4	Object-Oriented Testing Methods	517
19.4.1	The Test-Case Design Implications of OO Concepts	518
19.4.2	Applicability of Conventional Test-Case Design Methods	518
19.4.3	Fault-Based Testing	519
19.4.4	Test Cases and the Class Hierarchy	519
19.4.5	Scenario-Based Test Design	520
19.4.6	Testing Surface Structure and Deep Structure	522
19.5	Testing Methods Applicable at the Class Level	522
19.5.1	Random Testing for OO Classes	522
19.5.2	Partition Testing at the Class Level	524
19.6	Interclass Test-Case Design	524
19.6.1	Multiple Class Testing	524
19.6.2	Tests Derived from Behavior Models	526
19.7	Summary	527
PROBLEMS AND POINTS TO PONDER 528		
FURTHER READINGS AND INFORMATION SOURCES 528		

---

## CHAPTER 20 TESTING WEB APPLICATIONS 529

20.1	Testing Concepts for WebApps	530
20.1.1	Dimensions of Quality	530
20.1.2	Errors within a WebApp Environment	531
20.1.3	Testing Strategy	532
20.1.4	Test Planning	532
20.2	The Testing Process—An Overview	533
20.3	Content Testing	534
20.3.1	Content Testing Objectives	534
20.3.2	Database Testing	535
20.4	User Interface Testing	537
20.4.1	Interface Testing Strategy	537
20.4.2	Testing Interface Mechanisms	538
20.4.3	Testing Interface Semantics	540
20.4.4	Usability Tests	540
20.4.5	Compatibility Tests	542
20.5	Component-Level Testing	543
20.6	Navigation Testing	545
20.6.1	Testing Navigation Syntax	545
20.6.2	Testing Navigation Semantics	546
20.7	Configuration Testing	547
20.7.1	Server-Side Issues	547
20.7.2	Client-Side Issues	548
20.8	Security Testing	548
20.9	Performance Testing	550
20.9.1	Performance Testing Objectives	550
20.9.2	Load Testing	551
20.9.3	Stress Testing	552

20.10	Summary	553
PROBLEMS AND POINTS TO PONDER		554
FURTHER READINGS AND INFORMATION SOURCES		555

---

**CHAPTER 21 FORMAL MODELING AND VERIFICATION 557**

---

21.1	The Cleanroom Strategy	558
21.2	Functional Specification	560
21.2.1	Black-Box Specification	561
21.2.2	State-Box Specification	562
21.2.3	Clear-Box Specification	562
21.3	Cleanroom Design	563
21.3.1	Design Refinement	563
21.3.2	Design Verification	564
21.4	Cleanroom Testing	566
21.4.1	Statistical Use Testing	566
21.4.2	Certification	567
21.5	Formal Methods Concepts	568
21.6	Applying Mathematical Notation for Formal Specification	571
21.7	Formal Specification Languages	573
21.7.1	Object Constraint Language (OCL)	574
21.7.2	The Z Specification Language	577
21.8	Summary	580
PROBLEMS AND POINTS TO PONDER		581
FURTHER READINGS AND INFORMATION SOURCES		582

---

**CHAPTER 22 SOFTWARE CONFIGURATION MANAGEMENT 584**

---

22.1	Software Configuration Management	585
22.1.1	An SCM Scenario	586
22.1.2	Elements of a Configuration Management System	587
22.1.3	Baselines	587
22.1.4	Software Configuration Items	589
22.2	The SCM Repository	590
22.2.1	The Role of the Repository	590
22.2.2	General Features and Content	591
22.2.3	SCM Features	592
22.3	The SCM Process	593
22.3.1	Identification of Objects in the Software Configuration	594
22.3.2	Version Control	595
22.3.3	Change Control	596
22.3.4	Configuration Audit	599
22.3.5	Status Reporting	600
22.4	Configuration Management for WebApps	601
22.4.1	Dominant Issues	601
22.4.2	WebApp Configuration Objects	603
22.4.3	Content Management	603
22.4.4	Change Management	606
22.4.5	Version Control	608
22.4.6	Auditing and Reporting	609
22.5	Summary	610
PROBLEMS AND POINTS TO PONDER		611
FURTHER READINGS AND INFORMATION SOURCES		612

**CHAPTER 23 PRODUCT METRICS 613**


---

23.1	A Framework for Product Metrics	614
23.1.1	Measures, Metrics, and Indicators	614
23.1.2	The Challenge of Product Metrics	615
23.1.3	Measurement Principles	616
23.1.4	Goal-Oriented Software Measurement	617
23.1.5	The Attributes of Effective Software Metrics	618
23.2	Metrics for the Requirements Model	619
23.2.1	Function-Based Metrics	620
23.2.2	Metrics for Specification Quality	623
23.3	Metrics for the Design Model	624
23.3.1	Architectural Design Metrics	624
23.3.2	Metrics for Object-Oriented Design	627
23.3.3	Class-Oriented Metrics—The CK Metrics Suite	628
23.3.4	Class-Oriented Metrics—The MOOD Metrics Suite	631
23.3.5	OO Metrics Proposed by Lorenz and Kidd	632
23.3.6	Component-Level Design Metrics	632
23.3.7	Operation-Oriented Metrics	634
23.3.8	User Interface Design Metrics	635
23.4	Design Metrics for WebApps	636
23.5	Metrics for Source Code	638
23.6	Metrics for Testing	639
23.6.1	Halstead Metrics Applied to Testing	639
23.6.2	Metrics for Object-Oriented Testing	640
23.7	Metrics for Maintenance	641
23.8	Summary	642
	PROBLEMS AND POINTS TO PONDER	642
	FURTHER READINGS AND INFORMATION SOURCES	643

**PART FOUR MANAGING SOFTWARE PROJECTS 645****CHAPTER 24 PROJECT MANAGEMENT CONCEPTS 646**


---

24.1	The Management Spectrum	647
24.1.1	The People	647
24.1.2	The Product	648
24.1.3	The Process	648
24.1.4	The Project	648
24.2	People	649
24.2.1	The Stakeholders	649
24.2.2	Team Leaders	650
24.2.3	The Software Team	651
24.2.4	Agile Teams	654
24.2.5	Coordination and Communication Issues	655
24.3	The Product	656
24.3.1	Software Scope	656
24.3.2	Problem Decomposition	656
24.4	The Process	657
24.4.1	Melding the Product and the Process	657
24.4.2	Process Decomposition	658
24.5	The Project	660
24.6	The V <sup>5</sup> HH Principle	661

24.7	Critical Practices	662
24.8	Summary	663
PROBLEMS AND POINTS TO PONDER 663		
FURTHER READINGS AND INFORMATION SOURCES 664		

---

**CHAPTER 25 PROCESS AND PROJECT METRICS 666**

25.1	Metrics in the Process and Project Domains	667
25.1.1	Process Metrics and Software Process Improvement	667
25.1.2	Project Metrics	670
25.2	Software Measurement	671
25.2.1	Size-Oriented Metrics	672
25.2.2	Function-Oriented Metrics	673
25.2.3	Reconciling LOC and FP Metrics	673
25.2.4	Object-Oriented Metrics	675
25.2.5	Use-Case–Oriented Metrics	676
25.2.6	WebApp Project Metrics	677
25.3	Metrics for Software Quality	679
25.3.1	Measuring Quality	680
25.3.2	Defect Removal Efficiency	681
25.4	Integrating Metrics within the Software Process	682
25.4.1	Arguments for Software Metrics	683
25.4.2	Establishing a Baseline	683
25.4.3	Metrics Collection, Computation, and Evaluation	684
25.5	Metrics for Small Organizations	684
25.6	Establishing a Software Metrics Program	686
25.7	Summary	688
PROBLEMS AND POINTS TO PONDER 688		
FURTHER READINGS AND INFORMATION SOURCES 689		

---

**CHAPTER 26 ESTIMATION FOR SOFTWARE PROJECTS 691**

26.1	Observations on Estimation	692
26.2	The Project Planning Process	693
26.3	Software Scope and Feasibility	694
26.4	Resources	695
26.4.1	Human Resources	695
26.4.2	Reusable Software Resources	696
26.4.3	Environmental Resources	696
26.5	Software Project Estimation	697
26.6	Decomposition Techniques	698
26.6.1	Software Sizing	698
26.6.2	Problem-Based Estimation	699
26.6.3	An Example of LOC-Based Estimation	701
26.6.4	An Example of FP-Based Estimation	702
26.6.5	Process-Based Estimation	703
26.6.6	An Example of Process-Based Estimation	704
26.6.7	Estimation with Use Cases	705
26.6.8	An Example of Use-Case–Based Estimation	706
26.6.9	Reconciling Estimates	707
26.7	Empirical Estimation Models	708
26.7.1	The Structure of Estimation Models	709
26.7.2	The COCOMO II Model	709
26.7.3	The Software Equation	711

26.8	Estimation for Object-Oriented Projects	712
26.9	Specialized Estimation Techniques	713
26.9.1	Estimation for Agile Development	713
26.9.2	Estimation for WebApp Projects	714
26.10	The Make/Buy Decision	715
26.10.1	Creating a Decision Tree	715
26.10.2	Outsourcing	717
26.11	Summary	718
PROBLEMS AND POINTS TO PONDER 719		
FURTHER READINGS AND INFORMATION SOURCES 719		

**CHAPTER 27 PROJECT SCHEDULING 721**

---

27.1	Basic Concepts	722
27.2	Project Scheduling	724
27.2.1	Basic Principles	725
27.2.2	The Relationship Between People and Effort	725
27.2.3	Effort Distribution	727
27.3	Defining a Task Set for the Software Project	728
27.3.1	A Task Set Example	729
27.3.2	Refinement of Software Engineering Actions	730
27.4	Defining a Task Network	731
27.5	Scheduling	732
27.5.1	Time-Line Charts	732
27.5.2	Tracking the Schedule	734
27.5.3	Tracking Progress for an OO Project	735
27.5.4	Scheduling for WebApp Projects	736
27.6	Earned Value Analysis	739
27.7	Summary	741
PROBLEMS AND POINTS TO PONDER 741		
FURTHER READINGS AND INFORMATION SOURCES 743		

**CHAPTER 28 RISK MANAGEMENT 744**

---

28.1	Reactive versus Proactive Risk Strategies	745
28.2	Software Risks	745
28.3	Risk Identification	747
28.3.1	Assessing Overall Project Risk	748
28.3.2	Risk Components and Drivers	749
28.4	Risk Projection	749
28.4.1	Developing a Risk Table	750
28.4.2	Assessing Risk Impact	752
28.5	Risk Refinement	754
28.6	Risk Mitigation, Monitoring, and Management	755
28.7	The RMMM Plan	757
28.8	Summary	759
PROBLEMS AND POINTS TO PONDER 759		
FURTHER READINGS AND INFORMATION SOURCES 760		

**CHAPTER 29 MAINTENANCE AND REENGINEERING 761**

---

29.1	Software Maintenance	762
29.2	Software Supportability	764

29.3	Reengineering	764
29.4	Business Process Reengineering	765
29.4.1	Business Processes	765
29.4.2	A BPR Model	766
29.5	Software Reengineering	768
29.5.1	A Software Reengineering Process Model	768
29.5.2	Software Reengineering Activities	770
29.6	Reverse Engineering	772
29.6.1	Reverse Engineering to Understand Data	773
29.6.2	Reverse Engineering to Understand Processing	774
29.6.3	Reverse Engineering User Interfaces	775
29.7	Restructuring	776
29.7.1	Code Restructuring	776
29.7.2	Data Restructuring	777
29.8	Forward Engineering	778
29.8.1	Forward Engineering for Client-Server Architectures	779
29.8.2	Forward Engineering for Object-Oriented Architectures	780
29.9	The Economics of Reengineering	780
29.10	Summary	781
PROBLEMS AND POINTS TO PONDER 782		
FURTHER READINGS AND INFORMATION SOURCES 783		

**PART FIVE ADVANCED TOPICS 785****CHAPTER 30 SOFTWARE PROCESS IMPROVEMENT 786**

30.1	What Is SPI?	787
30.1.1	Approaches to SPI	787
30.1.2	Maturity Models	789
30.1.3	Is SPI for Everyone?	790
30.2	The SPI Process	791
30.2.1	Assessment and Gap Analysis	791
30.2.2	Education and Training	793
30.2.3	Selection and Justification	793
30.2.4	Installation/Migration	794
30.2.5	Evaluation	795
30.2.6	Risk Management for SPI	795
30.2.7	Critical Success Factors	796
30.3	The CMMI	797
30.4	The People CMM	801
30.5	Other SPI Frameworks	802
30.6	SPI Return on Investment	804
30.7	SPI Trends	805
30.8	Summary	806
PROBLEMS AND POINTS TO PONDER 806		
FURTHER READINGS AND INFORMATION SOURCES 807		

**CHAPTER 31 EMERGING TRENDS IN SOFTWARE ENGINEERING 808**

31.1	Technology Evolution	809
31.2	Observing Software Engineering Trends	811

31.3	Identifying "Soft Trends"	812
31.3.1	Managing Complexity	814
31.3.2	Open-World Software	815
31.3.3	Emergent Requirements	816
31.3.4	The Talent Mix	816
31.3.5	Software Building Blocks	817
31.3.6	Changing Perceptions of "Value"	818
31.3.7	Open Source	818
31.4	Technology Directions	819
31.4.1	Process Trends	819
31.4.2	The Grand Challenge	821
31.4.3	Collaborative Development	822
31.4.4	Requirements Engineering	824
31.4.5	Model-Driven Software Development	825
31.4.6	Postmodern Design	825
31.4.7	Test-Driven Development	826
31.5	Tools-Related Trends	827
31.5.1	Tools That Respond to Soft Trends	828
31.5.2	Tools That Address Technology Trends	830
31.6	Summary	830
PROBLEMS AND POINTS TO PONDER 831		
FURTHER READINGS AND INFORMATION SOURCES 831		

---

**CHAPTER 32 CONCLUDING COMMENTS 833**

32.1	The Importance of Software—Revisited	834
32.2	People and the Way They Build Systems	834
32.3	New Modes for Representing Information	835
32.4	The Long View	837
32.5	The Software Engineer's Responsibility	838
32.6	A Final Comment	839

**APPENDIX 1 AN INTRODUCTION TO UML 841****APPENDIX 2 OBJECT-ORIENTED CONCEPTS 863****REFERENCES 871****INDEX 889**

**W**hen computer software succeeds—when it meets the needs of the people who use it, when it performs flawlessly over a long period of time, when it is easy to modify and even easier to use—it can and does change things for the better. But when software fails—when its users are dissatisfied, when it is error prone, when it is difficult to change and even harder to use—bad things can and do happen. We all want to build software that makes things better, avoiding the bad things that lurk in the shadow of failed efforts. To succeed, we need discipline when software is designed and built. We need an engineering approach.

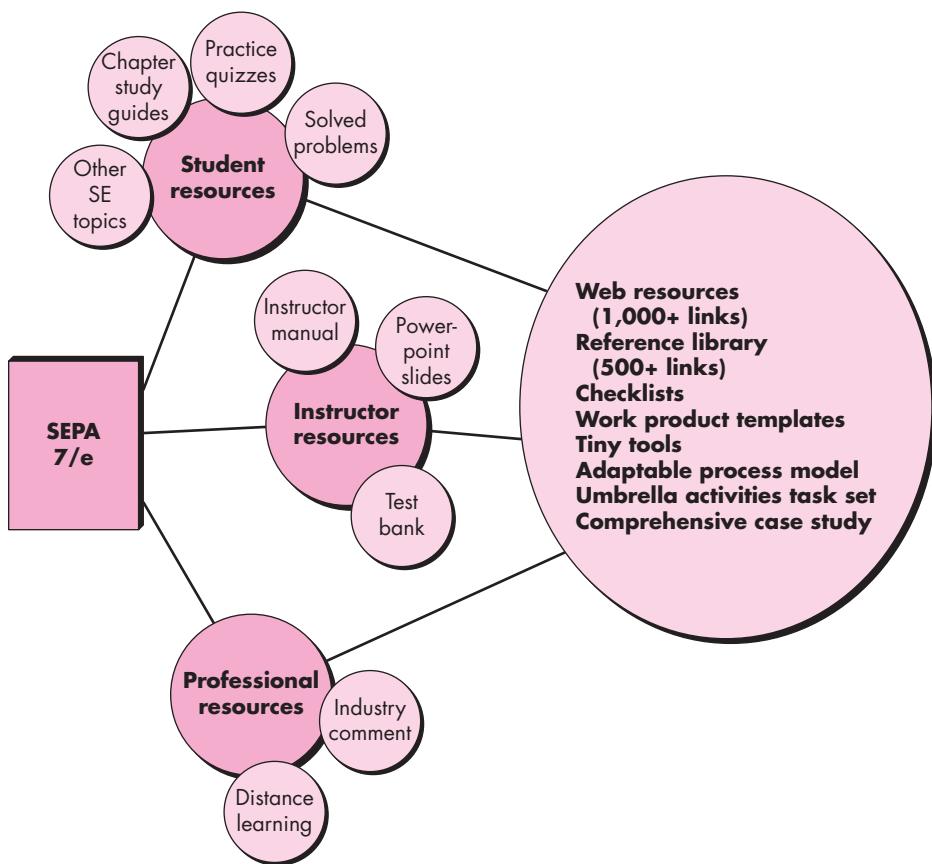
It has been almost three decades since the first edition of this book was written. During that time, software engineering has evolved from an obscure idea practiced by a relatively small number of zealots to a legitimate engineering discipline. Today, it is recognized as a subject worthy of serious research, conscientious study, and tumultuous debate. Throughout the industry, software engineer has replaced programmer as the job title of preference. Software process models, software engineering methods, and software tools have been adopted successfully across a broad spectrum of industry segments.

Although managers and practitioners alike recognize the need for a more disciplined approach to software, they continue to debate the manner in which discipline is to be applied. Many individuals and companies still develop software haphazardly, even as they build systems to service today's most advanced technologies. Many professionals and students are unaware of modern methods. And as a result, the quality of the software that we produce suffers, and bad things happen. In addition, debate and controversy about the true nature of the software engineering approach continue. The status of software engineering is a study in contrasts. Attitudes have changed, progress has been made, but much remains to be done before the discipline reaches full maturity.

The seventh edition of *Software Engineering: A Practitioner's Approach* is intended to serve as a guide to a maturing engineering discipline. Like the six editions that preceded it, the seventh edition is intended for both students and practitioners, retaining its appeal as a guide to the industry professional and a comprehensive introduction to the student at the upper-level undergraduate or first-year graduate level.

The seventh edition is considerably more than a simple update. The book has been revised and restructured to improve pedagogical flow and emphasize new and important software engineering processes and practices. In addition, a revised and updated “support system,” illustrated in the figure, provides a comprehensive set of student, instructor, and professional resources to complement the content of the book. These resources are presented as part of a website ([www.mhhe.com/pressman](http://www.mhhe.com/pressman)) specifically designed for *Software Engineering: A Practitioner's Approach*.

**The Seventh Edition.** The 32 chapters of the seventh edition have been reorganized into five parts. This organization, which differs considerably from the sixth edition, has been done to better compartmentalize topics and assist instructors who may not have the time to complete the entire book in one term.

**Support System for SEPA, 7/e**

Part 1, *The Process*, presents a variety of different views of software process, considering all important process models and addressing the debate between prescriptive and agile process philosophies. Part 2, *Modeling*, presents analysis and design methods with an emphasis on object-oriented techniques and UML modeling. Pattern-based design and design for Web applications are also considered. Part 3, *Quality Management*, presents the concepts, procedures, techniques, and methods that enable a software team to assess software quality, review software engineering work products, conduct SQA procedures, and apply an effective testing strategy and tactics. In addition, formal modeling and verification methods are also considered. Part 4, *Managing Software Projects*, presents topics that are relevant to those who plan, manage, and control a software development project. Part 5, *Advanced Topics*, considers software process improvement and software engineering trends. Continuing in the tradition of past editions, a series of sidebars is used throughout the book to present the trials and tribulations of a (fictional) software team and to provide supplementary materials about methods and tools that are relevant to chapter topics. Two new appendices provide brief tutorials on UML and object-oriented thinking for those who may be unfamiliar with these important topics.

The five-part organization of the seventh edition enables an instructor to “cluster” topics based on available time and student need. An entire one-term course can be built around one or more of the five parts. A software engineering survey course would select chapters from all five parts. A software engineering course that emphasizes analysis and design would select topics from Parts 1 and 2. A testing-oriented software engineering course would select topics from Parts 1 and 3, with a brief foray into Part 2. A “management course” would stress Parts 1 and 4. By organizing the seventh edition in this way, I have attempted to provide an instructor with a number of teaching options. In every case, the content of the seventh edition is complemented by the following elements of the *SEPA, 7/e Support System*.

**Student Resources.** A wide variety of student resources includes an extensive online learning center encompassing chapter-by-chapter study guides, practice quizzes, problem solutions, and a variety of Web-based resources including software engineering checklists, an evolving collection of “tiny tools,” a comprehensive case study, work product templates, and many other resources. In addition, over 1000 categorized *Web References* allow a student to explore software engineering in greater detail and a *Reference Library* with links to over 500 downloadable papers provides an in-depth source of advanced software engineering information.

**Instructor Resources.** A broad array of instructor resources has been developed to supplement the seventh edition. These include a complete online *Instructor’s Guide* (also downloadable) and supplementary teaching materials including a complete set of over 700 *PowerPoint Slides* that may be used for lectures, and a test bank. Of course, all resources available for students (e.g., tiny tools, the Web References, the downloadable Reference Library) and professionals are also available.

The *Instructor’s Guide for Software Engineering: A Practitioner’s Approach* presents suggestions for conducting various types of software engineering courses, recommendations for a variety of software projects to be conducted in conjunction with a course, solutions to selected problems, and a number of useful teaching aids.

**Professional Resources.** A collection of resources available to industry practitioners (as well as students and faculty) includes outlines and samples of software engineering documents and other work products, a useful set of software engineering checklists, a catalog of software engineering (CASE) tools, a comprehensive collection of Web-based resources, and an “adaptable process model” that provides a detailed task breakdown of the software engineering process.

When coupled with its online support system, the seventh edition of *Software Engineering: A Practitioner’s Approach*, provides flexibility and depth of content that cannot be achieved by a textbook alone.

**Acknowledgments.** My work on the seven editions of *Software Engineering: A Practitioner’s Approach* has been the longest continuing technical project of my life. Even when the writing stops, information extracted from the technical literature continues to be assimilated and organized, and criticism and suggestions from readers worldwide is evaluated and cataloged. For this reason, my thanks to the many authors of books, papers, and articles (in both hardcopy and electronic media) who have provided me with additional insight, ideas, and commentary over nearly 30 years.

Special thanks go to Tim Lethbridge of the University of Ottawa, who assisted me in the development of UML and OCL examples and developed the case study that accompanies this book, and Dale Skrien of Colby College, who developed the UML tutorial in

Appendix 1. Their assistance and comments were invaluable. Special thanks also go to Bruce Maxim of the University of Michigan–Dearborn, who assisted me in developing much of the pedagogical website content that accompanies this book. Finally, I wish to thank the reviewers of the seventh edition: Their in-depth comments and thoughtful criticism have been invaluable.

Osman Balci,  
*Virginia Tech University*

Max Fomitchev,  
*Penn State University*

Jerry (Zeyu) Gao,  
*San Jose State University*

Guillermo Garcia,  
*Universidad Alfonso X Madrid*

Pablo Gervas,  
*Universidad Complutense de Madrid*

SK Jain,  
*National Institute of Technology Hamirpur*

Saeed Monemi,  
*Cal Poly Pomona*

Ahmed Salem,  
*California State University*

Vasudeva Varma,  
*IIT Hyderabad*

The content of the seventh edition of *Software Engineering: A Practitioner's Approach* has been shaped by industry professionals, university professors, and students who have used earlier editions of the book and have taken the time to communicate their suggestions, criticisms, and ideas. My thanks to each of you. In addition, my personal thanks go to our many industry clients worldwide, who certainly have taught me as much or more than I could ever teach them.

As the editions of this book have evolved, my sons, Mathew and Michael, have grown from boys to men. Their maturity, character, and success in the real world have been an inspiration to me. Nothing has filled me with more pride. And finally, to Barbara, my love and thanks for tolerating the many, many hours in the office and encouraging still another edition of “the book.”

*Roger S. Pressman*

# SOFTWARE AND SOFTWARE ENGINEERING

1

## KEY CONCEPTS

application domains .....	7
characteristics of software .....	4
framework activities .....	15
legacy software ..	9
practice .....	17
principles .....	19
software engineering ..	12
software myths ..	21
software process ..	14
umbrella activities .....	16
WebApps .....	10

## QUICK LOOK

**What is it?** Computer software is the product that software professionals build and then support over the long term. It encompasses programs that execute within a computer of any size and architecture, content that is presented as the computer programs execute, and descriptive information in both hard copy and virtual forms that encompass virtually any electronic media. Software engineering encompasses a process, a collection of methods (practice) and an array of tools that allow professionals to build high-quality computer software.

**Who does it?** Software engineers build and support software, and virtually everyone in the industrialized world uses it either directly or indirectly.

**Why is it important?** Software is important because it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture, and our everyday activities.

**H**e had the classic look of a senior executive for a major software company—mid-40s, slightly graying at the temples, trim and athletic, with eyes that penetrated the listener as he spoke. But what he said shocked me. “Software is *dead*.”

I blinked with surprise and then smiled. “You’re joking, right? The world is driven by software and your company has profited handsomely because of it. It isn’t dead! It’s alive and growing.”

He shook his head emphatically. “No, it’s dead . . . at least as we once knew it.” I leaned forward. “Go on.”

He spoke while tapping the table for emphasis. “The old-school view of software—you buy it, you own it, and it’s your job to manage it—that’s coming to an end. Today, with Web 2.0 and pervasive computing coming on strong, we’re going to be seeing a completely different generation of software. It’ll be delivered via the Internet and will look exactly like it’s residing on each user’s computing device . . . but it’ll reside on a far-away server.”

Software engineering is important because it enables us to build complex systems in a timely manner and with high quality.

**What are the steps?** You build computer software like you build any successful product, by applying an agile, adaptable process that leads to a high-quality result that meets the needs of the people who will use the product. You apply a software engineering approach.

**What is the work product?** From the point of view of a software engineer, the work product is the set of programs, content (data), and other work products that are computer software. But from the user’s viewpoint, the work product is the resultant information that somehow makes the user’s world better.

**How do I ensure that I’ve done it right?** Read the remainder of this book, select those ideas that are applicable to the software that you build, and apply them to your work.

I had to agree. "So, your life will be much simpler. You guys won't have to worry about five different versions of the same App in use across tens of thousands of users."

He smiled. "Absolutely. Only the most current version residing on our servers. When we make a change or a correction, we supply updated functionality and content to every user. Everyone has it instantly!"

I grimaced. "But if you make a mistake, everyone has that instantly as well."

He chuckled. "True, that's why we're redoubling our efforts to do even better software engineering. Problem is, we have to do it 'fast' because the market has accelerated in every application area."

I leaned back and put my hands behind my head. "You know what they say, . . . you can have it fast, you can have it right, or you can have it cheap. Pick two!"

"I'll take it fast and right," he said as he began to get up.

I stood as well. "Then you really do need software engineering."

"I know that," he said as he began to move away. "The problem is, we've got to convince still another generation of techies that it's true!"

Is software *really* dead? If it was, you wouldn't be reading this book!

Computer software continues to be the single most important technology on the world stage. And it's also a prime example of the law of unintended consequences. Fifty years ago no one could have predicted that software would become an indispensable technology for business, science, and engineering; that software would enable the creation of new technologies (e.g., genetic engineering and nanotechnology), the extension of existing technologies (e.g., telecommunications), and the radical change in older technologies (e.g., the printing industry); that software would be the driving force behind the personal computer revolution; that shrink-wrapped software products would be purchased by consumers in neighborhood malls; that software would slowly evolve from a product to a service as "on-demand" software companies deliver just-in-time functionality via a Web browser; that a software company would become larger and more influential than almost all industrial-era companies; that a vast software-driven network called the Internet would evolve and change everything from library research to consumer shopping to political discourse to the dating habits of young (and not so young) adults.

No one could foresee that software would become embedded in systems of all kinds: transportation, medical, telecommunications, military, industrial, entertainment, office machines, . . . the list is almost endless. And if you believe the law of unintended consequences, there are many effects that we cannot yet predict.

No one could predict that millions of computer programs would have to be corrected, adapted, and enhanced as time passed. The burden of performing these "maintenance" activities would absorb more people and more resources than all work applied to the creation of new software.

As software's importance has grown, the software community has continually attempted to develop technologies that will make it easier, faster, and less expensive

**quote:**

"Ideas and technological discoveries are the driving engines of economic growth."

Wall Street Journal

to build and maintain high-quality computer programs. Some of these technologies are targeted at a specific application domain (e.g., website design and implementation); others focus on a technology domain (e.g., object-oriented systems or aspect-oriented programming); and still others are broad-based (e.g., operating systems such as Linux). However, we have yet to develop a software technology that does it all, and the likelihood of one arising in the future is small. And yet, people bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right.

This book presents a framework that can be used by those who build computer software—people who must get it right. The framework encompasses a process, a set of methods, and an array of tools that we call *software engineering*.

## 1.1 THE NATURE OF SOFTWARE



Software is both a product and a vehicle that delivers a product.

Today, software takes on a dual role. It is a product, and at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or more broadly, by a network of computers that are accessible by local hardware. Whether it resides within a mobile phone or operates inside a mainframe computer, software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources. As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

Software delivers the most important product of our time—*information*. It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information networks (e.g., the Internet), and provides the means for acquiring information in all of its forms.

The role of computer software has undergone significant change over the last half-century. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options, have all precipitated more sophisticated and complex computer-based systems. Sophistication and complexity can produce dazzling results when a system succeeds, but they can also pose huge problems for those who must build complex systems.

Today, a huge software industry has become a dominant factor in the economies of the industrialized world. Teams of software specialists, each focusing on one part of the technology required to deliver a complex application, have replaced the lone programmer of an earlier era. And yet, the questions that were asked of the lone

### note:

"Software is a place where dreams are planted and nightmares harvested, an abstract, mystical swamp where terrible demons compete with magical panaceas, a world of werewolves and silver bullets."

Brad J. Cox

programmer are the same questions that are asked when modern computer-based systems are built:<sup>1</sup>

- Why does it take so long to get software finished?
- Why are development costs so high?
- Why can't we find all errors before we give the software to our customers?
- Why do we spend so much time and effort maintaining existing programs?
- Why do we continue to have difficulty in measuring progress as software is being developed and maintained?

These, and many other questions, are a manifestation of the concern about software and the manner in which it is developed—a concern that has lead to the adoption of software engineering practice.

### 1.1.1 Defining Software

Today, most professionals and many members of the public at large feel that they understand software. But do they?

A textbook description of software might take the following form:



Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

There is no question that other more complete definitions could be offered.

But a more formal definition probably won't measurably improve your understanding. To accomplish that, it's important to examine the characteristics of software that make it different from other things that human beings build. Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

1. *Software is developed or engineered; it is not manufactured in the classical sense.*

Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent



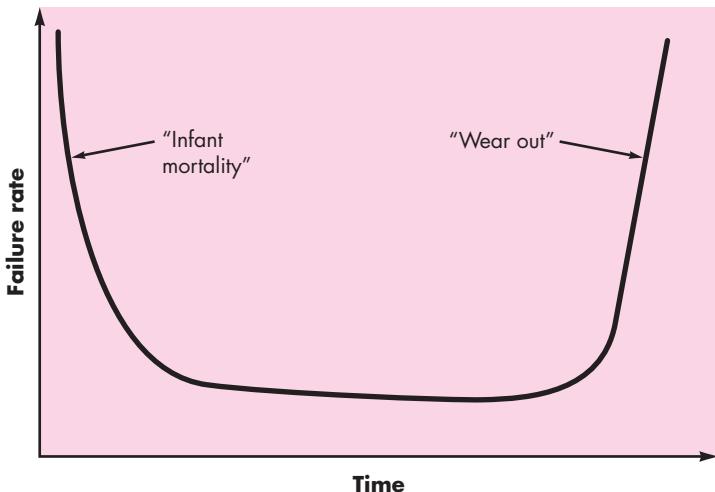
Software is  
engineered, not  
manufactured.

---

1 In an excellent book of essays on the software business, Tom DeMarco [DeM95] argues the counterpoint. He states: "Instead of asking why software costs so much, we need to begin asking 'What have we done to make it possible for today's software to cost so little?' The answer to that question will help us continue the extraordinary level of achievement that has always distinguished the software industry."

**FIGURE 1.1**

Failure curve  
for hardware



(or easily corrected) for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different (see Chapter 24). Both activities require the construction of a “product,” but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

### KEY POINT

Software doesn't wear out, but it does deteriorate.

**2. Software doesn't “wear out.”**

Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the “bathtub curve,” indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to *wear out*.

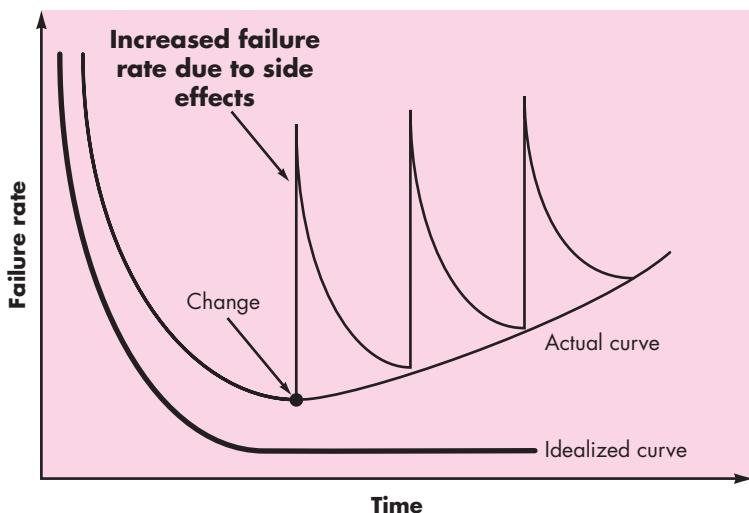
Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the “idealized curve” shown in Figure 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does deteriorate!

### ADVICE

If you want to reduce software deterioration, you'll have to do better software design (Chapters 8 to 13).

**FIGURE 1.2**

Failure curves  
for software



### KEY POINT

Software engineering methods strive to reduce the magnitude of the spikes and the slope of the actual curve in Figure 1.2.

This seeming contradiction can best be explained by considering the actual curve in Figure 1.2. During its life,<sup>2</sup> software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the “actual curve” (Figure 1.2). Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

- 3.** *Although the industry is moving toward component-based construction, most software continues to be custom built.*

As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent

### quote:

“Ideas are the building blocks of ideas.”

**Jason Zebekazy**

<sup>2</sup> In fact, from the moment that development begins and long before the first version is delivered, changes may be requested by a variety of different stakeholders.

something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts.<sup>3</sup> For example, today's interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

### 1.1.2 Software Application Domains

Today, seven broad categories of computer software present continuing challenges for software engineers:

**System software**—a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate,<sup>4</sup> information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data. In either case, the systems software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

**Application software**—stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. In addition to conventional data processing applications, application software is used to control business functions in real time (e.g., point-of-sale transaction processing, real-time manufacturing process control).

**Engineering/scientific software**—has been characterized by “number crunching” algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from

#### WebRef

One of the most comprehensive libraries of shareware/freeware can be found at [shareware.cnet.com](http://shareware.cnet.com)

<sup>3</sup> Component-based development is discussed in Chapter 10.

<sup>4</sup> Software is *determinate* if the order and timing of inputs, processing, and outputs is predictable. Software is *indeterminate* if the order and timing of inputs, processing, and outputs cannot be predicted in advance.

conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

**Embedded software**—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

**Product-line software**—designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).

**Web applications**—called “WebApps,” this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, as Web 2.0 emerges, WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

**Artificial intelligence software**—makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.



### Quote:

“There is no computer that has common sense.”

Marvin Minsky

Millions of software engineers worldwide are hard at work on software projects in one or more of these categories. In some cases, new systems are being built, but in many others, existing applications are being corrected, adapted, and enhanced. It is not uncommon for a young software engineer to work a program that is older than she is! Past generations of software people have left a legacy in each of the categories I have discussed. Hopefully, the legacy to be left behind by this generation will ease the burden of future software engineers. And yet, new challenges (Chapter 31) have appeared on the horizon:

**Open-world computing**—the rapid growth of wireless networking may soon lead to true pervasive, distributed computing. The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks.

**Netsourcing**—the World Wide Web is rapidly becoming a computing engine as well as a content provider. The challenge for software engineers is to architect simple (e.g., personal financial planning) and sophisticated applications that provide a benefit to targeted end-user markets worldwide.

**Open source**—a growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that many people can contribute to its development. The challenge for software engineers is to build source code that is self-descriptive, but more importantly, to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.



### Quote:

"You can't always predict, but you can always prepare."

Anonymous

Each of these new challenges will undoubtedly obey the law of unintended consequences and have effects (for businesspeople, software engineers, and end users) that cannot be predicted today. However, software engineers can prepare by instantiating a process that is agile and adaptable enough to accommodate dramatic changes in technology and to business rules that are sure to come over the next decade.

### 1.1.3 Legacy Software

Hundreds of thousands of computer programs fall into one of the seven broad application domains discussed in the preceding subsection. Some of these are state-of-the-art software—just released to individuals, industry, and government. But other programs are older, in some cases *much* older.

These older programs—often referred to as *legacy software*—have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues [Day99] describe legacy software in the following way:

Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

Liu and his colleagues [Liu98] extend this description by noting that “many legacy systems remain supportive to core business functions and are ‘indispensable’ to the business.” Hence, legacy software is characterized by longevity and business criticality.

Unfortunately, there is sometimes one additional characteristic that is present in legacy software—*poor quality*.<sup>5</sup> Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results



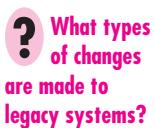
What do I do if I encounter a legacy system that exhibits poor quality?

---

<sup>5</sup> In this case, quality is judged based on modern software engineering thinking—a somewhat unfair criterion since some modern software engineering concepts and principles may not have been well understood at the time that the legacy software was developed.

that were never archived, a poorly managed change history—the list can be quite long. And yet, these systems support “core business functions and are indispensable to the business.” What to do?

The only reasonable answer may be: *Do nothing*, at least until the legacy system must undergo some significant change. If the legacy software meets the needs of its users and runs reliably, it isn’t broken and does not need to be fixed. However, as time passes, legacy systems often evolve for one or more of the following reasons:



- The software must be adapted to meet the needs of new computing environments or technology.
- The software must be enhanced to implement new business requirements.
- The software must be extended to make it interoperable with other more modern systems or databases.
- The software must be re-architected to make it viable within a network environment.



Every software engineer must recognize that change is natural. Don't try to fight it.

When these modes of evolution occur, a legacy system must be reengineered (Chapter 29) so that it remains viable into the future. The goal of modern software engineering is to “devise methodologies that are founded on the notion of evolution”; that is, the notion that software systems continually change, new software systems are built from the old ones, and . . . all must interoperate and cooperate with each other” [Day99].

## 1.2 THE UNIQUE NATURE OF WEBAPPS



“By the time we see any sort of stabilization, the Web will have turned into something completely different.”

Louis Monier

In the early days of the World Wide Web (circa 1990 to 1995), websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics. As time passed, the augmentation of HTML by development tools (e.g., XML, Java) enabled Web engineers to provide computing capability along with informational content. *Web-based systems and applications*<sup>6</sup> (I refer to these collectively as *WebApps*) were born. Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.

As noted in Section 1.1.2, WebApps are one of a number of distinct software categories. And yet, it can be argued that WebApps are different. Powell [Pow98] suggests that Web-based systems and applications “involve a mixture between print publishing and software development, between marketing and computing, between

---

6 In the context of this book, the term *Web application* (WebApp) encompasses everything from a simple Web page that might help a consumer compute an automobile lease payment to a comprehensive website that provides complete travel services for businesspeople and vacationers. Included within this category are complete websites, specialized functionality within websites, and information processing applications that reside on the Internet or on an Intranet or Extranet.

internal communications and external relations, and between art and technology.” The following attributes are encountered in the vast majority of WebApps.



**Network intensiveness.** A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).

**Concurrency.** A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.

**Unpredictable load.** The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.

**Performance.** If a WebApp user must wait too long (for access, for server-side processing, for client-side formatting and display), he or she may decide to go elsewhere.

**Availability.** Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis. Users in Australia or Asia might demand access during times when traditional domestic software applications in North America might be taken off-line for maintenance.

**Data driven.** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).

**Content sensitive.** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.

**Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously. It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.

**Immediacy.** Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.<sup>7</sup>

**Security.** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes

---

<sup>7</sup> With modern tools, sophisticated Web pages can be produced in only a few hours.

of data transmission, strong security measures must be implemented throughout the infrastructure that supports a WebApp and within the application itself.

**Aesthetics.** An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

It can be argued that other application categories discussed in Section 1.1.2 can exhibit some of the attributes noted. However, WebApps almost always exhibit all of them.

### 1.3 SOFTWARE ENGINEERING

In order to build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities:



Understand the problem before you build a solution.

- Software has become deeply embedded in virtually every aspect of our lives, and as a consequence, the number of people who have an interest in the features and functions provided by a specific application<sup>8</sup> has grown dramatically. When a new application or embedded system is to be built, many voices must be heard. And it sometimes seems that each of them has a slightly different idea of what software features and functions should be delivered. *It follows that a concerted effort should be made to understand the problem before a software solution is developed.*



Design is a pivotal software engineering activity.

- The information technology requirements demanded by individuals, businesses, and governments grow increasing complex with each passing year. Large teams of people now create computer programs that were once built by a single individual. Sophisticated software that was once implemented in a predictable, self-contained, computing environment is now embedded inside everything from consumer electronics to medical devices to weapons systems. The complexity of these new computer-based systems and products demands careful attention to the interactions of all system elements. *It follows that design becomes a pivotal activity.*



Both quality and maintainability are an outgrowth of good design.

- Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic failures. *It follows that software should exhibit high quality.*
- As the perceived value of a specific application grows, the likelihood is that its user base and longevity will also grow. As its user base and time-in-use

<sup>8</sup> I will call these people “stakeholders” later in this book.

increase, demands for adaptation and enhancement will also grow. *It follows that software should be maintainable.*

These simple realities lead to one conclusion: *software in all of its forms and across all of its application domains should be engineered*. And that leads us to the topic of this book—*software engineering*.

**quote:**

“More than a discipline or a body of knowledge, engineering is a verb, an action word, a way of approaching a problem.”

Scott Whitmire

**How do we define software engineering?**

**KEY POINT**

Software engineering encompasses a process, methods for managing and engineering software, and tools.

Although hundreds of authors have developed personal definitions of software engineering, a definition proposed by Fritz Bauer [Nau69] at the seminal conference on the subject still serves as a basis for discussion:

[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

You will be tempted to add to this definition.<sup>9</sup> It says little about the technical aspects of software quality; it does not directly address the need for customer satisfaction or timely product delivery; it omits mention of the importance of measurement and metrics; it does not state the importance of an effective process. And yet, Bauer’s definition provides us with a baseline. What are the “sound engineering principles” that can be applied to computer software development? How do we “economically” build software so that it is “reliable”? What is required to create computer programs that work “efficiently” on not one but many different “real machines”? These are the questions that continue to challenge software engineers.

The IEEE [IEE93a] has developed a more comprehensive definition when it states:

Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

And yet, a “systematic, disciplined, and quantifiable” approach applied by one software team may be burdensome to another. We need discipline, but we also need adaptability and agility.

Software engineering is a layered technology. Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies<sup>10</sup> foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus.

The foundation for software engineering is the *process* layer. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework

<sup>9</sup> For numerous additional definitions of *software engineering*, see [www.answers.com/topic/software-engineering#wp\\_note-13](http://www.answers.com/topic/software-engineering#wp_note-13).

<sup>10</sup> Quality management and related approaches are discussed in Chapter 14 and throughout Part 3 of this book.

**FIGURE 1.3**

Software engineering layers

**WebRef**

CrossTalk is a journal that provides pragmatic information on process, methods, and tools. It can be found at:  
[www.stsc.hill.af.mil](http://www.stsc.hill.af.mil).

that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering *methods* provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering *tools* provide automated or semiautomated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering*, is established.

## 1.4 THE SOFTWARE PROCESS

**?** **What are the elements of a software process?**

**quote:**

"A process defines who is doing what when and how to reach a certain goal."

Ivar Jacobson,  
Grady Booch,  
and James Rumbaugh

A *process* is a collection of activities, actions, and tasks that are performed when some work product is to be created. An *activity* strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied. An *action* (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model). A *task* focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

A *process framework* establishes the foundation for a complete software engineering process by identifying a small number of *framework activities* that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of *umbrella activities* that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:



**Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders)<sup>11</sup> The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

**Planning.** Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a *software project plan*—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

**Modeling.** Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

**Construction.** This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

**Deployment.** The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

<sup>11</sup> A *stakeholder* is anyone who has a stake in the successful outcome of the project—business managers, end users, software engineers, support people, etc. Rob Thomsett jokes that, "a stakeholder is a person holding a large and sharp stake. . . . If you don't look after your stakeholders, you know where the stake will end up.").

**quote:**  
"Einstein argued that there must be a simplified explanation of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity."

Fred Brooks

For many software projects, framework activities are applied iteratively as a project progresses. That is, **communication, planning, modeling, construction, and deployment** are applied repeatedly through a number of project iterations. Each project iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

Software engineering process framework activities are complemented by a number of *umbrella activities*. In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

### KEY POINT

Umbrella activities occur throughout the software process and focus primarily on project management, tracking, and control.

**Software project tracking and control**—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

**Risk management**—assesses risks that may affect the outcome of the project or the quality of the product.

**Software quality assurance**—defines and conducts the activities required to ensure software quality.

**Technical reviews**—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

**Measurement**—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

**Software configuration management**—manages the effects of change throughout the software process.

**Reusability management**—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

**Work product preparation and production**—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

### KEY POINT

Software process adaptation is essential for project success.

Each of these umbrella activities is discussed in detail later in this book.

Earlier in this section, I noted that the software engineering process is not a rigid prescription that must be followed dogmatically by a software team. Rather, it should be agile and adaptable (to the problem, to the project, to the team, and to the organizational culture). Therefore, a process adopted for one project might be significantly different than a process adopted for another project. Among the differences are

- Overall flow of activities, actions, and tasks and the interdependencies among them
- Degree to which actions and tasks are defined within each framework activity
- Degree to which work products are identified and required

### How do process models differ from one another?

**note:**

"I feel a recipe is only a theme which an intelligent cook can play each time with a variation."

**Madame Benoit**

- Manner in which quality assurance activities are applied
- Manner in which project tracking and control activities are applied
- Overall degree of detail and rigor with which the process is described
- Degree to which the customer and other stakeholders are involved with the project
- Level of autonomy given to the software team
- Degree to which team organization and roles are prescribed

In Part 1 of this book, I'll examine software process in considerable detail. *Prescriptive process models* (Chapter 2) stress detailed definition, identification, and application of process activities and tasks. Their intent is to improve system quality, make projects more manageable, make delivery dates and costs more predictable, and guide teams of software engineers as they perform the work required to build a system. Unfortunately, there have been times when these objectives were not achieved. If prescriptive models are applied dogmatically and without adaptation, they can increase the level of bureaucracy associated with building computer-based systems and inadvertently create difficulty for all stakeholders.

**What characterizes an "agile" process?**

*Agile process models* (Chapter 3) emphasize project "agility" and follow a set of principles that lead to a more informal (but, proponents argue, no less effective) approach to software process. These process models are generally characterized as "agile" because they emphasize maneuverability and adaptability. They are appropriate for many types of projects and are particularly useful when Web applications are engineered.

## 1.5 SOFTWARE ENGINEERING PRACTICE

**WebRef**

A variety of thought-provoking quotes on the practice of software engineering can be found at [www.literateprogramming.com](http://www.literateprogramming.com)

In Section 1.4, I introduced a generic software process model composed of a set of activities that establish a framework for software engineering practice. Generic framework activities—**communication, planning, modeling, construction, and deployment**—and umbrella activities establish a skeleton architecture for software engineering work. But how does the practice of software engineering fit in? In the sections that follow, you'll gain a basic understanding of the generic concepts and principles that apply to framework activities.<sup>12</sup>

**ADVICE**

You might argue that Polya's approach is simply common sense. True. But it's amazing how often common sense is uncommon in the software world.

### 1.5.1 The Essence of Practice

In a classic book, *How to Solve It*, written before modern computers existed, George Polya [Poly45] outlined the essence of problem solving, and consequently, the essence of software engineering practice:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).

<sup>12</sup> You should revisit relevant sections within this chapter as specific software engineering methods and umbrella activities are discussed later in this book.

3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).

In the context of software engineering, these commonsense steps lead to a series of essential questions [adapted from Pol45]:

**Understand the problem.** It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and then think, *Oh yeah, I understand, let's get on with solving this thing*. Unfortunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions:

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?



### note:

"There is a grain of discovery in the solution of any problem."

George Polya

**Plan the solution.** Now you understand the problem (or so you think) and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

**Carry out the plan.** The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

- *Does the solution conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

**Examine the result.** You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

### 1.5.2 General Principles

The dictionary defines the word *principle* as "an important underlying law or assumption required in a system of thought." Throughout this book I'll discuss principles at many different levels of abstraction. Some focus on software engineering as a whole, others consider a specific generic framework activity (e.g., **communication**), and still others focus on software engineering actions (e.g., architectural design) or technical tasks (e.g., write a usage scenario). Regardless of their level of focus, principles help you establish a mind-set for solid software engineering practice. They are important for that reason.

David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following paragraphs:<sup>13</sup>



*Before beginning a software project, be sure the software has a business purpose and that users perceive value in it.*

#### The First Principle: *The Reason It All Exists*

A software system exists for one reason: *to provide value to its users*. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: "Does this add real value to the system?" If the answer is "no," don't do it. All other principles support this one.

#### The Second Principle: *KISS (Keep It Simple, Stupid!)*

Software design is not a haphazard process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system. This is

---

<sup>13</sup> Reproduced with permission of the author [Hoo96]. Hooker defines patterns for these principles at <http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment>.

 **Quote:**

"There is a certain majesty in simplicity which is far above all the quaintness of wit."

Alexander Pope  
(1688–1744)

not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean “quick and dirty.” In fact, it often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

**The Third Principle: *Maintain the Vision***

*A clear vision is essential to the success of a software project.* Without one, a project almost unfailingly ends up being “of two [or more] minds” about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

**The Fourth Principle: *What You Produce, Others Will Consume***

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, *always specify, design, and implement knowing someone else will have to understand what you are doing*. The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

**The Fifth Principle: *Be Open to the Future***

A system with a long lifetime has more value. In today’s computing environments, where specifications change on a moment’s notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true “industrial-strength” software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner.* Always ask “what if,” and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.<sup>14</sup> This could very possibly lead to the reuse of an entire system.

 **KEY POINT**

If software has value, it will change over its useful life. For that reason, software must be built to be maintainable.

---

<sup>14</sup> This advice can be dangerous if it is taken to extremes. Designing for the “general problem” sometimes requires performance compromises and can make specific solutions inefficient.

### The Sixth Principle: Plan Ahead for Reuse

Reuse saves time and effort.<sup>15</sup> Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process. . . . *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.*

### The Seventh principle: Think!

This last principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results.* When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the first six principles requires intense thought, for which the potential rewards are enormous.

If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer-based systems would be eliminated.

## 1.6 SOFTWARE MYTHS

### Quote:

"In the absence of meaningful standards, a new industry like software comes to depend instead on folklore."

Tom DeMarco

Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score."

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

<sup>15</sup> Although this is true for those who reuse the software on future projects, reuse can be expensive for those who must design and build reusable components. Studies indicate that designing and building reusable components can cost between 25 to 200 percent more than targeted software. In some cases, the cost differential cannot be justified.

**WebRef**

The Software Project Managers Network at [www.spmn.com](http://www.spmn.com) can help you dispel these and other myths.

**Management myths.** Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

**Myth:** *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*

**Reality:** The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

**Myth:** *If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).*

**Reality:** Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

**Myth:** *If I decide to outsource the software project to a third party, I can just relax and let that firm build it.*

**Reality:** If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it out-sources software projects.

**Customer myths.** A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

**Myth:** *A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.*

**Reality:** Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (usually derived



Work very hard to understand what you have to do before you start. You may not be able to develop every detail, but the more you know, the less risk you take.

iteratively) are developed only through effective and continuous communication between customer and developer.

**Myth:** *Software requirements continually change, but change can be easily accommodated because software is flexible.*

**Reality:** It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small.<sup>16</sup> However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.



Whenever you think,  
we don't have time for  
software engineering,  
ask yourself, "Will we  
have time to do it over  
again?"

**Practitioner's myths.** Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

**Myth:** *Once we write the program and get it to work, our job is done.*

**Reality:** Someone once said that “the sooner you begin ‘writing code,’ the longer it’ll take you to get done.” Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

**Myth:** *Until I get the program “running” I have no way of assessing its quality.*

**Reality:** One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the *technical review*. Software reviews (described in Chapter 15) are a “quality filter” that have been found to be more effective than testing for finding certain classes of software defects.

**Myth:** *The only deliverable work product for a successful project is the working program.*

**Reality:** A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

**Myth:** *Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.*

**Reality:** Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

---

<sup>16</sup> Many software engineers have adopted an “agile” approach that accommodates change incrementally, thereby controlling its impact and cost. Agile methods are discussed in Chapter 3.

Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

## 1.7 How It All Starts

Every software project is precipitated by some business need—the need to correct a defect in an existing application; the need to adapt a “legacy system” to a changing business environment; the need to extend the functions and features of an existing application; or the need to create a new product, service, or system.

At the beginning of a software project, the business need is often expressed informally as part of a simple conversation. The conversation presented in the sidebar is typical.

### SAFEHOME<sup>17</sup>



#### *How a Project Starts*

**The scene:** Meeting room at CPI Corporation, a (fictional) company that makes consumer products for home and commercial use.

**The players:** Mal Golden, senior manager, product development; Lisa Perez, marketing manager; Lee Warren, engineering manager; Joe Camalleri, executive VP, business development

#### **The conversation:**

**Joe:** Okay, Lee, what's this I hear about your folks developing a what? A generic universal wireless box?

**Lee:** It's pretty cool . . . about the size of a small matchbook . . . we can attach it to sensors of all kinds, a digital camera, just about anything. Using the 802.11g wireless protocol. It allows us to access the device's output without wires. We think it'll lead to a whole new generation of products.

**Joe:** You agree, Mal?

**Mal:** I do. In fact, with sales as flat as they've been this year, we need something new. Lisa and I have been doing a little market research, and we think we've got a line of products that could be big.

**Joe:** How big . . . bottom line big?

**Mal (avoiding a direct commitment):** Tell him about our idea, Lisa.

**Lisa:** It's a whole new generation of what we call “home management products.” We call 'em *SafeHome*. They use the new wireless interface, provide homeowners or small-business people with a system that's controlled by their PC—home security, home surveillance, appliance and device control—you know, turn down the home air conditioner while you're driving home, that sort of thing.

**Lee (jumping in):** Engineering's done a technical feasibility study of this idea, Joe. It's doable at low manufacturing cost. Most hardware is off-the-shelf. Software is an issue, but it's nothing that we can't do.

**Joe:** Interesting. Now, I asked about the bottom line.

**Mal:** PCs have penetrated over 70 percent of all households in the USA. If we could price this thing right, it could be a killer-App. Nobody else has our wireless box . . . it's proprietary. We'll have a 2-year jump on the competition. Revenue? Maybe as much as 30 to 40 million dollars in the second year.

**Joe (smiling):** Let's take this to the next level. I'm interested.

17 The *SafeHome* project will be used throughout this book to illustrate the inner workings of a project team as it builds a software product. The company, the project, and the people are purely fictitious, but the situations and problems are real.

With the exception of a passing reference, software was hardly mentioned as part of the conversation. And yet, software will make or break the *SafeHome* product line. The engineering effort will succeed only if *SafeHome* software succeeds. The market will accept the product only if the software embedded within it properly meets the customer's (as yet unstated) needs. We'll follow the progression of *SafeHome* software engineering in many of the chapters that follow.

## 1.8 SUMMARY

Software is the key element in the evolution of computer-based systems and products and one of the most important technologies on the world stage. Over the past 50 years, software has evolved from a specialized problem solving and information analysis tool to an industry in itself. Yet we still have trouble developing high-quality software on time and within budget.

Software—programs, data, and descriptive information—addresses a wide array of technology and application areas. Legacy software continues to present special challenges to those who must maintain it.

Web-based systems and applications have evolved from simple collections of information content to sophisticated systems that present complex functionality and multimedia content. Although these WebApps have unique features and requirements, they are software nonetheless.

Software engineering encompasses process, methods, and tools that enable complex computer-based systems to be built in a timely manner with quality. The software process incorporates five framework activities—communication, planning, modeling, construction, and deployment—that are applicable to all software projects. Software engineering practice is a problem solving activity that follows a set of core principles.

A wide array of software myths continue to lead managers and practitioners astray, even as our collective knowledge of software and the technologies required to build it grows. As you learn more about software engineering, you'll begin to understand why these myths should be debunked whenever they are encountered.

## PROBLEMS AND POINTS TO PONDER

- 1.1.** Provide at least five additional examples of how the law of unintended consequences applies to computer software.
- 1.2.** Provide a number of examples (both positive and negative) that indicate the impact of software on our society.
- 1.3.** Develop your own answers to the five questions asked at the beginning of Section 1.1. Discuss them with your fellow students.
- 1.4.** Many modern applications change frequently—before they are presented to the end user and then after the first version has been put into use. Suggest a few ways to build software to stop deterioration due to change.

- 1.5.** Consider the seven software categories presented in Section 1.1.2. Do you think that the same approach to software engineering can be applied for each? Explain your answer.
- 1.6.** Figure 1.3 places the three software engineering layers on top of a layer entitled “a quality focus.” This implies an organizational quality program such as total quality management. Do a bit of research and develop an outline of the key tenets of a total quality management program.
- 1.7.** Is software engineering applicable when WebApps are built? If so, how might it be modified to accommodate the unique characteristics of WebApps?
- 1.8.** As software becomes more pervasive, risks to the public (due to faulty programs) become an increasingly significant concern. Develop a doomsday but realistic scenario in which the failure of a computer program could do great harm (either economic or human).
- 1.9.** Describe a process framework in your own words. When we say that framework activities are applicable to all projects, does this mean that the same work tasks are applied for all projects, regardless of size and complexity? Explain.
- 1.10.** Umbrella activities occur throughout the software process. Do you think they are applied evenly across the process, or are some concentrated in one or more framework activities.
- 1.11.** Add two additional myths to the list presented in Section 1.6. Also state the reality that accompanies the myth.

## FURTHER READINGS AND INFORMATION SOURCES<sup>18</sup>

There are literally thousands of books written about computer software. The vast majority discuss programming languages or software applications, but a few discuss software itself. Pressman and Herron (*Software Shock*, Dorset House, 1991) presented an early discussion (directed at the layperson) of software and the way professionals build it. Negroponte’s best-selling book (*Being Digital*, Alfred A. Knopf, Inc., 1995) provides a view of computing and its overall impact in the twenty-first century. DeMarco (*Why Does Software Cost So Much?* Dorset House, 1995) has produced a collection of amusing and insightful essays on software and the process through which it is developed.

Minasi (*The Software Conspiracy: Why Software Companies Put out Faulty Products, How They Can Hurt You, and What You Can Do*, McGraw-Hill, 2000) argues that the “modern scourge” of software bugs can be eliminated and suggests ways to accomplish this. Compaine (*Digital Divide: Facing a Crisis or Creating a Myth*, MIT Press, 2001) argues that the “divide” between those who have access to information resources (e.g., the Web) and those that do not is narrowing as we move into the first decade of this century. Books by Greenfield (*Everyware: The Dawning Age of Ubiquitous Computing*, New Riders Publishing, 2006) and Loke (*Context-Aware Pervasive Systems: Architectures for a New Breed of Applications*, Auerbach, 2006) introduce the concept of “open-world” software and predict a wireless environment in which software must adapt to requirements that emerge in real time.

The current state of the software engineering and the software process can best be determined from publications such as *IEEE Software*, *IEEE Computer*, *CrossTalk*, and *IEEE Transactions on Software Engineering*. Industry periodicals such as *Application Development Trends* and *Cutter*

<sup>18</sup> The *Further Reading and Information Sources* section presented at the conclusion of each chapter presents a brief overview of print sources that can help to expand your understanding of the major topics presented in the chapter. I have created a comprehensive website to support *Software Engineering: A Practitioner’s Approach* at [www.mhhe.com/compsci/pressman](http://www.mhhe.com/compsci/pressman). Among the many topics addressed within the website are chapter-by-chapter software engineering resources to Web-based information that can complement the material presented in each chapter. An Amazon.com link to every book noted in this section is contained within these resources.

*IT Journal* often contain articles on software engineering topics. The discipline is “summarized” every year in the *Proceeding of the International Conference on Software Engineering*, sponsored by the IEEE and ACM, and is discussed in depth in journals such as *ACM Transactions on Software Engineering and Methodology*, *ACM Software Engineering Notes*, and *Annals of Software Engineering*. Tens of thousands of websites are dedicated to software engineering and the software process.

Many books addressing the software process and software engineering have been published in recent years. Some present an overview of the entire process, while others delve into a few important topics to the exclusion of others. Among the more popular offerings (in addition to this book!) are

Abra, A., and J. Moore, *SWEBOk: Guide to the Software Engineering Body of Knowledge*, IEEE, 2002.

Andersson, E., et al., *Software Engineering for Internet Applications*, The MIT Press, 2006.

Christensen, M., and R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.

Glass, R., *Fact and Fallacies of Software Engineering*, Addison-Wesley, 2002.

Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, 2d ed., Addison-Wesley, 2008.

Jalote, P., *An Integrated Approach to Software Engineering*, Springer, 2006.

Pfleeger, S., *Software Engineering: Theory and Practice*, 3d ed., Prentice-Hall, 2005.

Schach, S., *Object-Oriented and Classical Software Engineering*, 7th ed., McGraw-Hill, 2006.

Sommerville, I., *Software Engineering*, 8th ed., Addison-Wesley, 2006.

Tsui, F., and O. Karam, *Essentials of Software Engineering*, Jones & Bartlett Publishers, 2006.

Many software engineering standards have been published by the IEEE, ISO, and their standards organizations over the past few decades. Moore (*The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006) provides a useful survey of relevant standards and how they apply to real projects.

A wide variety of information sources on software engineering and the software process are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process can be found at the SEPA website: [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm).



## THE SOFTWARE PROCESS

In this part of *Software Engineering: A Practitioner's Approach* you'll learn about the process that provides a framework for software engineering practice. These questions are addressed in the chapters that follow:

- What is a software process?
- What are the generic framework activities that are present in every software process?
- How are processes modeled and what are process patterns?
- What are the prescriptive process models and what are their strengths and weaknesses?
- Why is *agility* a watchword in modern software engineering work?
- What is agile software development and how does it differ from more traditional process models?

Once these questions are answered you'll be better prepared to understand the context in which software engineering practice is applied.

## CHAPTER

# 2

# PROCESS MODELS

### KEY CONCEPTS

component-based development	.....50
concurrent models	.....48
evolutionary process models	.....42
formal methods model	.....51
generic process model	.....31
incremental process models	.....41
personal software process	.....57
prescriptive process models	.....38
process patterns	.....35
task set	.....34
team software process	.....58
Unified Process	.....53

In a fascinating book that provides an economist's view of software and software engineering, Howard Baetjer, Jr. [Bae98], comments on the software process:

Because software, like all capital, is embodied knowledge, and because that knowledge is initially dispersed, tacit, latent, and incomplete in large measure, software development is a social learning process. The process is a dialogue in which the knowledge that must become the software is brought together and embodied in the software. The process provides interaction between users and designers, between users and evolving tools, and between designers and evolving tools [technology]. It is an iterative process in which the evolving tool itself serves as the medium for communication, with each new round of the dialogue eliciting more useful knowledge from the people involved.

Indeed, building computer software is an iterative social learning process, and the outcome, something that Baetjer would call "software capital," is an embodiment of knowledge collected, distilled, and organized as the process is conducted.

### QUICK LOOK

**What is it?** When you work to build a product or system, it's important to go through a series of predictable steps—a road map that helps you create a timely, high-quality result. The road map that you follow is called a "software process."

**Who does it?** Software engineers and their managers adapt the process to their needs and then follow it. In addition, the people who have requested the software have a role to play in the process of defining, building, and testing it.

**Why is it important?** Because it provides stability, control, and organization to an activity that can, if left uncontrolled, become quite chaotic. However, a modern software engineering approach must be "agile." It must demand only those activities, controls, and work products that are appropriate for the project team and the product that is to be produced.

**What are the steps?** At a detailed level, the process that you adopt depends on the software that you're building. One process might be appropriate for creating software for an aircraft avionics system, while an entirely different process would be indicated for the creation of a website.

**What is the work product?** From the point of view of a software engineer, the work products are the programs, documents, and data that are produced as a consequence of the activities and tasks defined by the process.

**How do I ensure that I've done it right?** There are a number of software process assessment mechanisms that enable organizations to determine the "maturity" of their software process. However, the quality, timeliness, and long-term viability of the product you build are the best indicators of the efficacy of the process that you use.

But what exactly is a software process from a technical point of view? Within the context of this book, I define a *software process* as a framework for the activities, actions, and tasks that are required to build high-quality software. Is “process” synonymous with software engineering? The answer is “yes and no.” A software process defines the approach that is taken as software is engineered. But software engineering also encompasses technologies that populate the process—technical methods and automated tools.

More important, software engineering is performed by creative, knowledgeable people who should adapt a mature software process so that it is appropriate for the products that they build and the demands of their marketplace.

## 2.1 A GENERIC PROCESS MODEL

In Chapter 1, a process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.

The software process is represented schematically in Figure 2.1. Referring to the figure, each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a *task set* that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

As I discussed in Chapter 1, a generic process framework for software engineering defines five framework activities—**communication, planning, modeling, construction, and deployment**. In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

You should note that one important aspect of the software process has not yet been discussed. This aspect—called *process flow*—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in Figure 2.2.

A *linear process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 2.2a). An *iterative process flow* repeats one or more of the activities before proceeding to the next (Figure 2.2b). An *evolutionary process flow* executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software (Figure 2.2c). A *parallel process flow* (Figure 2.2d) executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).

### KEY POINT

The hierarchy of technical work within the software process is activities, encompassing actions, populated by tasks.

### Quote:

“We think that software developers are missing a vital truth: most organizations don’t know what they do. They think they know, but they don’t know.”

Tom DeMarco

**FIGURE 2.1**

A software process framework

## Software process

### Process framework

#### Umbrella activities

##### framework activity # 1

###### software engineering action #1.1

Task sets

:

###### software engineering action #1.k

Task sets

:

##### framework activity # n

###### software engineering action #n.1

Task sets

:

###### software engineering action #n.m

Task sets

work tasks  
work products  
quality assurance points  
project milestones

work tasks  
work products  
quality assurance points  
project milestones

work tasks  
work products  
quality assurance points  
project milestones

work tasks  
work products  
quality assurance points  
project milestones

### 2.1.1 Defining a Framework Activity

Although I have described five framework activities and provided a basic definition of each in Chapter 1, a software team would need significantly more information before it could properly execute any one of these activities as part of the software process. Therefore, you are faced with a key question: *What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?*

**FIGURE 2.2** Process flow

?

**How does a framework activity change as the nature of the project changes?**

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the communication activity might encompass little more than a phone call with the appropriate stakeholder. Therefore, the only necessary action is *phone conversation*, and the work tasks (the *task set*) that this action encompasses are:

1. Make contact with stakeholder via telephone.
2. Discuss requirements and take notes.

3. Organize notes into a brief written statement of requirements.
4. E-mail to stakeholder for review and approval.

If the project was considerably more complex with many stakeholders, each with a different set of (sometime conflicting) requirements, the communication activity might have six distinct actions (described in Chapter 5): *inception, elicitation, elaboration, negotiation, specification, and validation*. Each of these software engineering actions would have many work tasks and a number of distinct work products.

## KEY POINT

Different projects demand different task sets. The software team chooses the task set based on problem and project characteristics.

### 2.1.2 Identifying a Task Set

Referring again to Figure 2.1, each software engineering action (e.g., *elicitation*, an action associated with the communication activity) can be represented by a number of different *task sets*—each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones. You should choose a task set that best accommodates the needs of the project and the characteristics of your team. This implies that a software engineering action can be adapted to the specific needs of the software project and the characteristics of the project team.



#### Task Set

A task set defines the actual work to be done to accomplish the objectives of a software engineering action. For example, *elicitation* (more commonly called “requirements gathering”) is an important software engineering action that occurs during the communication activity. The goal of requirements gathering is to understand what various stakeholders want from the software that is to be built.

For a small, relatively simple project, the task set for requirements gathering might look like this:

1. Make a list of stakeholders for the project.
2. Invite all stakeholders to an informal meeting.
3. Ask each stakeholder to make a list of features and functions required.
4. Discuss requirements and build a final list.
5. Prioritize requirements.
6. Note areas of uncertainty.

For a larger, more complex software project, a different task set would be required. It might encompass the following work tasks:

1. Make a list of stakeholders for the project.
2. Interview each stakeholder separately to determine overall wants and needs.

#### INFO

3. Build a preliminary list of functions and features based on stakeholder input.
4. Schedule a series of facilitated application specification meetings.
5. Conduct meetings.
6. Produce informal user scenarios as part of each meeting.
7. Refine user scenarios based on stakeholder feedback.
8. Build a revised list of stakeholder requirements.
9. Use quality function deployment techniques to prioritize requirements.
10. Package requirements so that they can be delivered incrementally.
11. Note constraints and restrictions that will be placed on the system.
12. Discuss methods for validating the system.

Both of these task sets achieve “requirements gathering,” but they are quite different in their depth and formality. The software team chooses the task set that will allow it to achieve the goal of each action and still maintain quality and agility.

### 2.1.3 Process Patterns



What is a process pattern?



**note:**

"The repetition of patterns is quite a different thing than the repetition of parts. Indeed, the different parts will be unique because the patterns are the same."

Christopher Alexander



A pattern template provides a consistent means for describing a pattern.

Every software team encounters problems as it moves through the software process. It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly. A *process pattern*<sup>1</sup> describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem. Stated in more general terms, a process pattern provides you with a template [Amb98]—a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.

Patterns can be defined at any level of abstraction.<sup>2</sup> In some cases, a pattern might be used to describe a problem (and solution) associated with a complete process model (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating).

Ambler [Amb98] has proposed a template for describing a process pattern:

**Pattern Name.** The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

**Forces.** The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

**Type.** The pattern type is specified. Ambler [Amb98] suggests three types:

1. *Stage pattern*—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity). An example of a stage pattern might be **EstablishingCommunication**. This pattern would incorporate the task pattern **RequirementsGathering** and others.
2. *Task pattern*—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **RequirementsGathering** is a task pattern).
3. *Phase pattern*—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be **SpiralModel** or **Prototyping**.<sup>3</sup>

<sup>1</sup> A detailed discussion of patterns is presented in Chapter 12.

<sup>2</sup> Patterns are applicable to many software engineering activities. Analysis, design, and testing patterns are discussed in Chapters 7, 9, 10, 12, and 14. Patterns and “antipatterns” for project management activities are discussed in Part 4 of this book.

<sup>3</sup> These phase patterns are discussed in Section 2.3.3.

**Initial context.** Describes the conditions under which the pattern applies.

Prior to the initiation of the pattern: (1) What organizational or team-related activities have already occurred? (2) What is the entry state for the process? (3) What software engineering information or project information already exists?

For example, the **Planning** pattern (a stage pattern) requires that (1) customers and software engineers have established a collaborative communication; (2) successful completion of a number of task patterns [specified] for the **Communication** pattern has occurred; and (3) the project scope, basic business requirements, and project constraints are known.

**Problem.** The specific problem to be solved by the pattern.

**Solution.** Describes how to implement the pattern successfully. This section describes how the initial state of the process (that exists before the pattern is implemented) is modified as a consequence of the initiation of the pattern. It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern.

**Resulting Context.** Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern:

- (1) What organizational or team-related activities must have occurred?
- (2) What is the exit state for the process? (3) What software engineering information or project information has been developed?

**Related Patterns.** Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form. For example, the stage pattern **Communication** encompasses the task patterns: **ProjectTeam**, **CollaborativeGuidelines**, **ScopeIsolation**, **RequirementsGathering**, **ConstraintDescription**, and **ScenarioCreation**.

**Known Uses and Examples.** Indicate the specific instances in which the pattern is applicable. For example, **Communication** is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the deployment activity is under way.

Process patterns provide an effective mechanism for addressing problems associated with any software process. The patterns enable you to develop a hierarchical process description that begins at a high level of abstraction (a phase pattern). The description is then refined into a set of stage patterns that describe framework activities and are further refined in a hierarchical fashion into more detailed task patterns for each stage pattern. Once process patterns have been developed, they can be reused for the definition of process variants—that is, a customized process model can be defined by a software team using the patterns as building blocks for the process model.

### WebRef

Comprehensive resources on process patterns can be found at [www.ambyssoft.com/processPatternsPage.html](http://www.ambyssoft.com/processPatternsPage.html).



### An Example Process Pattern

The following abbreviated process pattern describes an approach that may be applicable when stakeholders have a general idea of what must be done but are unsure of specific software requirements.

#### **Pattern name.** RequirementsUnclear

**Intent.** This pattern describes an approach for building a model (a prototype) that can be assessed iteratively by stakeholders in an effort to identify or solidify software requirements.

**Type.** Phase pattern.

**Initial context.** The following conditions must be met prior to the initiation of this pattern: (1) stakeholders have been identified; (2) a mode of communication between stakeholders and the software team has been established; (3) the overriding software problem to be solved has been identified by stakeholders; (4) an initial understanding of project scope, basic business requirements, and project constraints has been developed.

**Problem.** Requirements are hazy or nonexistent, yet there is clear recognition that there is a problem to be

### INFO

solved, and the problem must be addressed with a software solution. Stakeholders are unsure of what they want; that is, they cannot describe software requirements in any detail.

**Solution.** A description of the prototyping process would be presented here and is described later in Section 2.3.3.

**Resulting context.** A software prototype that identifies basic requirements (e.g., modes of interaction, computational features, processing functions) is approved by stakeholders. Following this, (1) the prototype may evolve through a series of increments to become the production software or (2) the prototype may be discarded and the production software built using some other process pattern.

**Related patterns.** The following patterns are related to this pattern: **CustomerCommunication**, **IterativeDesign**, **IterativeDevelopment**, **CustomerAssessment**, **RequirementExtraction**.

**Known uses and examples.** Prototyping is recommended when requirements are uncertain.

## 2.2 PROCESS ASSESSMENT AND IMPROVEMENT

### KEY POINT

Assessment attempts to understand the current state of the software process with the intent of improving it.

What formal techniques are available for assessing the software process?

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics (Chapters 14 and 16). Process patterns must be coupled with solid software engineering practice (Part 2 of this book). In addition, the process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.<sup>4</sup>

A number of different approaches to software process assessment and improvement have been proposed over the past few decades:

#### **Standard CMMI Assessment Method for Process Improvement**

**(SCAMPI)**—provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment [SEI00].

4 The SEI's CMMI [CMM07] describes the characteristics of a software process and the criteria for a successful process in voluminous detail.

**Q**uote:

"Software organizations have exhibited significant shortcomings in their ability to capitalize on the experiences gained from completed projects."

NASA

**CMM-Based Appraisal for Internal Process Improvement (CBA IPI)—**

provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment [Dun01].

**SPICE (ISO/IEC 15504)**—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process [ISO08].

**ISO 9001:2000 for Software**—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies [Ant06].

A more detailed discussion of software assessment and process improvement methods is presented in Chapter 30.

## 2.3 PRESCRIPTIVE PROCESS MODELS

Prescriptive process models were originally proposed to bring order to the chaos of software development. History has indicated that these traditional models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams. However, software engineering work and the product that it produces remain on “the edge of chaos.”

In an intriguing paper on the strange relationship between order and chaos in the software world, Nogueira and his colleagues [Nog00] state

The edge of chaos is defined as “a natural state between order and chaos, a grand compromise between structure and surprise” [Kau95]. The edge of chaos can be visualized as an unstable, partially structured state. . . . It is unstable because it is constantly attracted to chaos or to absolute order.

We have the tendency to think that order is the ideal state of nature. This could be a mistake. Research . . . supports the theory that operation away from equilibrium generates creativity, self-organized processes, and increasing returns [Roo96]. Absolute order means the absence of variability, which could be an advantage under unpredictable environments. Change occurs when there is some structure so that the change can be organized, but not so rigid that it cannot occur. Too much chaos, on the other hand, can make coordination and coherence impossible. Lack of structure does not always mean disorder.

The philosophical implications of this argument are significant for software engineering. If prescriptive process models<sup>5</sup> strive for structure and order, are they inappropriate for a software world that thrives on change? Yet, if we reject traditional process

<sup>5</sup> Prescriptive process models are sometimes referred to as “traditional” process models.

## KEY POINT

Prescriptive process models define a prescribed set of process elements and a predictable process work flow.

models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

There are no easy answers to these questions, but there are alternatives available to software engineers. In the sections that follow, I examine the prescriptive process approach in which order and project consistency are dominant issues. I call them “prescriptive” because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.

All software process models can accommodate the generic framework activities described in Chapter 1, but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner.

### 2.3.1 The Waterfall Model

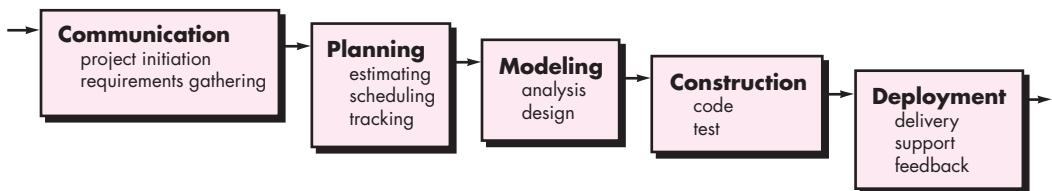
There are times when the requirements for a problem are well understood—when work flows from **communication** through **deployment** in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made (e.g., an adaptation to accounting software that has been mandated because of changes to government regulations). It may also occur in a limited number of new development efforts, but only when requirements are well defined and reasonably stable.

The *waterfall model*, sometimes called the *classic life cycle*, suggests a systematic, sequential approach<sup>6</sup> to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software (Figure 2.3).

A variation in the representation of the waterfall model is called the *V-model*. Represented in Figure 2.4, the V-model [Buc99] depicts the relationship of quality

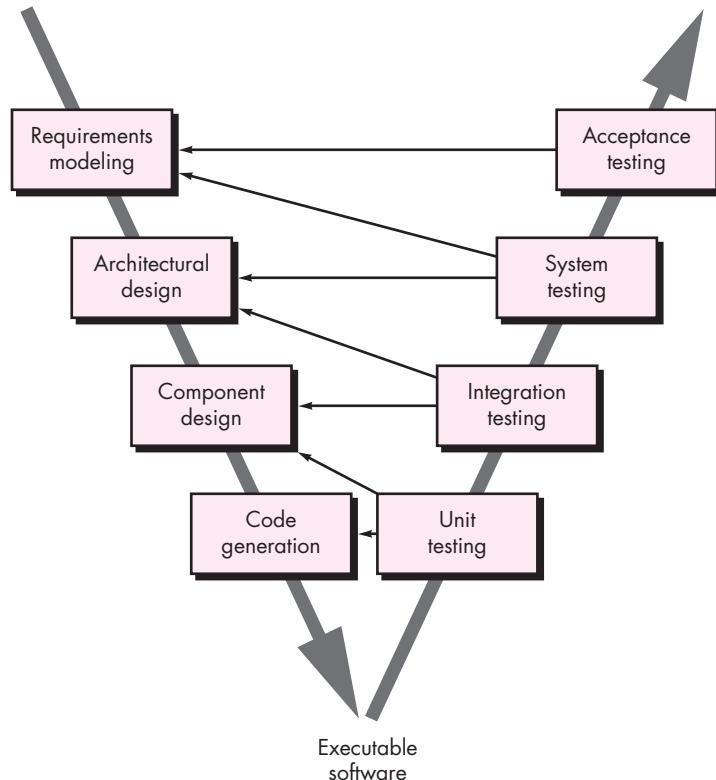
**FIGURE 2.3**

The waterfall model



<sup>6</sup> Although the original waterfall model proposed by Winston Royce [Roy70] made provision for “feedback loops,” the vast majority of organizations that apply this process model treat it as if it were strictly linear.

**FIGURE 2.4**  
The V-model



### KEY POINT

The V-model illustrates how verification and validation actions are associated with earlier engineering actions.

assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.<sup>7</sup> In reality, there is no fundamental difference between the classic life cycle and the V-model. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

The waterfall model is the oldest paradigm for software engineering. However, over the past three decades, criticism of this process model has caused even ardent supporters to question its efficacy [Han95]. Among the problems that are sometimes encountered when the waterfall model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

### Why does the waterfall model sometimes fail?

<sup>7</sup> A detailed discussion of quality assurance actions is presented in Part 3 of this book.

2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

**QUOTE:**

"Too often, software work follows the first law of bicycling: No matter where you're going, it's uphill and against the wind."

**Author unknown**

In an interesting analysis of actual projects, Bradac [Bra94] found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking states tend to be more prevalent at the beginning and end of a linear sequential process.

Today, software work is fast-paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work. However, it can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.

### 2.3.2 Incremental Process Models



The incremental model delivers a series of releases, called increments, that provide progressively more functionality for the customer as each increment is delivered.

There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.

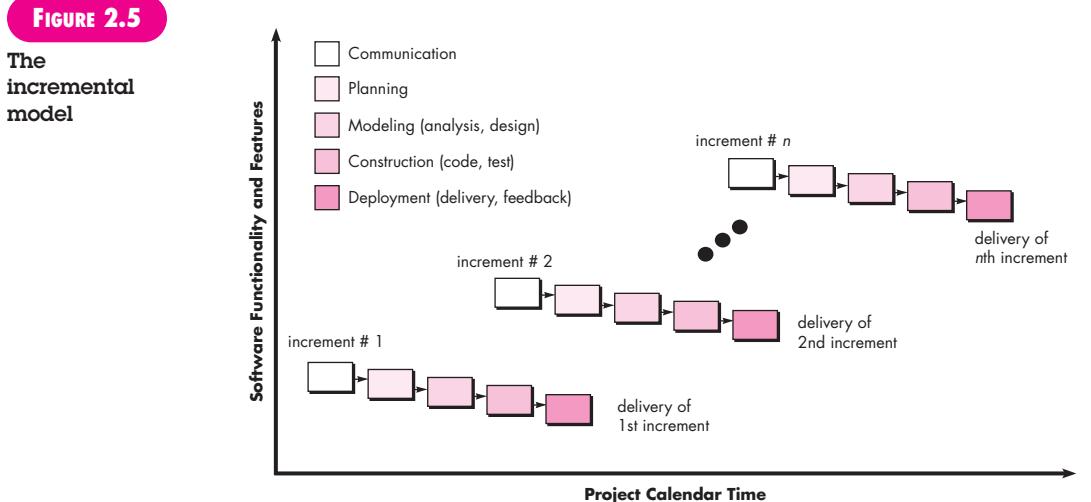
The *incremental* model combines elements of linear and parallel process flows discussed in Section 2.1. Referring to Figure 2.5, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software [McD93] in a manner that is similar to the increments produced by an evolutionary process flow (Section 2.3.3).



Your customer demands delivery by a date that is impossible to meet. Suggest delivering one or more increments by that date and the rest of the software (additional increments) later.

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

When an incremental model is used, the first increment is often a *core product*. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/or evaluation, a



plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.<sup>8</sup>

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

## KEY POINT

Evolutionary process models produce an increasingly more complete version of the software with each iteration.

### 2.3.3 Evolutionary Process Models

Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to

<sup>8</sup> It is important to note that an incremental philosophy is also used for all “agile” process models discussed in Chapter 3.

meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined. In these and similar situations, you need a process model that has been explicitly designed to accommodate a product that evolves over time.

Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software. In the paragraphs that follow, I present two common evolutionary process models.

**NOTE:**

"Plan to throw one away. You will do that, anyway. Your only choice is whether to try to sell the throwaway to customers."

Frederick P. Brooks

**ADVICE**

When your customer has a legitimate need, but is clueless about the details, develop a prototype as a first step.

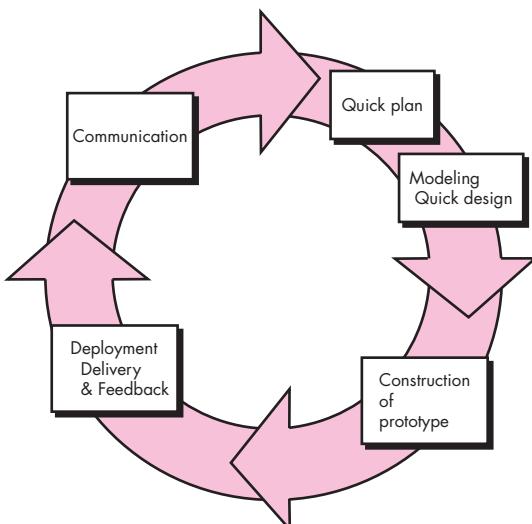
**Prototyping.** Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a *prototyping paradigm* may offer the best approach.

Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models noted in this chapter. Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.

The prototyping paradigm (Figure 2.6) begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display

**FIGURE 2.6**

The prototyping paradigm



formats). The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools (e.g., report generators and window managers) that enable working programs to be generated quickly.

But what do you do with the prototype when it has served the purpose described earlier? Brooks [Bro95] provides one answer:

In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved.

The prototype can serve as “the first system.” The one that Brooks recommends you throw away. But this may be an idealized view. Although some prototypes are built as “throwaways,” others are evolutionary in the sense that the prototype slowly evolves into the actual system.

Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. Yet, prototyping can be problematic for the following reasons:



*Resist pressure to extend a rough prototype into a production product. Quality almost always suffers as a result.*

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven’t considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that “a few fixes” be applied to make the prototype a working product. Too often, software development management relents.
2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, all stakeholders should agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part), and the actual software is engineered with an eye toward quality.

## SAFEHOME



### Selecting a Process Model, Part 1

**The scene:** Meeting room for the software engineering group at CPI Corporation, a (fictional) company that makes consumer products for home and commercial use.

**The players:** Lee Warren, engineering manager; Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member; and Ed Robbins, software team member.

#### The conversation:

**Lee:** So let's recapitulate. I've spent some time discussing the *SafeHome* product line as we see it at the moment. No doubt, we've got a lot of work to do to simply define the thing, but I'd like you guys to begin thinking about how you're going to approach the software part of this project.

**Doug:** Seems like we've been pretty disorganized in our approach to software in the past.

**Ed:** I don't know, Doug, we always got product out the door.

**Doug:** True, but not without a lot of grief, and this project looks like it's bigger and more complex than anything we've done in the past.

**Jamie:** Doesn't look that hard, but I agree . . . our ad hoc approach to past projects won't work here, particularly if we have a very tight time line.

**Doug (smiling):** I want to be a bit more professional in our approach. I went to a short course last week and learned a lot about software engineering . . . good stuff. We need a process here.

**Jamie (with a frown):** My job is to build computer programs, not push paper around.

**Doug:** Give it a chance before you go negative on me. Here's what I mean. [Doug proceeds to describe the process framework described in this chapter and the prescriptive process models presented to this point.]

**Doug:** So anyway, it seems to me that a linear model is not for us . . . assumes we have all requirements up front and, knowing this place, that's not likely.

**Vinod:** Yeah, and it sounds way too IT-oriented . . . probably good for building an inventory control system or something, but it's just not right for *SafeHome*.

**Doug:** I agree.

**Ed:** That prototyping approach seems OK. A lot like what we do here anyway.

**Vinod:** That's a problem. I'm worried that it doesn't provide us with enough structure.

**Doug:** Not to worry. We've got plenty of other options, and I want you guys to pick what's best for the team and best for the project.

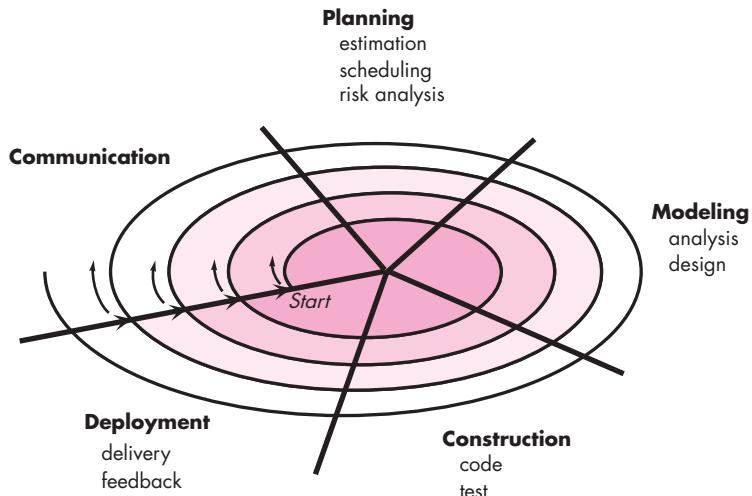
**The Spiral Model.** Originally proposed by Barry Boehm [Boe88], the *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software. Boehm [Boe01a] describes the model in the following manner:

The spiral development model is a *risk-driven process model* generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a *cyclic* approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of *anchor point milestones* for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

**FIGURE 2.7**

A typical spiral model



### KEY POINT

The spiral model can be adapted to apply throughout the entire life cycle of an application, from concept development to maintenance.

#### WebRef

Useful information about the spiral model can be obtained at:

[www.sei.cmu.edu/publications/documents/00\\_reports/00sr008.html](http://www.sei.cmu.edu/publications/documents/00_reports/00sr008.html)

A spiral model is divided into a set of framework activities defined by the software engineering team. For illustrative purposes, I use the generic framework activities discussed earlier.<sup>9</sup> Each of the framework activities represent one segment of the spiral path illustrated in Figure 2.7. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk (Chapter 28) is considered as each revolution is made. *Anchor point milestones*—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.

The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.

Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. Therefore, the first circuit around the spiral might represent a “concept development project” that starts at the core of the spiral and continues for multiple iterations<sup>10</sup> until concept

9 The spiral model discussed in this section is a variation on the model proposed by Boehm. For further information on the original spiral model, see [Boe88]. More recent discussion of Boehm's spiral model can be found in [Boe98].

10 The arrows pointing inward along the axis separating the **deployment** region from the **communication** region indicate a potential for local iteration along the same spiral path.



*If your management demands fixed-budget development (generally a bad idea), the spiral can be a problem. As each circuit is completed, project cost is revisited and revised.*

development is complete. If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a “new product development project” commences. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a “product enhancement project.” In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more important, enables you to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world. The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.

**“I’m only this far and only tomorrow leads my way.”**

**Dave Matthews Band**

## SAFEHOME



### Selecting a Process Model, Part 2

**The scene:** Meeting room for the software engineering group at CPI Corporation, a company that makes consumer products for home and commercial use.

**The players:** Lee Warren, engineering manager; Doug Miller, software engineering manager; Vinod and Jamie, members of the software engineering team.

**The conversation:** [Doug describes evolutionary process options.]

**Jamie:** Now I see something I like. An incremental approach makes sense, and I really like the flow of that spiral model thing. That’s keepin’ it real.

**Vinod:** I agree. We deliver an increment, learn from customer feedback, replan, and then deliver another increment. It also fits into the nature of the product. We

can have something on the market fast and then add functionality with each version, er, increment.

**Lee:** Wait a minute. Did you say that we regenerate the plan with each tour around the spiral, Doug? That’s not so great; we need one plan, one schedule, and we’ve got to stick to it.

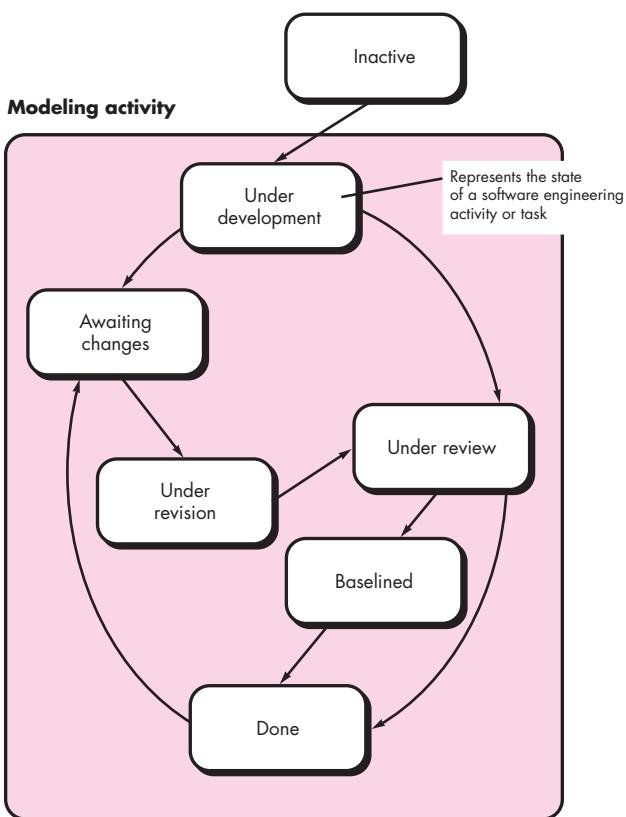
**Doug:** That’s old-school thinking, Lee. Like the guys said, we’ve got to keep it real. I submit that it’s better to tweak the plan as we learn more and as changes are requested. It’s way more realistic. What’s the point of a plan if it doesn’t reflect reality?

**Lee** (frowning): I suppose so, but . . . senior management’s not going to like this . . . they want a fixed plan.

**Doug** (smiling): Then you’ll have to reeducate them, buddy.

**FIGURE 2.8**

One element of the concurrent process model



### 2.3.4 Concurrent Models

The *concurrent development model*, sometimes called *concurrent engineering*, allows a software team to represent iterative and concurrent elements of any of the process models described in this chapter. For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following software engineering actions: prototyping, analysis, and design.<sup>11</sup>

Figure 2.8 provides a schematic representation of one software engineering activity within the modeling activity using a concurrent modeling approach. The activity—**modeling**—may be in any one of the states<sup>12</sup> noted at any given time. Similarly, other activities, actions, or tasks (e.g., **communication** or **construction**) can be represented in an analogous manner. All software engineering activities exist concurrently but reside in different states.



The concurrent model is often more appropriate for product engineering projects where different engineering teams are involved.

<sup>11</sup> It should be noted that analysis and design are complex tasks that require substantial discussion.

Part 2 of this book considers these topics in detail.

<sup>12</sup> A *state* is some externally observable mode of behavior.

For example, early in a project the communication activity (not shown in the figure) has completed its first iteration and exists in the **awaiting changes** state. The modeling activity (which existed in the **inactive** state while initial communication was completed, now makes a transition into the **under development** state. If, however, the customer indicates that changes in requirements must be made, the modeling activity moves from the **under development** state into the **awaiting changes** state.

Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks. For example, during early stages of design (a major software engineering action that occurs during the modeling activity), an inconsistency in the requirements model is uncovered. This generates the event *analysis model correction*, which will trigger the requirements analysis action from the **done** state into the **awaiting changes** state.

Concurrent modeling is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities, actions, and tasks to a sequence of events, it defines a process network. Each activity, action, or task on the network exists simultaneously with other activities, actions, or tasks. Events generated at one point in the process network trigger transitions among the states.

**note:**

"Every process in your organization has a customer, and without a customer a process has no purpose."

V. Daniel Hunt

### 2.3.5 A Final Word on Evolutionary Processes

I have already noted that modern computer software is characterized by continual change, by very tight time lines, and by an emphatic need for customer-user satisfaction. In many cases, time-to-market is the most important management requirement. If a market window is missed, the software project itself may be meaningless.<sup>13</sup>

Evolutionary process models were conceived to address these issues, and yet, as a general class of process models, they too have weaknesses. These are summarized by Nogueira and his colleagues [Nog00] :

Despite the unquestionable benefits of evolutionary software processes, we have some concerns. The first concern is that prototyping [and other more sophisticated evolutionary processes] poses a problem to project planning because of the uncertain number of cycles required to construct the product. Most project management and estimation techniques are based on linear layouts of activities, so they do not fit completely.

Second, evolutionary software processes do not establish the maximum speed of the evolution. If the evolutions occur too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand if the speed is too slow then productivity could be affected . . .

---

<sup>13</sup> It is important to note, however, that being the first to reach a market is no guarantee of success. In fact, many very successful software products have been second or even third to reach the market (learning from the mistakes of their predecessors).

Third, software processes should be focused on flexibility and extensibility rather than on high quality. This assertion sounds scary. However, we should prioritize the speed of the development over zero defects. Extending the development in order to reach high quality could result in a late delivery of the product, when the opportunity niche has disappeared. This paradigm shift is imposed by the competition on the edge of chaos.

Indeed, a software process that focuses on flexibility, extensibility, and speed of development over high quality does sound scary. And yet, this idea has been proposed by a number of well-respected software engineering experts (e.g., [You95], [Bac97]).

The intent of evolutionary models is to develop high-quality software<sup>14</sup> in an iterative or incremental manner. However, it is possible to use an evolutionary process to emphasize flexibility, extensibility, and speed of development. The challenge for software teams and their managers is to establish a proper balance between these critical project and product parameters and customer satisfaction (the ultimate arbiter of software quality).

## 2.4 SPECIALIZED PROCESS MODELS

Specialized process models take on many of the characteristics of one or more of the traditional models presented in the preceding sections. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.<sup>15</sup>

### 2.4.1 Component-Based Development

#### WebRef

Useful information on component-based development can be obtained at: [www.cbd-hq.com](http://www.cbd-hq.com).

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The *component-based development model* incorporates many of the characteristics of the spiral model. It is evolutionary in nature [Nie92], demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software components.

Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages<sup>16</sup> of classes. Regardless of the

<sup>14</sup> In this context software quality is defined quite broadly to encompass not only customer satisfaction, but also a variety of technical criteria discussed in Chapters 14 and 16.

<sup>15</sup> In some cases, these specialized process models might better be characterized as a collection of techniques or a “methodology” for accomplishing a specific software development goal. However, they do imply a process.

<sup>16</sup> Object-oriented concepts are discussed in Appendix 2 and are used throughout Part 2 of this book. In this context, a class encompasses a set of data and the procedures that process the data. A package of classes is a collection of related classes that work together to achieve some end result.

technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Your software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost if component reuse becomes part of your culture. Component-based development is discussed in more detail in Chapter 10.

### 2.4.2 The Formal Methods Model

The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called *cleanroom software engineering* [Mil87, Dye92], is currently applied by some software development organizations.

When formal methods (Chapter 21) are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through ad hoc review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable you to discover and correct errors that might otherwise go undetected.

Although not a mainstream approach, the formal methods model offers the promise of defect-free software. Yet, concern about its applicability in a business environment has been voiced:

If formal  
methods can  
demonstrate  
software  
correctness, why  
is it they are not  
widely used?

- The development of formal models is currently quite time consuming and expensive.
- Because few software developers have the necessary background to apply formal methods, extensive training is required.
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

These concerns notwithstanding, the formal methods approach has gained adherents among software developers who must build safety-critical software

(e.g., developers of aircraft avionics and medical devices) and among developers that would suffer severe economic hardship should software errors occur.

### 2.4.3 Aspect-Oriented Software Development

#### WebRef

A wide array of resources and information on AOP can be found at: [aosd.net](http://aosd.net).

#### KEY POINT

AOSD defines “aspects” that express customer concerns that cut across multiple system functions, features, and information.

Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modeled as components (e.g., object-oriented classes) and then constructed within the context of a system architecture. As modern computer-based systems become more sophisticated (and complex), certain *concerns*—customer required properties or areas of technical interest—span the entire architecture. Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).

When concerns cut across multiple system functions, features, and information, they are often referred to as *crosscutting concerns*. *Aspectual requirements* define those crosscutting concerns that have an impact across the software architecture. *Aspect-oriented software development* (AOSD), often referred to as *aspect-oriented programming* (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing *aspects*—“mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern” [Elr01].

Grundy [Gru02] provides further discussion of aspects in the context of what he calls *aspect-oriented component engineering* (AOCE):

AOCE uses a concept of horizontal slices through vertically-decomposed software components, called “aspects,” to characterize cross-cutting functional and non-functional properties of components. Common, systemic aspects include user interfaces, collaborative work, distribution, persistency, memory management, transaction processing, security, integrity and so on. Components may provide or require one or more “aspect details” relating to a particular aspect, such as a viewing mechanism, extensible affordance and interface kind (user interface aspects); event generation, transport and receiving (distribution aspects); data store/retrieve and indexing (persistency aspects); authentication, encoding and access rights (security aspects); transaction atomicity, concurrency control and logging strategy (transaction aspects); and so on. Each aspect detail has a number of properties, relating to functional and/or non-functional characteristics of the aspect detail.

A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both evolutionary and concurrent process models. The evolutionary model is appropriate as aspects are identified and then constructed. The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components and yet, aspects have a direct impact on these components. Hence, it is essential to

stantiate asynchronous communication between the software process activities applied to the engineering and construction of aspects and components.

A detailed discussion of aspect-oriented software development is best left to books dedicated to the subject. If you have further interest, see [Saf08], [Cla05], [Jac04], and [Gra03].



### Process Management

**Objective:** To assist in the definition, execution, and management of prescriptive process models.

**Mechanics:** Process management tools allow a software organization or team to define a complete software process model (framework activities, actions, tasks, QA checkpoints, milestones, and work products). In addition, the tools provide a road map as software engineers do technical work and a template for managers who must track and control the software process.

#### Representative Tools:<sup>17</sup>

GDPA, a research process definition tool suite, developed at Bremen University in Germany ([www.informatik](http://www.informatik.uni-bremen.de/uniform/gdpa/home.htm)

### SOFTWARE TOOLS

[.uni-bremen.de/uniform/gdpa/home.htm](http://uni-bremen.de/uniform/gdpa/home.htm), provides a wide array of process modeling and management functions.

SpeeDev, developed by SpeeDev Corporation ([www.speeudev.com](http://www.speeudev.com)) encompasses a suite of tools for process definition, requirements management, issue resolution, project planning, and tracking.

ProVision BPMx, developed by Proforma ([www.proformacorp.com](http://www.proformacorp.com)), is representative of many tools that assist in process definition and workflow automation.

A worthwhile listing of many different tools associated with the software process can be found at [www.processwave.net/Links/tool\\_links.htm](http://www.processwave.net/Links/tool_links.htm).

## 2.5 THE UNIFIED PROCESS

In their seminal book on the *Unified Process*, Ivar Jacobson, Grady Booch, and James Rumbaugh [Jac99] discuss the need for a “use case driven, architecture-centric, iterative and incremental” software process when they state:

Today, the trend in software is toward bigger, more complex systems. That is due in part to the fact that computers become more powerful every year, leading users to expect more from them. This trend has also been influenced by the expanding use of the Internet for exchanging all kinds of information. . . . Our appetite for ever-more sophisticated software grows as we learn from one product release to the next how the product could be improved. We want software that is better adapted to our needs, but that, in turn, merely makes the software more complex. In short, we want more.

In some ways the Unified Process is an attempt to draw on the best features and characteristics of traditional software process models, but characterize them in a way that implements many of the best principles of agile software development

---

17 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

(Chapter 3). The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system (the use case<sup>18</sup>). It emphasizes the important role of software architecture and "helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse" [Jac99]. It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

### 2.5.1 A Brief History

During the early 1990s James Rumbaugh [Rum91], Grady Booch [Boo94], and Ivar Jacobson [Jac92] began working on a "unified method" that would combine the best features of each of their individual object-oriented analysis and design methods and adopt additional features proposed by other experts (e.g., [Wir90]) in object-oriented modeling. The result was UML—a *unified modeling language* that contains a robust notation for the modeling and development of object-oriented systems. By 1997, UML became a de facto industry standard for object-oriented software development.

UML is used throughout Part 2 of this book to represent both requirements and design models. Appendix 1 presents an introductory tutorial for those who are unfamiliar with basic UML notation and modeling rules. A comprehensive presentation of UML is best left to textbooks dedicated to the subject. Recommended books are listed in Appendix 1.

UML provided the necessary technology to support object-oriented software engineering practice, but it did not provide the process framework to guide project teams in their application of the technology. Over the next few years, Jacobson, Rumbaugh, and Booch developed the *Unified Process*, a framework for object-oriented software engineering using UML. Today, the Unified Process (UP) and UML are widely used on object-oriented projects of all kinds. The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

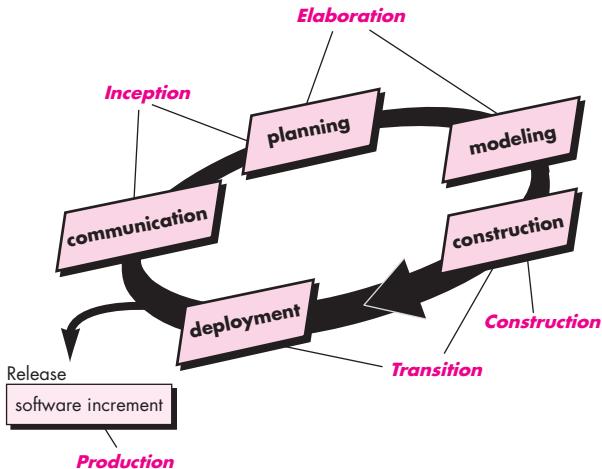
### 2.5.2 Phases of the Unified Process<sup>19</sup>

Earlier in this chapter, I discussed five generic framework activities and argued that they may be used to describe any software process model. The Unified Process is no exception. Figure 2.9 depicts the "phases" of the UP and relates them to the generic activities that have been discussed in Chapter 1 and earlier in this chapter.

---

<sup>18</sup> A *use case* (Chapter 5) is a text narrative or template that describes a system function or feature from the user's point of view. A use case is written by the user and serves as a basis for the creation of a more comprehensive requirements model.

<sup>19</sup> The Unified Process is sometimes called the *Rational Unified Process* (RUP) after the Rational Corporation (subsequently acquired by IBM), an early contributor to the development and refinement of the UP and a builder of complete environments (tools and technology) that support the process.

**FIGURE 2.9**
**The Unified Process**


**UP phases** are similar in intent to the generic framework activities defined in this book.

The *inception* phase of the UP encompasses both customer communication and planning activities. By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is proposed; and a plan for the iterative, incremental nature of the ensuing project is developed. Fundamental business requirements are described through a set of preliminary use cases (Chapter 5) that describe which features and functions each major class of users desires. Architecture at this point is nothing more than a tentative outline of major subsystems and the function and features that populate them. Later, the architecture will be refined and expanded into a set of models that will represent different views of the system. Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis for the phases that are to be applied as the software increment is developed.

The *elaboration* phase encompasses the communication and modeling activities of the generic process model (Figure 2.9). Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use case model, the requirements model, the design model, the implementation model, and the deployment model. In some cases, elaboration creates an “executable architectural baseline” [Arl02] that represents a “first cut” executable system.<sup>20</sup> The architectural baseline demonstrates the viability of the architecture but does not provide all features and functions required to use the system. In addition, the plan is carefully reviewed at the culmination of the elaboration phase to ensure that scope, risks, and delivery dates remain reasonable. Modifications to the plan are often made at this time.

<sup>20</sup> It is important to note that the architectural baseline is not a prototype in that it is not thrown away. Rather, the baseline is fleshed out during the next UP phase.

**WebRef**

An interesting discussion of the UP in the context of agile development can be found at [www.ambyssoft.com/unifiedprocess/agileUP.html](http://www.ambyssoft.com/unifiedprocess/agileUP.html).

The *construction phase* of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users. To accomplish this, requirements and design models that were started during the elaboration phase are completed to reflect the final version of the software increment. All necessary and required features and functions for the software increment (i.e., the release) are then implemented in source code. As components are being implemented, unit tests<sup>21</sup> are designed and executed for each. In addition, integration activities (component assembly and integration testing) are conducted. Use cases are used to derive a suite of acceptance tests that are executed prior to the initiation of the next UP phase.

The *transition phase* of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity. Software is given to end users for beta testing and user feedback reports both defects and necessary changes. In addition, the software team creates the necessary support information (e.g., user manuals, troubleshooting guides, installation procedures) that is required for the release. At the conclusion of the transition phase, the software increment becomes a usable software release.

The *production phase* of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

It is likely that at the same time the construction, transition, and production phases are being conducted, work may have already begun on the next software increment. This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.

A software engineering workflow is distributed across all UP phases. In the context of UP, a *workflow* is analogous to a task set (described earlier in this chapter). That is, a workflow identifies the tasks required to accomplish an important software engineering action and the work products that are produced as a consequence of successfully completing the tasks. It should be noted that not every task identified for a UP workflow is conducted for every software project. The team adapts the process (actions, tasks, subtasks, and work products) to meet its needs.

## 2.6 PERSONAL AND TEAM PROCESS MODELS

The best software process is one that is close to the people who will be doing the work. If a software process model has been developed at a corporate or organizational level, it can be effective only if it is amenable to significant adaptation to meet

<sup>21</sup> A comprehensive discussion of software testing (including *unit tests*) is presented in Chapters 17 through 20.

**note:**

"A person who is successful has simply formed the habit of doing things that unsuccessful people will not do."

Dexter Yager

**WebRef**

A wide array of resources for PSP can be found at [www.ipd.uka.de/PSP/](http://www.ipd.uka.de/PSP/).

**?** What framework activities are used during PSP?

the needs of the project team that is actually doing software engineering work. In an ideal setting, you would create a process that best fits your needs, and at the same time, meets the broader needs of the team and the organization. Alternatively, the team itself can create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization. Watts Humphrey ([Hum97] and [Hum00]) argues that it is possible to create a "personal software process" and/or a "team software process." Both require hard work, training, and coordination, but both are achievable.<sup>22</sup>

### 2.6.1 Personal Software Process (PSP)

Every developer uses some process to build computer software. The process may be haphazard or ad hoc; may change on a daily basis; may not be efficient, effective, or even successful; but a "process" does exist. Watts Humphrey [Hum97] suggests that in order to change an ineffective personal process, an individual must move through four phases, each requiring training and careful instrumentation. The *Personal Software Process* (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed. The PSP model defines five framework activities:

**Planning.** This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

**High-level design.** External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

**High-level design review.** Formal verification methods (Chapter 21) are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

**Development.** The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.

**Postmortem.** Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

---

<sup>22</sup> It's worth noting the proponents of agile software development (Chapter 3) also argue that the process should remain close to the team. They propose an alternative method for achieving this.

## KEY POINT

PSP emphasizes the need to record and analyze the types of errors you make, so that you can develop strategies to eliminate them.

PSP stresses the need to identify errors early and, just as important, to understand the types of errors that you are likely to make. This is accomplished through a rigorous assessment activity performed on all work products you produce.

PSP represents a disciplined, metrics-based approach to software engineering that may lead to culture shock for many practitioners. However, when PSP is properly introduced to software engineers [Hum96], the resulting improvement in software engineering productivity and software quality are significant [Fer97]. However, PSP has not been widely adopted throughout the industry. The reasons, sadly, have more to do with human nature and organizational inertia than they do with the strengths and weaknesses of the PSP approach. PSP is intellectually challenging and demands a level of commitment (by practitioners and their managers) that is not always possible to obtain. Training is relatively lengthy, and training costs are high. The required level of measurement is culturally difficult for many software people.

Can PSP be used as an effective software process at a personal level? The answer is an unequivocal “yes.” But even if PSP is not adopted in its entirety, many of the personal process improvement concepts that it introduces are well worth learning.

### 2.6.2 Team Software Process (TSP)

#### WebRef

Information on building high-performance teams using TSP and PSP can be obtained at:  
[www.sei.cmu.edu/tsp/](http://www.sei.cmu.edu/tsp/).

Because many industry-grade software projects are addressed by a team of practitioners, Watts Humphrey extended the lessons learned from the introduction of PSP and proposed a *Team Software Process* (TSP). The goal of TSP is to build a “self-directed” project team that organizes itself to produce high-quality software. Humphrey [Hum98] defines the following objectives for TSP:

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPTs) of 3 to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM<sup>23</sup> Level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

## ADVICE

To form a self-directed team, you must collaborate well internally and communicate well externally.

A self-directed team has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitative project data (about productivity and quality); identifies a team process that is appropriate for the project and a strategy for implementing the process; defines local standards that are applicable to the team’s software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.

<sup>23</sup> The Capability Maturity Model (CMM), a measure of the effectiveness of a software process, is discussed in Chapter 30.

TSP defines the following framework activities: **project launch, high-level design, implementation, integration and test, and postmortem.** Like their counterparts in PSP (note that terminology is somewhat different), these activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product. The postmortem sets the stage for process improvements.

TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work. "Scripts" define specific process activities (i.e., project launch, design, implementation, integration and system testing, postmortem) and other more detailed work functions (e.g., development planning, requirements development, software configuration management, unit test) that are part of the team process.



TSP scripts define elements of the team process and activities that occur within the process.

TSP recognizes that the best software teams are self-directed.<sup>24</sup> Team members set project objectives, adapt the process to meet their needs, control the project schedule, and through measurement and analysis of the metrics collected, work continually to improve the team's approach to software engineering.

Like PSP, TSP is a rigorous approach to software engineering that provides distinct and quantifiable benefits in productivity and quality. The team must make a full commitment to the process and must undergo thorough training to ensure that the approach is properly applied.

## 2.7 PROCESS TECHNOLOGY

One or more of the process models discussed in the preceding sections must be adapted for use by a software team. To accomplish this, *process technology tools* have been developed to help software organizations analyze their current process, organize work tasks, control and monitor progress, and manage technical quality.

Process technology tools allow a software organization to build an automated model of the process framework, task sets, and umbrella activities discussed in Section 2.1. The model, normally represented as a network, can then be analyzed to determine typical workflow and examine alternative process structures that might lead to reduced development time or cost.

Once an acceptable process has been created, other process technology tools can be used to allocate, monitor, and even control all software engineering activities, actions, and tasks defined as part of the process model. Each member of a software team can use such tools to develop a checklist of work tasks to be performed, work products to be produced, and quality assurance activities to be conducted. The process technology tool can also be used to coordinate the use of other software engineering tools that are appropriate for a particular work task.

<sup>24</sup> In Chapter 3 I discuss the importance of "self-organizing" teams as a key element in agile software development.



### **Process Modeling Tools**

**Objective:** If an organization works to improve a business (or software) process, it must first understand it. Process modeling tools (also called *process technology* or *process management* tools) are used to represent the key elements of a process so that it can be better understood. Such tools can also provide links to process descriptions that help those involved in the process to understand the actions and work tasks that are required to perform it. Process modeling tools provide links to other tools that provide support to defined process activities.

**Mechanics:** Tools in this category allow a team to define the elements of a unique process model (actions, tasks, work products, QA points), provide detailed guidance on

### **SOFTWARE TOOLS**

the content or description of each process element, and then manage the process as it is conducted. In some cases, the process technology tools incorporate standard project management tasks such as estimating, scheduling, tracking, and control.

#### **Representative Tools:<sup>25</sup>**

*Igrafx Process Tools*—tools that enable a team to map, measure, and model the software process ([www.micrografx.com](http://www.micrografx.com))

*Adeptia BPM Server*—designed to manage, automate, and optimize business processes ([www.adeptia.com](http://www.adeptia.com))

*SpeedDev Suite*—a collection of six tools with a heavy emphasis on the management of communication and modeling activities ([www.speeddev.com](http://www.speeddev.com))

## **2.8 PRODUCT AND PROCESS**

If the process is weak, the end product will undoubtedly suffer. But an obsessive over-reliance on process is also dangerous. In a brief essay written many years ago, Margaret Davis [Dav95a] makes timeless comments on the duality of product and process:

About every ten years give or take five, the software community redefines “the problem” by shifting its focus from product issues to process issues. Thus, we have embraced structured programming languages (product) followed by structured analysis methods (process) followed by data encapsulation (product) followed by the current emphasis on the Software Engineering Institute’s Software Development Capability Maturity Model (process) [followed by object-oriented methods, followed by agile software development].

While the natural tendency of a pendulum is to come to rest at a point midway between two extremes, the software community’s focus constantly shifts because new force is applied when the last swing fails. These swings are harmful in and of themselves because they confuse the average software practitioner by radically changing what it means to perform the job let alone perform it well. The swings also do not solve “the problem” for they are doomed to fail as long as product and process are treated as forming a dichotomy instead of a duality.

There is precedence in the scientific community to advance notions of duality when contradictions in observations cannot be fully explained by one competing theory or another. The dual nature of light, which seems to be simultaneously particle and wave, has been accepted since the 1920s when Louis de Broglie proposed it. I believe that the

25 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

observations we can make on the artifacts of software and its development demonstrate a fundamental duality between product and process. You can never derive or understand the full artifact, its context, use, meaning, and worth if you view it as only a process or only a product . . .

All of human activity may be a process, but each of us derives a sense of self-worth from those activities that result in a representation or instance that can be used or appreciated either by more than one person, used over and over, or used in some other context not considered. That is, we derive feelings of satisfaction from reuse of our products by ourselves or others.

Thus, while the rapid assimilation of reuse goals into software development potentially increases the satisfaction software practitioners derive from their work, it also increases the urgency for acceptance of the duality of product and process. Thinking of a reusable artifact as only product or only process either obscures the context and ways to use it or obscures the fact that each use results in product that will, in turn, be used as input to some other software development activity. Taking one view over the other dramatically reduces the opportunities for reuse and, hence, loses the opportunity for increasing job satisfaction.

People derive as much (or more) satisfaction from the creative process as they do from the end product. An artist enjoys the brush strokes as much as the framed result. A writer enjoys the search for the proper metaphor as much as the finished book. As creative software professional, you should also derive as much satisfaction from the process as the end product. The duality of product and process is one important element in keeping creative people engaged as software engineering continues to evolve.

## 2.9 SUMMARY

A generic process model for software engineering encompasses a set of framework and umbrella activities, actions, and work tasks. Each of a variety of process models can be described by a different process flow—a description of how the framework activities, actions, and tasks are organized sequentially and chronologically. Process patterns can be used to solve common problems that are encountered as part of the software process.

Prescriptive process models have been applied for many years in an effort to bring order and structure to software development. Each of these models suggests a somewhat different process flow, but all perform the same set of generic framework activities: communication, planning, modeling, construction, and deployment.

Sequential process models, such as the waterfall and V models, are the oldest software engineering paradigms. They suggest a linear process flow that is often inconsistent with modern realities (e.g., continuous change, evolving systems, tight time lines) in the software world. They do, however, have applicability in situations where requirements are well defined and stable.

Incremental process models are iterative in nature and produce working versions of software quite rapidly. Evolutionary process models recognize the iterative, incremental nature of most software engineering projects and are designed to accommodate change. Evolutionary models, such as prototyping and the spiral model, produce incremental work products (or working versions of the software) quickly. These models can be adopted to apply across all software engineering activities—from concept development to long-term system maintenance.

The concurrent process model allows a software team to represent iterative and concurrent elements of any process model. Specialized models include the component-based model that emphasizes component reuse and assembly; the formal methods model that encourages a mathematically based approach to software development and verification; and the aspect-oriented model that accommodates crosscutting concerns spanning the entire system architecture. The Unified Process is a “use case driven, architecture-centric, iterative and incremental” software process designed as a framework for UML methods and tools.

Personal and team models for the software process have been proposed. Both emphasize measurement, planning, and self-direction as key ingredients for a successful software process.

## PROBLEMS AND POINTS TO PONDER

**2.1.** In the introduction to this chapter Baetjer notes: “The process provides interaction between users and designers, between users and evolving tools, and between designers and evolving tools [technology].” List five questions that (a) designers should ask users, (b) users should ask designers, (c) users should ask themselves about the software product that is to be built, (d) designers should ask themselves about the software product that is to be built and the process that will be used to build it.

**2.2.** Try to develop a set of actions for the communication activity. Select one action and define a task set for it.

**2.3.** A common problem during **communication** occurs when you encounter two stakeholders who have conflicting ideas about what the software should be. That is, you have mutually conflicting requirements. Develop a process pattern (this would be a stage pattern) using the template presented in Section 2.1.3 that addresses this problem and suggest an effective approach to it.

**2.4.** Do some research on PSP and present a brief presentation that describes the types of measurements that an individual software engineer is asked to make and how those measurement can be used to improve personal effectiveness.

**2.5.** The use of “scripts” (a required mechanism in TSP) is not universally praised within the software community. Make a list of pros and cons regarding scripts and suggest at least two situations in which they would be useful and another two situations where they might provide less benefit.

**2.6.** Read [Nog00] and write a two- or three-page paper that discusses the impact of “chaos” on software engineering.

**2.7.** Provide three examples of software projects that would be amenable to the waterfall model. Be specific.

- 2.8.** Provide three examples of software projects that would be amenable to the prototyping model. Be specific.
- 2.9.** What process adaptations are required if the prototype will evolve into a deliverable system or product?
- 2.10.** Provide three examples of software projects that would be amenable to the incremental model. Be specific.
- 2.11.** As you move outward along the spiral process flow, what can you say about the software that is being developed or maintained?
- 2.12.** Is it possible to combine process models? If so, provide an example.
- 2.13.** The concurrent process model defines a set of “states.” Describe what these states represent in your own words, and then indicate how they come into play within the concurrent process model.
- 2.14.** What are the advantages and disadvantages of developing software in which quality is “good enough”? That is, what happens when we emphasize development speed over product quality?
- 2.15.** Provide three examples of software projects that would be amenable to the component-based model. Be specific.
- 2.16.** It is possible to prove that a software component and even an entire program is correct. So why doesn’t everyone do this?
- 2.17.** Are the Unified Process and UML the same thing? Explain your answer.

## FURTHER READINGS AND INFORMATION SOURCES

Most software engineering textbooks consider traditional process models in some detail. Books by Sommerville (*Software Engineering*, 8th ed., Addison-Wesley, 2006), Pfleeger and Atlee (*Software Engineering*, 3d ed., Prentice-Hall, 2005), and Schach (*Object-Oriented and Classical Software Engineering*, 7th ed., McGraw-Hill, 2006) consider traditional paradigms and discuss their strengths and weaknesses. Glass (*Facts and Fallacies of Software Engineering*, Prentice-Hall, 2002) provides an unvarnished, pragmatic view of the software engineering process. Although not specifically dedicated to process, Brooks (*The Mythical Man-Month*, 2d ed., Addison-Wesley, 1995) presents age-old project wisdom that has everything to do with process.

Firesmith and Henderson-Sellers (*The OPEN Process Framework: An Introduction*, Addison-Wesley, 2001) present a general template for creating “flexible, yet discipline software processes” and discuss process attributes and objectives. Madachy (*Software Process Dynamics*, Wiley-IEEE, 2008) discusses modeling techniques that allow the interrelated technical and social elements of the software process to be analyzed. Sharpe and McDermott (*Workflow Modeling: Tools for Process Improvement and Application Development*, Artech House, 2001) present tools for modeling both software and business processes.

Lim (*Managing Software Reuse*, Prentice Hall, 2004) discusses reuse from a manager’s perspective. Ezran, Morisio, and Tully (*Practical Software Reuse*, Springer, 2002) and Jacobson, Griss, and Jonsson (*Software Reuse*, Addison-Wesley, 1997) present much useful information on component-based development. Heineman and Council (*Component-Based Software Engineering*, Addison-Wesley, 2001) describe the process required to implement component-based systems. Kenett and Baker (*Software Process Quality: Management and Control*, Marcel Dekker, 1999) consider how quality management and process design are intimately connected to one another.

Nygard (*Release It!: Design and Deploy Production-Ready Software*, Pragmatic Bookshelf, 2007) and Richardson and Gwaltney (*Ship it! A Practical Guide to Successful Software Projects*, Pragmatic Bookshelf, 2005) present a broad collection of useful guidelines that are applicable to the deployment activity.

In addition to Jacobson, Rumbaugh, and Booch's seminal book on the Unified Process [Jac99], books by Arlow and Neustadt (*UML 2 and the Unified Process*, Addison-Wesley, 2005), Kroll and Kruchten (*The Rational Unified Process Made Easy*, Addison-Wesley, 2003), and Farve (*UML and the Unified Process*, IRM Press, 2003) provide excellent complementary information. Gibbs (*Project Management with the IBM Rational Unified Process*, IBM Press, 2006) discusses project management within the context of the UP.

A wide variety of information sources on software engineering and the software process are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process can be found at the SEPA website: [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm).

# AGILE DEVELOPMENT

## KEY CONCEPTS

<b>Adaptive Software Development</b>	...81
<b>agile process</b>	...68
<b>Agile Unified Process</b>	.....89
<b>agility</b>	.....67
<b>Crystal</b>	.....85
<b>DSDM</b>	.....84
<b>Extreme Programming</b>	...72

## QUICK LOOK

**What is it?** Agile software engineering combines a philosophy and a set of development guidelines. The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity. The development guidelines stress delivery over analysis and design (although these activities are not discouraged), and active and continuous communication between developers and customers.

**Who does it?** Software engineers and other project stakeholders (managers, customers, end users) work together on an agile team—a team that is self-organizing and in control of its own destiny. An agile team fosters communication and collaboration among all who serve on it.

**Why is it important?** The modern business environment that spawns computer-based systems and software products is fast-paced and ever-changing. Agile software engineering represents a reasonable alternative to conventional

In 2001, Kent Beck and 16 other noted software developers, writers, and consultants [Bec01a] (referred to as the “Agile Alliance”) signed the “Manifesto for Agile Software Development.” It stated:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions* over processes and tools
- Working software* over comprehensive documentation
- Customer collaboration* over contract negotiation
- Responding to change* over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

software engineering for certain classes of software and certain types of software projects. It has been demonstrated to deliver successful systems quickly.

**What are the steps?** Agile development might best be termed “software engineering lite.” The basic framework activities—communication, planning, modeling, construction, and deployment—remain. But they morph into a minimal task set that pushes the project team toward construction and delivery (some would argue that this is done at the expense of problem analysis and solution design).

**What is the work product?** Both the customer and the software engineer have the same view—the only really important work product is an operational “software increment” that is delivered to the customer on the appropriate commitment date.

**How do I ensure that I’ve done it right?** If the agile team agrees that the process works, and the team produces deliverable software increments that satisfy the customer, you’ve done it right.

FDD .....	.86
Industrial XP .....	.77
Lean Software Development .....	.87
pair programming .....	.76
project velocity .....	.74
refactoring .....	.75
Scrum .....	.82
stories .....	.74
XP process .....	.73

A manifesto is normally associated with an emerging political movement—one that attacks the old guard and suggests revolutionary change (hopefully for the better). In some ways, that's exactly what agile development is all about.

Although the underlying ideas that guide agile development have been with us for many years, it has been less than two decades since these ideas have crystallized into a “movement.” In essence, agile<sup>1</sup> methods were developed in an effort to overcome perceived and actual weaknesses in conventional software engineering. Agile development can provide important benefits, but it is not applicable to all projects, all products, all people, and all situations. It is also *not* antithetical to solid software engineering practice and can be applied as an overriding philosophy for all software work.

In the modern economy, it is often difficult or impossible to predict how a computer-based system (e.g., a Web-based application) will evolve as time passes. Market conditions change rapidly, end-user needs evolve, and new competitive threats emerge without warning. In many situations, you won't be able to define requirements fully before the project begins. You must be agile enough to respond to a fluid business environment.

Fluidity implies change, and change is expensive. Particularly if it is uncontrolled or poorly managed. One of the most compelling characteristics of the agile approach is its ability to reduce the costs of change throughout the software process.

Does this mean that a recognition of challenges posed by modern realities causes you to discard valuable software engineering principles, concepts, methods, and tools? Absolutely not! Like all engineering disciplines, software engineering continues to evolve. It can be adapted easily to meet the challenges posed by a demand for agility.

In a thought-provoking book on agile software development, Alistair Cockburn [Coc02] argues that the prescriptive process models introduced in Chapter 2 have a major failing: *they forget the frailties of the people who build computer software*. Software engineers are not robots. They exhibit great variation in working styles; significant differences in skill level, creativity, orderliness, consistency, and spontaneity. Some communicate well in written form, others do not. Cockburn argues that process models can “deal with people’s common weaknesses with [either] discipline or tolerance” and that most prescriptive process models choose discipline. He states: “Because consistency in action is a human weakness, high discipline methodologies are fragile.”

If process models are to work, they must provide a realistic mechanism for encouraging the discipline that is necessary, or they must be characterized in a manner that shows “tolerance” for the people who do software engineering work. Invariably, tolerant practices are easier for software people to adopt and sustain, but (as Cockburn admits) they may be less productive. Like most things in life, trade-offs must be considered.

---

<sup>1</sup> Agile methods are sometimes referred to as *light methods* or *lean methods*.

 **vote:**

“Agility: 1,  
everything else: 0.”

**Tom DeMarco**

### 3.1 WHAT IS AGILITY?

Just what is agility in the context of software engineering work? Ivar Jacobson [Jac02a] provides a useful discussion:

*Agility has become today's buzzword when describing a modern software process. Everyone is agile. An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.*

In Jacobson's view, the pervasiveness of change is the primary driver for agility. Software engineers must be quick on their feet if they are to accommodate the rapid changes that Jacobson describes.



*Don't make the mistake of assuming that agility gives you license to hack out solutions. A process is required and discipline is essential.*

But agility is more than an effective response to change. It also encompasses the philosophy espoused in the manifesto noted at the beginning of this chapter. It encourages team structures and attitudes that make communication (among team members, between technologists and business people, between software engineers and their managers) more facile. It emphasizes rapid delivery of operational software and de-emphasizes the importance of intermediate work products (not always a good thing); it adopts the customer as a part of the development team and works to eliminate the "us and them" attitude that continues to pervade many software projects; it recognizes that planning in an uncertain world has its limits and that a project plan must be flexible.

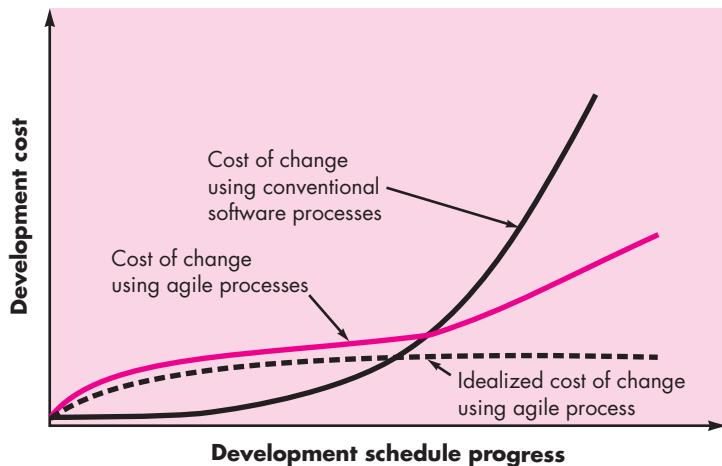
Agility can be applied to any software process. However, to accomplish this, it is essential that the process be designed in a way that allows the project team to adapt tasks and to streamline them, conduct planning in a way that understands the fluidity of an agile development approach, eliminate all but the most essential work products and keep them lean, and emphasize an incremental delivery strategy that gets working software to the customer as rapidly as feasible for the product type and operational environment.

### 3.2 AGILITY AND THE COST OF CHANGE

The conventional wisdom in software development (supported by decades of experience) is that the cost of change increases nonlinearly as a project progresses (Figure 3.1, solid black curve). It is relatively easy to accommodate a change when a software team is gathering requirements (early in a project). A usage scenario might have to be modified, a list of functions may be extended, or a written specification can be edited. The costs of doing this work are minimal, and the time required will

**FIGURE 3.1**

**Change costs as a function of time in development**

**QUOTE:**

"Agility is dynamic, content specific, aggressively change embracing, and growth oriented."

—Steven Goldman et al.

**KEY POINT**

An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment.

not adversely affect the outcome of the project. But what if we fast-forward a number of months? The team is in the middle of validation testing (something that occurs relatively late in the project), and an important stakeholder is requesting a major functional change. The change requires a modification to the architectural design of the software, the design and construction of three new components, modifications to another five components, the design of new tests, and so on. Costs escalate quickly, and the time and cost required to ensure that the change is made without unintended side effects is nontrivial.

Proponents of agility (e.g., [Bec00], [Amb04]) argue that a well-designed agile process “flattens” the cost of change curve (Figure 3.1, shaded, solid curve), allowing a software team to accommodate changes late in a software project without dramatic cost and time impact. You’ve already learned that the agile process encompasses incremental delivery. When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming (discussed later in this chapter), the cost of making a change is attenuated. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence [Coc01a] to suggest that a significant reduction in the cost of change can be achieved.

### 3.3 WHAT IS AN AGILE PROCESS?

Any agile software process is characterized in a manner that addresses a number of key assumptions [Fow02] about the majority of software projects:

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.

**WebRef**

A comprehensive collection of articles on the agile process can be found at  
[www.aanpo.org/articles/index](http://www.aanpo.org/articles/index).

2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

Given these three assumptions, an important question arises: How do we create a process that can manage *unpredictability*? The answer, as I have already noted, lies in process adaptability (to rapidly changing project and technical conditions). An agile process, therefore, must be *adaptable*.

But continual adaptation without forward progress accomplishes little. Therefore, an agile software process must adapt *incrementally*. To accomplish incremental adaptation, an agile team requires customer feedback (so that the appropriate adaptations can be made). An effective catalyst for customer feedback is an operational prototype or a portion of an operational system. Hence, an *incremental development strategy* should be instituted. *Software increments* (executable prototypes or portions of an operational system) must be delivered in short time periods so that adaptation keeps pace with change (unpredictability). This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.



Although agile processes embrace change, it is still important to examine the reasons for change.

### 3.3.1 Agility Principles

The Agile Alliance (see [Agi03], [Fow01]) defines 12 agility principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.



Working software is important, but don't forget that it must also exhibit a variety of quality attributes including reliability, usability, and maintainability.

9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Not every agile process model applies these 12 principles with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more of the principles. However, the principles define an *agile spirit* that is maintained in each of the process models presented in this chapter.

### 3.3.2 The Politics of Agile Development

There is considerable debate (sometimes strident) about the benefits and applicability of agile software development as opposed to more conventional software engineering processes. Jim Highsmith [Hig02a] (facetiously) states the extremes when he characterizes the feeling of the pro-agility camp (“agilists”). “Traditional methodologists are a bunch of stick-in-the-muds who’d rather produce flawless documentation than a working system that meets business needs.” As a counterpoint, he states (again, facetiously) the position of the traditional software engineering camp: “Light-weight, er, ‘agile’ methodologists are a bunch of glorified hackers who are going to be in for a heck of a surprise when they try to scale up their toys into enterprise-wide software.”

Like all software technology arguments, this methodology debate risks degenerating into a religious war. If warfare breaks out, rational thought disappears and beliefs rather than facts guide decision making.

No one is against agility. The real question is: What is the best way to achieve it? As important, how do you build software that meets customers’ needs today and exhibits the quality characteristics that will enable it to be extended and scaled to meet customers’ needs over the long term?

There are no absolute answers to either of these questions. Even within the agile school itself, there are many proposed process models (Section 3.4), each with a subtly different approach to the agility problem. Within each model there is a set of “ideas” (agilists are loath to call them “work tasks”) that represent a significant departure from traditional software engineering. And yet, many agile concepts are simply adaptations of good software engineering concepts. Bottom line: there is much that can be gained by considering the best of both schools and virtually nothing to be gained by denigrating either approach.

If you have further interest, see [Hig01], [Hig02a], and [DeM02] for an entertaining summary of other important technical and political issues.



*You don't have to choose between agility and software engineering. Rather, define a software engineering approach that is agile.*

**quote:**

"Agile methods derive much of their agility by relying on the tacit knowledge embodied in the team, rather than writing the knowledge down in plans."

**Barry Boehm**



**What key traits must exist among the people on an effective software team?**

**quote:**

"What counts as barely sufficient for one team is either overly sufficient or insufficient for another."

**Alistair Cockburn**

### 3.3.3 Human Factors

Proponents of agile software development take great pains to emphasize the importance of "people factors." As Cockburn and Highsmith [Coc01a] state, "Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that *the process molds to the needs of the people and team*, not the other way around.<sup>2</sup>

If members of the software team are to drive the characteristics of the process that is applied to build software, a number of key traits must exist among the people on an agile team and the team itself:

**Competence.** In an agile development (as well as software engineering) context, "competence" encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.

**Common focus.** Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.

**Collaboration.** Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.

**Decision-making ability.** Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.

**Fuzzy problem-solving ability.** Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change. In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs to be solved tomorrow. However, lessons learned from any problem-solving

---

<sup>2</sup> Successful software engineering organizations recognize this reality regardless of the process model they choose.

activity (including those that solve the wrong problem) may be of benefit to the team later in the project.

**Mutual trust and respect.** The agile team must become what DeMarco and Lister [DeM98] call a “jelled” team (Chapter 24). A jelled team exhibits the trust and respect that are necessary to make them “so strongly knit that the whole is greater than the sum of the parts.” [DeM98]

## KEY POINT

A self-organizing team is in control of the work it performs. The team makes its own commitments and defines plans to achieve them.

**Self-organization.** In the context of agile development, self-organization implies three things: (1) the agile team organizes itself for the work to be done, (2) the team organizes the process to best accommodate its local environment, (3) the team organizes the work schedule to best achieve delivery of the software increment. Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale. In essence, the team serves as its own management. Ken Schwaber [Sch02] addresses these issues when he writes: “The team selects how much work it believes it can perform within the iteration, and the team commits to the work. Nothing demotivates a team as much as someone else making commitments for it. Nothing motivates a team as much as accepting the responsibility for fulfilling commitments that it made itself.”

## 3.4 EXTREME PROGRAMMING (XP)

In order to illustrate an agile process in a bit more detail, I'll provide you with an overview of *Extreme Programming* (XP), the most widely used approach to agile software development. Although early work on the ideas and methods associated with XP occurred during the late 1980s, the seminal work on the subject has been written by Kent Beck [Bec04a]. More recently, a variant of XP, called *Industrial XP* (IXP) has been proposed [Ker05]. IXP refines XP and targets the agile process specifically for use within large organizations.

### 3.4.1 XP Values

Beck [Bec04a] defines a set of five *values* that establish a foundation for all work performed as part of XP—communication, simplicity, feedback, courage, and respect. Each of these values is used as a driver for specific XP activities, actions, and tasks.

In order to achieve effective *communication* between software engineers and other stakeholders (e.g., to establish required features and functions for the software), XP emphasizes close, yet informal (verbal) collaboration between customers and developers, the establishment of effective metaphors<sup>3</sup> for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.

---

<sup>3</sup> In the XP context, a *metaphor* is “a story that everyone—customers, programmers, and managers—can tell about how the system works” [Bec04a].



*Keep it simple whenever you can, but recognize that continual “refactoring” can absorb significant time and resources.*

### vote:

*“XP is the answer to the question, ‘How little can we do and still build great software?’”*

**Anonymous**

### WebRef

An excellent overview of “rules” for XP can be found at [www.extremeprogramming.org/rules.html](http://www.extremeprogramming.org/rules.html).

To achieve *simplicity*, XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code). If the design must be improved, it can be *refactored*<sup>4</sup> at a later time.

*Feedback* is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy (Chapters 17 through 20), the software (via test results) provides the agile team with feedback. XP makes use of the *unit test* as its primary testing tactic. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality. As an increment is delivered to a customer, the *user stories* or *use cases* (Chapter 5) that are implemented by the increment are used as a basis for acceptance tests. The degree to which the software implements the output, function, and behavior of the use case is a form of feedback. Finally, as new requirements are derived as part of iterative planning, the team provides the customer with rapid feedback regarding cost and schedule impact.

Beck [Bec04a] argues that strict adherence to certain XP practices demands *courage*. A better word might be *discipline*. For example, there is often significant pressure to design for future requirements. Most software teams succumb, arguing that “designing for tomorrow” will save time and effort in the long run. An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.

By following each of these values, the agile team inculcates *respect* among its members, between other stakeholders and team members, and indirectly, for the software itself. As they achieve successful delivery of software increments, the team develops growing respect for the XP process.

### 3.4.2 The XP Process

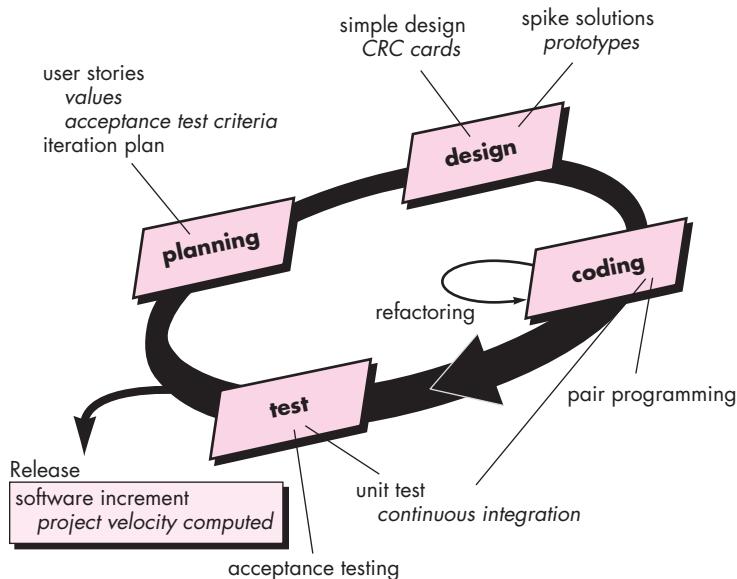
Extreme Programming uses an object-oriented approach (Appendix 2) as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure 3.2 illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity. Key XP activities are summarized in the paragraphs that follow.

**Planning.** The planning activity (also called *the planning game*) begins with *listening*—a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad

<sup>4</sup> Refactoring allows a software engineer to improve the internal structure of a design (or source code) without changing its external functionality or behavior. In essence, refactoring can be used to improve the efficiency, readability, or performance of a design or the code that implements a design.

**FIGURE 3.2**

The Extreme Programming process



**What is an XP “story”?**

feel for required output and major features and functionality. Listening leads to the creation of a set of “stories” (also called *user stories*) that describe required output, features, and functionality for software to be built. Each *story* (similar to use cases described in Chapter 5) is written by the customer and is placed on an index card. The customer assigns a *value* (i.e., a priority) to the story based on the overall business value of the feature or function.<sup>5</sup> Members of the XP team then assess each story and assign a *cost*—measured in development weeks—to it. If the story is estimated to require more than three development weeks, the customer is asked to split the story into smaller stories and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time.

**WebRef**

A worthwhile XP “planning game” can be found at:  
[c2.com/cgi/wiki?planningGame](http://c2.com/cgi/wiki?planningGame).

Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team. Once a basic *commitment* (agreement on stories to be included, delivery date, and other project matters) is made for a release, the XP team orders the stories that will be developed in one of three ways: (1) all stories will be implemented immediately (within a few weeks), (2) the stories with highest value will be moved up in the schedule and implemented first, or (3) the riskiest stories will be moved up in the schedule and implemented first.

After the first project release (also called a software increment) has been delivered, the XP team computes project velocity. Stated simply, *project velocity* is the

<sup>5</sup> The value of a story may also be dependent on the presence of another story.



**Project velocity is a subtle measure of team productivity.**



*XP deemphasizes the importance of design. Not everyone agrees. In fact, there are times when design should be emphasized.*

#### WebRef

Refactoring techniques and tools can be found at:

[www.refactoring.com](http://www.refactoring.com).

number of customer stories implemented during the first release. Project velocity can then be used to (1) help estimate delivery dates and schedule for subsequent releases and (2) determine whether an overcommitment has been made for all stories across the entire development project. If an overcommitment occurs, the content of releases is modified or end delivery dates are changed.

As development work proceeds, the customer can add stories, change the value of an existing story, split stories, or eliminate them. The XP team then reconsiders all remaining releases and modifies its plans accordingly.

**Design.** XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality (because the developer assumes it will be required later) is discouraged.<sup>6</sup>

XP encourages the use of CRC cards (Chapter 7) as an effective mechanism for thinking about the software in an object-oriented context. CRC (class-responsibility-collaborator) cards identify and organize the object-oriented classes<sup>7</sup> that are relevant to the current software increment. The XP team conducts the design exercise using a process similar to the one described in Chapter 8. The CRC cards are the only design work product produced as part of the XP process.

If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a *spike solution*, the design prototype is implemented and evaluated. The intent is to lower risk when true implementation starts and to validate the original estimates for the story containing the design problem.

In the preceding section, we noted that XP encourages *refactoring*—a construction technique that is also a method for design optimization. Fowler [Fow00] describes refactoring in the following manner:

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure. It is a disciplined way to clean up code [and modify/simplify the internal design] that minimizes the chances of introducing bugs. In essence, when you refactor you are improving the design of the code after it has been written.

Because XP design uses virtually no notation and produces few, if any, work products other than CRC cards and spike solutions, design is viewed as a transient artifact that can and should be continually modified as construction proceeds. The intent of refactoring is to control these modifications by suggesting small design changes

---

<sup>6</sup> These design guidelines should be followed in every software engineering method, although there are times when sophisticated design notation and terminology may get in the way of simplicity.

<sup>7</sup> Object-oriented classes are discussed in Appendix 2, in Chapter 8, and throughout Part 2 of this book.

## KEY POINT

Refactoring improves the internal structure of a design (or source code) without changing its external functionality or behavior.

### WebRef

Useful information on XP can be obtained at [www.xprogramming.com](http://www.xprogramming.com).

### What is pair programming?

## ADVICE

Many software teams are populated by individualists. You'll have to work to change that culture if pair programming is to work effectively.

### How are unit tests used in XP?

that "can radically improve the design" [Fow00]. It should be noted, however, that the effort required for refactoring can grow dramatically as the size of an application grows.

A central notion in XP is that design occurs both before *and after* coding commences. Refactoring means that design occurs continuously as the system is constructed. In fact, the construction activity itself will provide the XP team with guidance on how to improve the design.

**Coding.** After stories are developed and preliminary design work is done, the team does *not* move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment).<sup>8</sup> Once the unit test<sup>9</sup> has been created, the developer is better able to focus on what must be implemented to pass the test. Nothing extraneous is added (KIS). Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

A key concept during the coding activity (and one of the most talked about aspects of XP) is *pair programming*. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real-time problem solving (two heads are often better than one) and real-time quality assurance (the code is reviewed as it is created). It also keeps the developers focused on the problem at hand. In practice, each person takes on a slightly different role. For example, one person might think about the coding details of a particular portion of the design while the other ensures that coding standards (a required part of XP) are being followed or that the code for the story will satisfy the unit test that has been developed to validate the code against the story.

As pair programmers complete their work, the code they develop is integrated with the work of others. In some cases this is performed on a daily basis by an integration team. In other cases, the pair programmers have integration responsibility. This "continuous integration" strategy helps to avoid compatibility and interfacing problems and provides a "smoke testing" environment (Chapter 17) that helps to uncover errors early.

**Testing.** I have already noted that the creation of unit tests before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and repeatedly). This encourages a regression testing strategy (Chapter 17) whenever code is modified (which is often, given the XP refactoring philosophy).

<sup>8</sup> This approach is analogous to knowing the exam questions before you begin to study. It makes studying much easier by focusing attention only on the questions that will be asked.

<sup>9</sup> Unit testing, discussed in detail in Chapter 17, focuses on an individual software component, exercising the component's interface, data structures, and functionality in an effort to uncover errors that are local to the component.

As the individual unit tests are organized into a “universal testing suite” [Wel99], integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells [Wel99] states: “Fixing small problems every few hours takes less time than fixing huge problems just before the deadline.”



XP acceptance tests are derived from user stories.

XP *acceptance tests*, also called *customer tests*, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

### 3.4.3 Industrial XP

Joshua Kerievsky [Ker05] describes *Industrial Extreme Programming* (IXP) in the following manner: “IXP is an organic evolution of XP. It is imbued with XP’s minimalist, customer-centric, test-driven spirit. IXP differs most from the original XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices.” IXP incorporates six new practices that are designed to help ensure that an XP project works successfully for significant projects within a large organization.

**What new practices are appended to XP to create IXP?**

**vote:**  
“Ability is what you’re capable of doing. Motivation determines what you do. Attitude determines how well you do it.”

Lou Holtz

**Readiness assessment.** Prior to the initiation of an IXP project, the organization should conduct a *readiness assessment*. The assessment ascertains whether (1) an appropriate development environment exists to support IXP, (2) the team will be populated by the proper set of stakeholders, (3) the organization has a distinct quality program and supports continuous improvement, (4) the organizational culture will support the new values of an agile team, and (5) the broader project community will be populated appropriately.

**Project community.** Classic XP suggests that the right people be used to populate the agile team to ensure success. The implication is that people on the team must be well-trained, adaptable and skilled, and have the proper temperament to contribute to a self-organizing team. When XP is to be applied for a significant project in a large organization, the concept of the “team” should morph into that of a *community*. A community may have a technologist and customers who are central to the success of a project as well as many other stakeholders (e.g., legal staff, quality auditors, manufacturing or sales types) who “are often at the periphery of an IXP project yet they may play important roles on the project” [Ker05]. In IXP, the community members and their roles should be explicitly defined and mechanisms for communication and coordination between community members should be established.

**Project chartering.** The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the

organization. Chartering also examines the context of the project to determine how it complements, extends, or replaces existing systems or processes.

**Test-driven management.** An IXP project requires measurable criteria for assessing the state of the project and the progress that has been made to date. Test-driven management establishes a series of measurable “destinations” [Ker05] and then defines mechanisms for determining whether or not these destinations have been reached.

**Retrospectives.** An IXP team conducts a specialized technical review (Chapter 15) after a software increment is delivered. Called a *retrospective*, the review examines “issues, events, and lessons-learned” [Ker05] across a software increment and/or the entire software release. The intent is to improve the IXP process.

**Continuous learning.** Because learning is a vital part of continuous process improvement, members of the XP team are encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher-quality product.

In addition to the six new practices discussed, IXP modifies a number of existing XP practices. *Story-driven development* (SDD) insists that stories for acceptance tests be written before a single line of code is generated. *Domain-driven design* (DDD) is an improvement on the “system metaphor” concept used in XP. DDD [Eva03] suggests the evolutionary creation of a domain model that “accurately represents how domain experts think about their subject” [Ker05]. *Pairing* extends the XP pair-programming concept to include managers and other stakeholders. The intent is to improve knowledge sharing among XP team members who may not be directly involved in technical development. *Iterative usability* discourages front-loaded interface design in favor of usability design that evolves as software increments are delivered and users’ interaction with the software is studied.

IXP makes smaller modifications to other XP practices and redefines certain roles and responsibilities to make them more amenable to significant projects for large organizations. For further discussion of IXP, visit <http://industrialxp.org>.

### 3.4.4 The XP Debate

All new process models and methods spur worthwhile discussion and in some instances heated debate. Extreme Programming has done both. In an interesting book that examines the efficacy of XP, Stephens and Rosenberg [Ste03] argue that many XP practices are worthwhile, but others have been overhyped, and a few are problematic. The authors suggest that the codependent nature of XP practices are both its strength and its weakness. Because many organizations adopt only a subset of XP practices, they weaken the efficacy of the entire process. Proponents counter that XP is continuously evolving and that many of the issues raised by critics have been

addressed as XP practice matures. Among the issues that continue to trouble some critics of XP are:<sup>10</sup>



- *Requirements volatility.* Because the customer is an active member of the XP team, changes to requirements are requested informally. As a consequence, the scope of the project can change and earlier work may have to be modified to accommodate current needs. Proponents argue that this happens regardless of the process that is applied and that XP provides mechanisms for controlling scope creep.
- *Conflicting customer needs.* Many projects have multiple customers, each with his own set of needs. In XP, the team itself is tasked with assimilating the needs of different customers, a job that may be beyond their scope of authority.
- *Requirements are expressed informally.* User stories and acceptance tests are the only explicit manifestation of requirements in XP. Critics argue that a more formal model or specification is often needed to ensure that omissions, inconsistencies, and errors are uncovered before the system is built. Proponents counter that the changing nature of requirements makes such models and specification obsolete almost as soon as they are developed.
- *Lack of formal design.* XP deemphasizes the need for architectural design and in many instances, suggests that design of all kinds should be relatively informal. Critics argue that when complex systems are built, design must be emphasized to ensure that the overall structure of the software will exhibit quality and maintainability. XP proponents suggest that the incremental nature of the XP process limits complexity (simplicity is a core value) and therefore reduces the need for extensive design.

You should note that every software process has flaws and that many software organizations have used XP successfully. The key is to recognize where a process may have weaknesses and to adapt it to the specific needs of your organization.

## SAFEHOME



### Considering Agile Software Development

**The scene:** Doug Miller's office.

**The Players:** Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member.

**The conversation:**

(A knock on the door, Jamie and Vinod enter Doug's office)

**Jamie:** Doug, you got a minute?

<sup>10</sup> For a detailed look at some thoughtful criticism that has been leveled at XP, visit [www.softwarereality.com/ExtremeProgramming.jsp](http://www.softwarereality.com/ExtremeProgramming.jsp).

**Doug:** Sure Jamie, what's up?

**Jamie:** We've been thinking about our process discussion yesterday . . . you know, what process we're going to choose for this new *SafeHome* project.

**Doug:** And?

**Vinod:** I was talking to a friend at another company, and he was telling me about Extreme Programming. It's an agile process model . . . heard of it?

**Doug:** Yeah, some good, some bad.

**Jamie:** Well, it sounds pretty good to us. Lets you develop software really fast, uses something called pair programming to do real-time quality checks . . . it's pretty cool, I think.

**Doug:** It does have a lot of really good ideas. I like the pair-programming concept, for instance, and the idea that stakeholders should be part of the team.

**Jamie:** Huh? You mean that marketing will work on the project team with us?

**Doug (nodding):** They're a stakeholder, aren't they?

**Jamie:** Jeez . . . they'll be requesting changes every five minutes.

**Vinod:** Not necessarily. My friend said that there are ways to "embrace" changes during an XP project.

**Doug:** So you guys think we should use XP?

**Jamie:** It's definitely worth considering.

**Doug:** I agree. And even if we choose an incremental model as our approach, there's no reason why we can't incorporate much of what XP has to offer.

**Vinod:** Doug, before you said "some good, some bad." What was the "bad"?

**Doug:** The thing I don't like is the way XP downplays analysis and design . . . sort of says that writing code is where the action is . . .

(The team members look at one another and smile.)

**Doug:** So you agree with the XP approach?

**Jamie (speaking for both):** Writing code is what we do, Boss!

**Doug (laughing):** True, but I'd like to see you spend a little less time coding and then recoding and a little more time analyzing what has to be done and designing a solution that works.

**Vinod:** Maybe we can have it both ways, agility with a little discipline.

**Doug:** I think we can, Vinod. In fact, I'm sure of it.

### 3.5 OTHER AGILE PROCESS MODELS

#### Quote:

"Our profession goes through methodologies like a 14-year-old goes through clothing."

Stephen Hawrysh and Jim Ruprecht

The history of software engineering is littered with dozens of obsolete process descriptions and methodologies, modeling methods and notations, tools, and technology. Each flared in notoriety and was then eclipsed by something new and (purportedly) better. With the introduction of a wide array of agile process models—each contending for acceptance within the software development community—the agile movement is following the same historical path.<sup>11</sup>

As I noted in the last section, the most widely used of all agile process models is Extreme Programming (XP). But many other agile process models have been proposed and are in use across the industry. Among the most common are:

- Adaptive Software Development (ASD)
- Scrum
- Dynamic Systems Development Method (DSDM)

<sup>11</sup> This is not a bad thing. Before one or more models or methods are accepted as a de facto standard, all must contend for the hearts and minds of software engineers. The "winners" evolve into best practice, while the "losers" either disappear or merge with the winning models.

- Crystal
- Feature Drive Development (FDD)
- Lean Software Development (LSD)
- Agile Modeling (AM)
- Agile Unified Process (AUP)

In the sections that follow, I present a very brief overview of each of these agile process models. It is important to note that *all* agile process models conform (to a greater or lesser degree) to the *Manifesto for Agile Software Development* and the principles noted in Section 3.3.1. For additional detail, refer to the references noted in each subsection or for a survey, examine the “agile software development” entry in Wikipedia.<sup>12</sup>

### 3.5.1 Adaptive Software Development (ASD)

#### WebRef

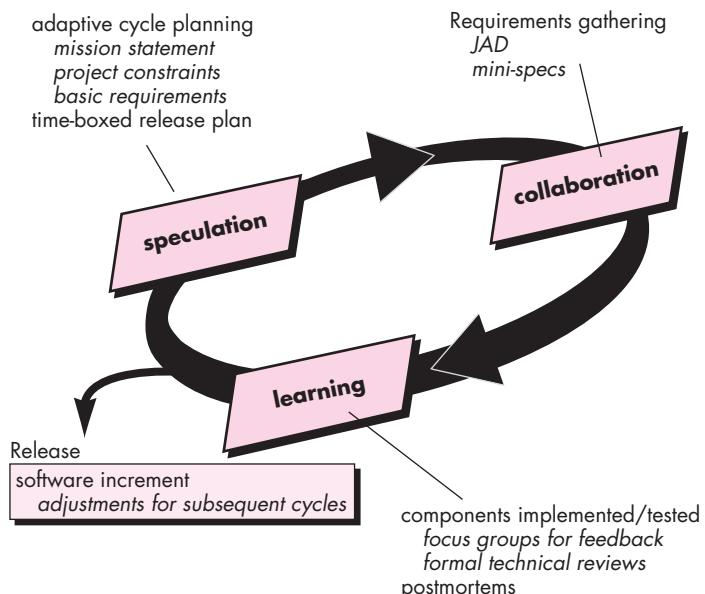
Useful resources for ASD can be found at [www.adaptivesd.com](http://www.adaptivesd.com).

*Adaptive Software Development* (ASD) has been proposed by Jim Highsmith [Hig00] as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization.

Highsmith argues that an agile, adaptive development approach based on collaboration is “as much a source of *order* in our complex interactions as discipline and engineering.” He defines an ASD “life cycle” (Figure 3.3) that incorporates three phases, speculation, collaboration, and learning.

**FIGURE 3.3**

Adaptive software development



12 See [http://en.wikipedia.org/wiki/Agile\\_software\\_development#Agile\\_methods](http://en.wikipedia.org/wiki/Agile_software_development#Agile_methods).

During *speculation*, the project is initiated and *adaptive cycle planning* is conducted. Adaptive cycle planning uses project initiation information—the customer's mission statement, project constraints (e.g., delivery dates or user descriptions), and basic requirements—to define the set of release cycles (software increments) that will be required for the project.



*Effective collaboration with your customer will only occur if you jettison any "us and them" attitudes.*

No matter how complete and farsighted the cycle plan, it will invariably change. Based on information obtained at the completion of the first cycle, the plan is reviewed and adjusted so that planned work better fits the reality in which an ASD team is working.

Motivated people use *collaboration* in a way that multiplies their talent and creative output beyond their absolute numbers. This approach is a recurring theme in all agile methods. But collaboration is not easy. It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking. It is, above all, a matter of trust. People working together must trust one another to (1) criticize without animosity, (2) assist without resentment, (3) work as hard as or harder than they do, (4) have the skill set to contribute to the work at hand, and (5) communicate problems or concerns in a way that leads to effective action.



ASD emphasizes learning as a key element in achieving a "self-organizing" team.

As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on "learning" as much as it is on progress toward a completed cycle. In fact, Highsmith [Hig00] argues that software developers often overestimate their own understanding (of the technology, the process, and the project) and that learning will help them to improve their level of real understanding. ASD teams learn in three ways: focus groups (Chapter 5), technical reviews (Chapter 14), and project postmortems.

The ASD philosophy has merit regardless of the process model that is used. ASD's overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning yield software project teams that have a much higher likelihood of success.

### 3.5.2 Scrum

Scrum (the name is derived from an activity that occurs during a rugby match<sup>13</sup>) is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. In recent years, further development on the Scrum methods has been performed by Schwaber and Beedle [Sch01a].

Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each

#### WebRef

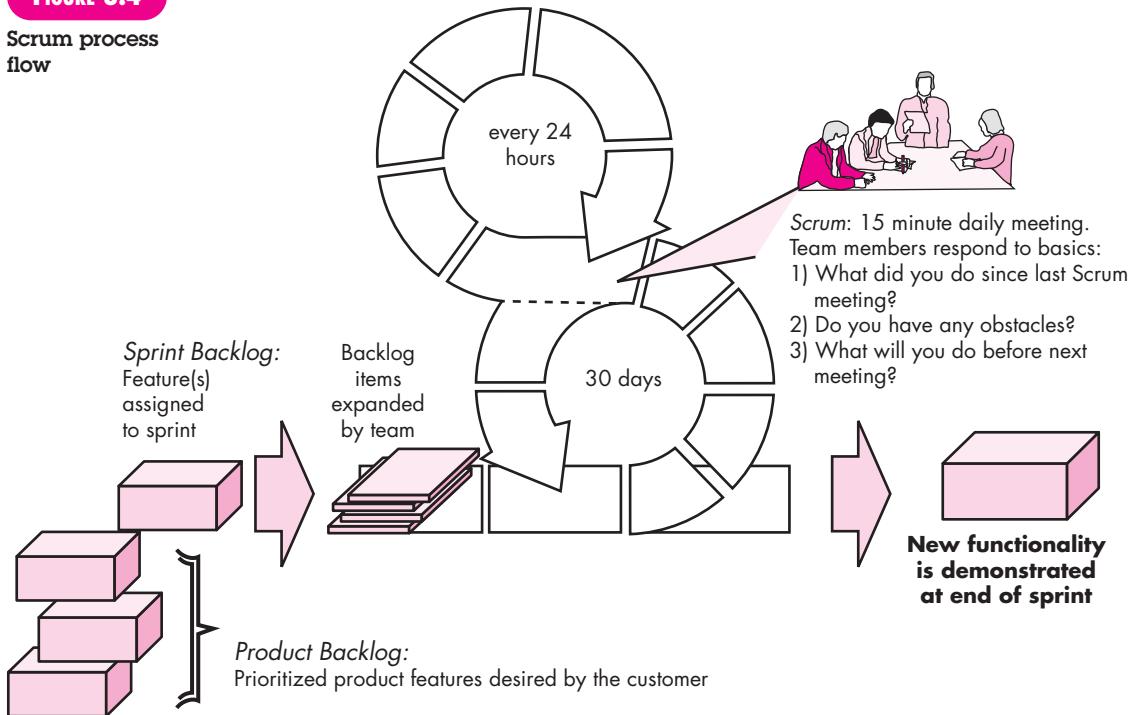
Useful Scrum information and resources can be found at [www.controlchaos.com](http://www.controlchaos.com).

---

<sup>13</sup> A group of players forms around the ball and the teammates work together (sometimes violently!) to move the ball downfield.

**FIGURE 3.4**

Scrum process flow



framework activity, work tasks occur within a process pattern (discussed in the following paragraph) called a *sprint*. The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team. The overall flow of the Scrum process is illustrated in Figure 3.4.

### KEY POINT

Scrum incorporates a set of process patterns that emphasize project priorities, compartmentalized work units, communication, and frequent customer feedback.

Scrum emphasizes the use of a set of software process patterns [Noy02] that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development actions:

*Backlog*—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time (this is how changes are introduced). The product manager assesses the backlog and updates priorities as required.

*Sprints*—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box<sup>14</sup> (typically 30 days).

<sup>14</sup> A *time-box* is a project management term (see Part 4 of this book) that indicates a period of time that has been allocated to accomplish some task.

Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.

*Scrum meetings*—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members [Noy02]:

- What did you do since the last team meeting?
- What obstacles are you encountering?
- What do you plan to accomplish by the next team meeting?

A team leader, called a *Scrum master*, leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to “knowledge socialization” [Bee99] and thereby promote a self-organizing team structure.

*Demos*—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

Beedle and his colleagues [Bee99] present a comprehensive discussion of these patterns in which they state: “Scrum assumes up-front the existence of chaos. . . .” The Scrum process patterns enable a software team to work successfully in a world where the elimination of uncertainty is impossible.

### 3.5.3 Dynamic Systems Development Method (DSDM)

#### WebRef

Useful resources for DSSD can be found at  
[www.dsdm.org](http://www.dsdm.org).

The *Dynamic Systems Development Method* (DSDM) [Sta97] is an agile software development approach that “provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment” [CCS02]. The DSDM philosophy is borrowed from a modified version of the Pareto principle—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete (100 percent) application.

DSDM is an iterative software process in which each iteration follows the 80 percent rule. That is, only enough work is required for each increment to facilitate movement to the next increment. The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

The DSDM Consortium ([www.dsdm.org](http://www.dsdm.org)) is a worldwide group of member companies that collectively take on the role of “keeper” of the method. The consortium has defined an agile process model, called the *DSDM life cycle* that defines three different iterative cycles, preceded by two additional life cycle activities:

*Feasibility study*—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.



DSDM is a process framework that can adopt the tactics of another agile approach such as XP.

*Business study*—establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.

*Functional model iteration*—produces a set of incremental prototypes that demonstrate functionality for the customer. (Note: All DSDM prototypes are intended to evolve into the deliverable application.) The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.

*Design and build iteration*—revisits prototypes built during *functional model iteration* to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, *functional model iteration* and *design and build iteration* occur concurrently.

*Implementation*—places the latest software increment (an “operationalized” prototype) into the operational environment. It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place. In either case, DSDM development work continues by returning to the functional model iteration activity.

DSDM can be combined with XP (Section 3.4) to provide a combination approach that defines a solid process model (the DSDM life cycle) with the nuts and bolts practices (XP) that are required to build software increments. In addition, the ASD concepts of collaboration and self-organizing teams can be adapted to a combined process model.

### 3.5.4 Crystal

Alistair Cockburn [Coc05] and Jim Highsmith [Hig02b] created the *Crystal family of agile methods*<sup>15</sup> in order to achieve a software development approach that puts a premium on “maneuverability” during what Cockburn characterizes as “a resource-limited, cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game” [Coc02].

To achieve maneuverability, Cockburn and Highsmith have defined a set of methodologies, each with core elements that are common to all, and roles, process patterns, work products, and practice that are unique to each. The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects. The intent is to allow agile teams to select the member of the crystal family that is most appropriate for their project and environment.



Crystal is a family of process models with the same “genetic code” but different methods for adapting to project characteristics.

15 The name “crystal” is derived from the characteristics of geological crystals, each with its own color, shape, and hardness.

### 3.5.5 Feature Driven Development (FDD)

*Feature Driven Development* (FDD) was originally conceived by Peter Coad and his colleagues [Coa99] as a practical process model for object-oriented software engineering. Stephen Palmer and John Felsing [Pal02] have extended and improved Coad's work, describing an adaptive, agile process that can be applied to moderately sized and larger software projects.

#### WebRef

A wide variety of articles and presentations on FDD can be found at:  
[www.featuredrivendevelopment.com/](http://www.featuredrivendevelopment.com/).

Like other agile approaches, FDD adopts a philosophy that (1) emphasizes collaboration among people on an FDD team; (2) manages problem and project complexity using feature-based decomposition followed by the integration of software increments, and (3) communication of technical detail using verbal, graphical, and text-based means. FDD emphasizes software quality assurance activities by encouraging an incremental development strategy, the use of design and code inspections, the application of software quality assurance audits (Chapter 16), the collection of metrics, and the use of patterns (for analysis, design, and construction).

In the context of FDD, a *feature* "is a client-valued function that can be implemented in two weeks or less" [Coa99]. The emphasis on the definition of features provides the following benefits:

- Because features are small blocks of deliverable functionality, users can describe them more easily; understand how they relate to one another more readily; and better review them for ambiguity, error, or omissions.
- Features can be organized into a hierarchical business-related grouping.
- Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks.
- Because features are small, their design and code representations are easier to inspect effectively.
- Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

Coad and his colleagues [Coa99] suggest the following template for defining a feature:

**<action> the <result> <by | for | of | to> a(n) <object>**

where an **<object>** is "a person, place, or thing (including roles, moments in time or intervals of time, or catalog-entry-like descriptions)." Examples of features for an e-commerce application might be:

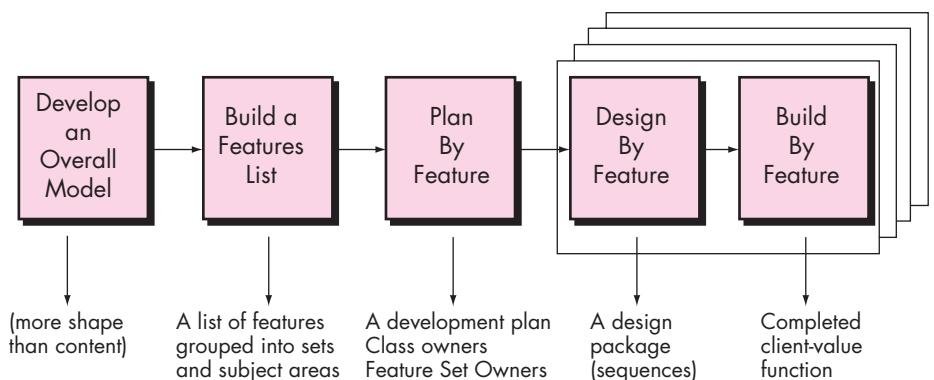
*Add the product to shopping cart*

*Display the technical-specifications of the product*

*Store the shipping-information for the customer*

**FIGURE 3.5**

**Feature Driven Development [Coa99] (with permission)**



A feature set groups related features into business-related categories and is defined [Coa99] as:

**<action><-ing> a(n) <object>**

For example: *Making a product sale* is a feature set that would encompass the features noted earlier and others.

The FDD approach defines five “collaborating” [Coa99] framework activities (in FDD these are called “processes”) as shown in Figure 3.5.

FDD provides greater emphasis on project management guidelines and techniques than many other agile methods. As projects grow in size and complexity, ad hoc project management is often inadequate. It is essential for developers, their managers, and other stakeholders to understand project status—what accomplishments have been made and problems have been encountered. If deadline pressure is significant, it is critical to determine if software increments (features) are properly scheduled. To accomplish this, FDD defines six milestones during the design and implementation of a feature: “design walkthrough, design, design inspection, code, code inspection, promote to build” [Coa99].

### 3.5.6 Lean Software Development (LSD)

*Lean Software Development* (LSD) has adapted the principles of lean manufacturing to the world of software engineering. The lean principles that inspire the LSD process can be summarized ([Pop03], [Pop06a]) as *eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people, and optimize the whole*.

Each of these principles can be adapted to the software process. For example, *eliminate waste* within the context of an agile software project can be interpreted to mean [Das05]: (1) adding no extraneous features or functions, (2) assessing the cost and schedule impact of any newly requested requirement, (3) removing any superfluous process steps, (4) establishing mechanisms to improve the way team members find information, (5) ensuring the testing finds as many errors as possible,

(6) reducing the time required to request and get a decision that affects the software or the process that is applied to create it, and (7) streamlining the manner in which information is transmitted to all stakeholders involved in the process.

For a detailed discussion of LSD and pragmatic guidelines for implementing the process, you should examine [Pop06a] and [Pop06b].

### 3.5.7 Agile Modeling (AM)

#### WebRef

Comprehensive information on agile modeling can be found at: [www.agilemodeling.com](http://www.agilemodeling.com).

There are many situations in which software engineers must build large, business-critical systems. The scope and complexity of such systems must be modeled so that (1) all constituencies can better understand what needs to be accomplished, (2) the problem can be partitioned effectively among the people who must solve it, and (3) quality can be assessed as the system is being engineered and built.

Over the past 30 years, a wide variety of software engineering modeling methods and notation have been proposed for analysis and design (both architectural and component-level). These methods have merit, but they have proven to be difficult to apply and challenging to sustain (over many projects). Part of the problem is the “weight” of these modeling methods. By this I mean the volume of notation required, the degree of formalism suggested, the sheer size of the models for large projects, and the difficulty in maintaining the model(s) as changes occur. Yet analysis and design modeling have substantial benefit for large projects—if for no other reason than to make these projects intellectually manageable. Is there an agile approach to software engineering modeling that might provide an alternative?

At “The Official Agile Modeling Site,” Scott Ambler [Amb02a] describes *agile modeling* (AM) in the following manner:

Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. Agile models are more effective than traditional models because they are just barely good, they don’t have to be perfect.

Agile modeling adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refactor. The team must also have the humility to recognize that technologists do not have all the answers and that business experts and other stakeholders should be respected and embraced.

Although AM suggests a wide array of “core” and “supplementary” modeling principles, those that make AM unique are [Amb02a]:

**Model with a purpose.** A developer who uses AM should have a specific goal (e.g., to communicate information to the customer or to help better understand some aspect of the software) in mind before creating the model.

Once the goal for the model is identified, the type of notation to be used and level of detail required will be more obvious.

#### Q uote:

“I was in the drug store the other day trying to get a cold medication . . . not easy. There’s an entire wall of products you need. You stand there going, Well, this one is quick acting but this is long lasting. . . . Which is more important, the present or the future?”

**Jerry Seinfeld**

**Use multiple models.** There are many different models and notations that can be used to describe software. Only a small subset is essential for most projects. AM suggests that to provide needed insight, each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.



*"Traveling light" is an appropriate philosophy for all software engineering work. Build only those models that provide value ... no more, no less.*

**Travel light.** As software engineering work proceeds, keep only those models that will provide long-term value and jettison the rest. Every work product that is kept must be maintained as changes occur. This represents work that slows the team down. Ambler [Amb02a] notes that "Every time you decide to keep a model you trade-off agility for the convenience of having that information available to your team in an abstract manner (hence potentially enhancing communication within your team as well as with project stakeholders)."

**Content is more important than representation.** Modeling should impart information to its intended audience. A syntactically perfect model that imparts little useful content is not as valuable as a model with flawed notation that nevertheless provides valuable content for its audience.

**Know the models and the tools you use to create them.** Understand the strengths and weaknesses of each model and the tools that are used to create it.

**Adapt locally.** The modeling approach should be adapted to the needs of the agile team.

A major segment of the software engineering community has adopted the Unified Modeling Language (UML)<sup>16</sup> as the preferred method for representing analysis and design models. The Unified Process (Chapter 2) has been developed to provide a framework for the application of UML. Scott Ambler [Amb06] has developed a simplified version of the UP that integrates his agile modeling philosophy.

### 3.5.8 Agile Unified Process (AUP)

The *Agile Unified Process* (AUP) adopts a "serial in the large" and "iterative in the small" [Amb06] philosophy for building computer-based systems. By adopting the classic UP phased activities—*inception, elaboration, construction, and transition*—AUP provides a serial overlay (i.e., a linear sequence of software engineering activities) that enables a team to visualize the overall process flow for a software project. However, within each of the activities, the team iterates to achieve agility and to deliver meaningful software increments to end users as rapidly as possible. Each AUP iteration addresses the following activities [Amb06]:

- *Modeling.* UML representations of the business and problem domains are created. However, to stay agile, these models should be "just barely good enough" [Amb06] to allow the team to proceed.

---

<sup>16</sup> A brief tutorial on UML is presented in Appendix 1.

- *Implementation.* Models are translated into source code.
- *Testing.* Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.
- *Deployment.* Like the generic process activity discussed in Chapters 1 and 2, deployment in this context focuses on the delivery of a software increment and the acquisition of feedback from end users.
- *Configuration and project management.* In the context of AUP, configuration management (Chapter 22) addresses change management, risk management, and the control of any persistent work products<sup>17</sup> that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.
- *Environment management.* Environment management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

Although the AUP has historical and technical connections to the Unified Modeling Language, it is important to note that UML modeling can be used in conjunction with any of the agile process models described in Section 3.5.

## SOFTWARE TOOLS



### Agile Development

**Objective:** The objective of agile development tools is to assist in one or more aspects of agile development with an emphasis on facilitating the rapid generation of operational software. These tools can also be used when prescriptive process models (Chapter 2) are applied.

**Mechanics:** Tool mechanics vary. In general, agile tool sets encompass automated support for project planning, use case development and requirements gathering, rapid design, code generation, and testing.

#### Representative Tools:<sup>18</sup>

Note: Because agile development is a hot topic, most software tools vendors purport to sell tools that support

the agile approach. The tools noted here have characteristics that make them particularly useful for agile projects.

*OnTime*, developed by Axosoft ([www.axosoft.com](http://www.axosoft.com)), provides agile process management support for various technical activities within the process.

*Ideogramic UML*, developed by Ideogramic ([www.ideogramic.com](http://www.ideogramic.com)) is a UML tool set specifically developed for use within an agile process.

*Together Tool Set*, distributed by Borland ([www.borland.com](http://www.borland.com)), provides a tools suite that supports many technical activities within XP and other agile processes.

<sup>17</sup> A *persistent work product* is a model or document or test case produced by the team that will be kept for an indeterminate period of time. It will *not* be discarded once the software increment is delivered.

<sup>18</sup> Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

### 3.6 A TOOL SET FOR THE AGILE PROCESS



The “tool set” that supports agile processes focuses more on people issues than it does on technology issues.

Some proponents of the agile philosophy argue that automated software tools (e.g., design tools) should be viewed as a minor supplement to the team’s activities, and not at all pivotal to the success of the team. However, Alistair Cockburn [Coc04] suggests that tools can have a benefit and that “agile teams stress using tools that permit the rapid flow of understanding. Some of those tools are social, starting even at the hiring stage. Some tools are technological, helping distributed teams simulate being physically present. Many tools are physical, allowing people to manipulate them in workshops.”

Because acquiring the right people (hiring), team collaboration, stakeholder communication, and indirect management are key elements in virtually all agile process models, Cockburn argues that “tools” that address these issues are critical success factors for agility. For example, a hiring “tool” might be the requirement to have a prospective team member spend a few hours pair programming with an existing member of the team. The “fit” can be assessed immediately.

Collaborative and communication “tools” are generally low tech and incorporate any mechanism (“physical proximity, whiteboards, poster sheets, index cards, and sticky notes” [Coc04]) that provides information and coordination among agile developers. Active communication is achieved via the team dynamics (e.g., pair programming), while passive communication is achieved by “information radiators” (e.g., a flat panel display that presents the overall status of different components of an increment). Project management tools deemphasize the Gantt chart and replace it with earned value charts or “graphs of tests created versus passed . . . other agile tools are used to optimize the environment in which the agile team works (e.g., more efficient meeting areas), improve the team culture by nurturing social interactions (e.g., collocated teams), physical devices (e.g., electronic whiteboards), and process enhancement (e.g., pair programming or time-boxing)” [Coc04].

Are any of these things really tools? They are, if they facilitate the work performed by an agile team member and enhance the quality of the end product.

### 3.7 SUMMARY

In a modern economy, market conditions change rapidly, customer and end-user needs evolve, and new competitive threats emerge without warning. Practitioners must approach software engineering in a manner that allows them to remain agile—to define maneuverable, adaptive, lean processes that can accommodate the needs of modern business.

An agile philosophy for software engineering stresses four key issues: the importance of self-organizing teams that have control over the work they perform, communication and collaboration between team members and between practitioners and their customers, a recognition that change represents an opportunity, and

an emphasis on rapid delivery of software that satisfies the customer. Agile process models have been designed to address each of these issues.

Extreme programming (XP) is the most widely used agile process. Organized as four framework activities—planning, design, coding, and testing—XP suggests a number of innovative and powerful techniques that allow an agile team to create frequent software releases that deliver features and functionality that have been described and then prioritized by stakeholders.

Other agile process models also stress human collaboration and team self-organization, but define their own framework activities and select different points of emphasis. For example, ASD uses an iterative process that incorporates adaptive cycle planning, relatively rigorous requirement gathering methods, and an iterative development cycle that incorporates customer focus groups and formal technical reviews as real-time feedback mechanisms. Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight time lines, changing requirements, and business criticality. Each process pattern defines a set of development tasks and allows the Scrum team to construct a process that is adapted to the needs of the project. The Dynamic Systems Development Method (DSDM) advocates the use of time-box scheduling and suggests that only enough work is required for each software increment to facilitate movement to the next increment. Crystal is a family of agile process models that can be adopted to the specific characteristics of a project.

Feature Driven Development (FDD) is somewhat more “formal” than other agile methods, but still maintains agility by focusing the project team on the development of features—a client-valued function that can be implemented in two weeks or less. Lean Software Development (LSD) has adapted the principles of lean manufacturing to the world of software engineering. Agile modeling (AM) suggests that modeling is essential for all systems, but that the complexity, type, and size of the model must be tuned to the software to be built. The Agile Unified Process (AUP) adopts a “serial in the large” and “iterative in the small” philosophy for building software.

### PROBLEMS AND POINTS TO PONDER

**3.1.** Reread “The Manifesto for Agile Software Development” at the beginning of this chapter. Can you think of a situation in which one or more of the four “values” could get a software team into trouble?

**3.2.** Describe agility (for software projects) in your own words.

**3.3.** Why does an iterative process make it easier to manage change? Is every agile process discussed in this chapter iterative? Is it possible to complete a project in just one iteration and still be agile? Explain your answers.

**3.4.** Could each of the agile processes be described using the generic framework activities noted in Chapter 2? Build a table that maps the generic activities into the activities defined for each agile process.

**3.5.** Try to come up with one more “agility principle” that would help a software engineering team become even more maneuverable.

**3.6.** Select one agility principle noted in Section 3.3.1 and try to determine whether each of the process models presented in this chapter exhibits the principle. [Note: I have presented an overview of these process models only, so it may not be possible to determine whether a principle has been addressed by one or more of the models, unless you do additional research (which is not required for this problem).]

**3.7.** Why do requirements change so much? After all, don't people know what they want?

**3.8.** Most agile process models recommend face-to-face communication. Yet today, members of a software team and their customers may be geographically separated from one another. Do you think this implies that geographical separation is something to avoid? Can you think of ways to overcome this problem?

**3.9.** Write an XP user story that describes the “favorite places” or “bookmarks” feature available on most Web browsers.

**3.10.** What is a spike solution in XP?

**3.11.** Describe the XP concepts of refactoring and pair programming in your own words.

**3.12.** Do a bit more reading and describe what a time-box is. How does this assist an ASD team in delivering software increments in a short time period?

**3.13.** Do the 80 percent rule in DSDM and the time-boxing approach defined for ASD achieve the same result?

**3.14.** Using the process pattern template presented in Chapter 2, develop a process pattern for any one of the Scrum patterns presented in Section 3.5.2.

**3.15.** Why is Crystal called a family of agile methods?

**3.16.** Using the FDD feature template described in Section 3.5.5, define a feature set for a Web browser. Now develop a set of features for the feature set.

**3.17.** Visit the Official Agile Modeling Site and make a complete list of all core and supplementary AM principles.

**3.18.** The tool set proposed in Section 3.6 supports many of the “soft” aspects of agile methods. Since communication is so important, recommend an actual tool set that might be used to enhance communication among stakeholders on an agile team.

## FURTHER READINGS AND INFORMATION SOURCES

The overall philosophy and underlying principles of agile software development are considered in depth in many of the books referenced in the body of this chapter. In addition, books by Shaw and Warden (*The Art of Agile Development*, O'Reilly Media, Inc., 2008), Hunt (*Agile Software Construction*, Springer, 2005), and Carmichael and Haywood (*Better Software Faster*, Prentice-Hall, 2002) present useful discussions of the subject. Aguanno (*Managing Agile Projects*, Multi-Media Publications, 2005), Highsmith (*Agile Project Management: Creating Innovative Products*, Addison-Wesley, 2004), and Larman (*Agile and Iterative Development: A Manager's Guide*, Addison-Wesley, 2003) present a management overview and consider project management issues. Highsmith (*Agile Software Development Ecosystems*, Addison-Wesley, 2002) presents a survey of agile principles, processes, and practices. A worthwhile discussion of the delicate balance between agility and discipline is presented by Booch and his colleagues (*Balancing Agility and Discipline*, Addison-Wesley, 2004).

Martin (*Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice-Hall, 2009) presents the principles, patterns, and practices required to develop “clean code” in an agile software engineering environment. Leffingwell (*Scaling Software Agility: Best Practices for Large Enterprises*, Addison-Wesley, 2007) discusses strategies for scaling up agile practices for large projects. Lippert and Rook (*Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*, Wiley, 2006) discuss the use of refactoring when applied in large, complex systems.

Stamelos and Sfetsos (*Agile Software Development Quality Assurance*, IGI Global, 2007) discuss SQA techniques that conform to the agile philosophy.

Dozens of books have been written about Extreme Programming over the past decade. Beck (*Extreme Programming Explained: Embrace Change*, 2d ed., Addison-Wesley, 2004) remains the definitive treatment of the subject. In addition, Jeffries and his colleagues (*Extreme Programming Installed*, Addison-Wesley, 2000), Succi and Marchesi (*Extreme Programming Examined*, Addison-Wesley, 2001), Newkirk and Martin (*Extreme Programming in Practice*, Addison-Wesley, 2001), and Auer and his colleagues (*Extreme Programming Applied: Play to Win*, Addison-Wesley, 2001) provide a nuts-and-bolts discussion of XP along with guidance on how best to apply it. McBreen (*Questioning Extreme Programming*, Addison-Wesley, 2003) takes a critical look at XP, defining when and where it is appropriate. An in-depth consideration of pair programming is presented by McBreen (*Pair Programming Illuminated*, Addison-Wesley, 2003).

ASD is addressed in depth by Highsmith [Hig00]. Schwaber (*The Enterprise and Scrum*, Microsoft Press, 2007) discusses the use of Scrum for projects that have a major business impact. The nuts and bolts of Scrum are discussed by Schwaber and Beedle (*Agile Software Development with SCRUM*, Prentice-Hall, 2001). Worthwhile treatments of DSDM have been written by the DSDM Consortium (*DSDM: Business Focused Development*, 2d ed., Pearson Education, 2003) and Stapleton (*DSDM: The Method in Practice*, Addison-Wesley, 1997). Cockburn (*Crystal Clear*, Addison-Wesley, 2005) presents an excellent overview of the Crystal family of processes. Palmer and Felsing [Pal02] present a detailed treatment of FDD. Carmichael and Haywood (*Better Software Faster*, Prentice-Hall, 2002) provides another useful treatment of FDD that includes a step-by-step journey through the mechanics of the process. Poppdieck and Poppdieck (*Lean Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003) provide guidelines for managing and controlling agile projects. Ambler and Jeffries (*Agile Modeling*, Wiley, 2002) discuss AM in some depth.

A wide variety of information sources on agile software development are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the agile process can be found at the SEPA website: [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm).

# Two

## MODELING

In this part of *Software Engineering: A Practitioner's Approach* you'll learn about the principles, concepts, and methods that are used to create high-quality requirements and design models. These questions are addressed in the chapters that follow:

- What concepts and principles guide software engineering practice?
- What is requirements engineering and what are the underlying concepts that lead to good requirements analysis?
- How is the requirements model created and what are its elements?
- What are the elements of a good design?
- How does architectural design establish a framework for all other design actions and what models are used?
- How do we design high-quality software components?
- What concepts, models, and methods are applied as a user interface is designed?
- What is pattern-based design?
- What specialized strategies and methods are used to design WebApps?

Once these questions are answered you'll be better prepared to apply software engineering practice.

## 4

PRINCIPLES THAT  
GUIDE PRACTICEKEY  
CONCEPTS**Core principles** ... 98**Principles that govern:**

coding ..... 111

communication ..... 101

deployment ..... 113

design ..... 109

modeling ..... 105

planning ..... 103

requirements ..... 107

testing ..... 112

QUICK  
LOOK

**What is it?** Software engineering practice is a broad array of principles, concepts, methods, and tools that you must consider as software is planned and developed. Principles that guide practice establish a foundation from which software engineering is conducted.

**Who does it?** Practitioners (software engineers) and their managers conduct a variety of software engineering tasks.

**Why is it important?** The software process provides everyone involved in the creation of a computer-based system or product with a road map for getting to a successful destination. Practice provides you with the detail you'll need to drive along the road. It tells you where the bridges, the roadblocks, and the forks are located. It helps you understand the concepts and principles that must be understood and followed to drive safely and rapidly. It instructs you on how to drive, where to slow down, and where to speed up. In the context of software engineering,

In a book that explores the lives and thoughts of software engineers, Ellen Ullman [Ull97] depicts a slice of life as she relates the thoughts of practitioner under pressure:

I have no idea what time it is. There are no windows in this office and no clock, only the blinking red LED display of a microwave, which flashes 12:00, 12:00, 12:00, 12:00. Joel and I have been programming for days. We have a bug, a stubborn demon of a bug. So the red pulse no-time feels right, like a read-out of our brains, which have somehow synchronized themselves at the same blink rate . . .

What are we working on? . . . The details escape me just now. We may be helping poor sick people or tuning a set of low-level routines to verify bits on a distributed database protocol—I don't care. I should care; in another part of my being—later, perhaps when we emerge from this room full of computers—I will care very much why and for whom and for what purpose I am writing software. But just now: no. I have passed through a membrane where the real world and its uses no longer matter. I am a software engineer. . . .

practice is what you do day in and day out as software evolves from an idea to a reality.

**What are the steps?** Three elements of practice apply regardless of the process model that is chosen. They are: principles, concepts, and methods. A fourth element of practice—tools—supports the application of methods.

**What is the work product?** Practice encompasses the technical activities that produce all work products that are defined by the software process model that has been chosen.

**How do I ensure that I've done it right?** First, have a firm understanding of the principles that apply to the work (e.g., design) that you're doing at the moment. Then, be certain that you've chosen an appropriate method for the work, be sure that you understand how to apply the method, use automated tools when they're appropriate for the task, and be adamant about the need for techniques to ensure the quality of work products that are produced.

A dark image of software engineering practice to be sure, but upon reflection, many of the readers of this book will be able to relate to it.

People who create computer software practice the art or craft or discipline<sup>1</sup> that is software engineering. But what is software engineering “practice”? In a generic sense, *practice* is a collection of concepts, principles, methods, and tools that a software engineer calls upon on a daily basis. Practice allows managers to manage software projects and software engineers to build computer programs. Practice populates a software process model with the necessary technical and management how-to’s to get the job done. Practice transforms a haphazard unfocused approach into something that is more organized, more effective, and more likely to achieve success.

Various aspects of software engineering practice will be examined throughout the remainder of this book. In this chapter, my focus is on principles and concepts that guide software engineering practice in general.

## 4.1 SOFTWARE ENGINEERING KNOWLEDGE

In an editorial published in *IEEE Software* a decade ago, Steve McConnell [McC99] made the following comment:

Many software practitioners think of software engineering knowledge almost exclusively as knowledge of specific technologies: Java, Perl, html, C++, Linux, Windows NT, and so on. Knowledge of specific technology details is necessary to perform computer programming. If someone assigns you to write a program in C++, you have to know something about C++ to get your program to work.

You often hear people say that software development knowledge has a 3-year half-life: half of what you need to know today will be obsolete within 3 years. In the domain of technology-related knowledge, that's probably about right. But there is another kind of software development knowledge—a kind that I think of as “software engineering principles”—that does not have a three-year half-life. These software engineering principles are likely to serve a professional programmer throughout his or her career.

McConnell goes on to argue that the body of software engineering knowledge (circa the year 2000) had evolved to a “stable core” that he estimated represented about “75 percent of the knowledge needed to develop a complex system.” But what resides within this stable core?

As McConnell indicates, core principles—the elemental ideas that guide software engineers in the work that they do—now provide a foundation from which software engineering models, methods, and tools can be applied and evaluated.

---

<sup>1</sup> Some writers argue for one of these terms to the exclusion of the others. In reality, software engineering is all three.

## 4.2 CORE PRINCIPLES

**Quote:**

"In theory there is no difference between theory and practice. But, in practice, there is."

Jan van de Snepscheut

Software engineering is guided by a collection of core principles that help in the application of a meaningful software process and the execution of effective software engineering methods. At the process level, core principles establish a philosophical foundation that guides a software team as it performs framework and umbrella activities, navigates the process flow, and produces a set of software engineering work products. At the level of practice, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community.

In Chapter 1, I identified a set of general principles that span software engineering process and practice: (1) provide value to end users, (2) keep it simple, (3) maintain the vision (of the product and the project), (4) recognize that others consume (and must understand) what you produce, (5) be open to the future, (6) plan ahead for reuse, and (7) think! Although these general principles are important, they are characterized at such a high level of abstraction that they are sometimes difficult to translate into day-to-day software engineering practice. In the subsections that follow, I take a more detailed look at the core principles that guide process and practice.

### 4.2.1 Principles That Guide Process

In Part 1 of this book I discussed the importance of the software process and described the many different process models that have been proposed for software engineering work. Regardless of whether a model is linear or iterative, prescriptive or agile, it can be characterized using the generic process framework that is applicable for all process models. The following set of core principles can be applied to the framework, and by extension, to every software process.



*Every project and every team is unique. That means that you must adapt your process to best fit your needs.*

**Principle 1. Be agile.** Whether the process model you choose is prescriptive or agile, the basic tenets of agile development should govern your approach. Every aspect of the work you do should emphasize economy of action—keep your technical approach as simple as possible, keep the work products you produce as concise as possible, and make decisions locally whenever possible.

**Principle 2. Focus on quality at every step.** The exit condition for every process activity, action, and task should focus on the quality of the work product that has been produced.

**Principle 3. Be ready to adapt.** Process is not a religious experience, and dogma has no place in it. When necessary, adapt your approach to constraints imposed by the problem, the people, and the project itself.

**Principle 4. Build an effective team.** Software engineering process and practice are important, but the bottom line is people. Build a self-organizing team that has mutual trust and respect.

**Principle 5. Establish mechanisms for communication and coordination.**

Projects fail because important information falls into the cracks and/or stakeholders fail to coordinate their efforts to create a successful end product. These are management issues and they must be addressed.

 **note:**

"The truth of the matter is that you always know the right thing to do. The hard part is doing it."

General H.  
Norman  
Schwarzkopf

**Principle 6. Manage change.** The approach may be either formal or informal, but mechanisms must be established to manage the way changes are requested, assessed, approved, and implemented.

**Principle 7. Assess risk.** Lots of things can go wrong as software is being developed. It's essential that you establish contingency plans.

**Principle 8. Create work products that provide value for others.**

Create only those work products that provide value for other process activities, actions, or tasks. Every work product that is produced as part of software engineering practice will be passed on to someone else. A list of required functions and features will be passed along to the person (people) who will develop a design, the design will be passed along to those who generate code, and so on. Be sure that the work product imparts the necessary information without ambiguity or omission.

Part 4 of this book focuses on project and process management issues and considers various aspects of each of these principles in some detail.

#### 4.2.2 Principles That Guide Practice

Software engineering practice has a single overriding goal—to deliver on-time, high-quality, operational software that contains functions and features that meet the needs of all stakeholders. To achieve this goal, you should adopt a set of core principles that guide your technical work. These principles have merit regardless of the analysis and design methods that you apply, the construction techniques (e.g., programming language, automated tools) that you use, or the verification and validation approach that you choose. The following set of core principles are fundamental to the practice of software engineering:



Problems are easier to solve when they are subdivided into separate concerns, each distinct, individually solvable, and verifiable.

**Principle 1. Divide and conquer.** Stated in a more technical manner, analysis and design should always emphasize *separation of concerns* (SoC). A large problem is easier to solve if it is subdivided into a collection of elements (or *concerns*). Ideally, each concern delivers distinct functionality that can be developed, and in some cases validated, independently of other concerns.

**Principle 2. Understand the use of abstraction.** At its core, an abstraction is a simplification of some complex element of a system used to communicate meaning in a single phrase. When I use the abstraction *spreadsheet*, it is assumed that you understand what a spreadsheet is, the general structure of content that a spreadsheet presents, and the typical functions that can be applied to it. In software engineering practice, you use many different levels

of abstraction, each imparting or implying meaning that must be communicated. In analysis and design work, a software team normally begins with models that represent high levels of abstraction (e.g., a spreadsheet) and slowly refines those models into lower levels of abstraction (e.g., a column or the *SUM* function).

Joel Spolsky [Spo02] suggests that “all non-trivial abstractions, to some degree, are leaky.” The intent of an abstraction is to eliminate the need to communicate details. But sometimes, problematic effects precipitated by these details “leak” through. Without an understanding of the details, the cause of a problem cannot be easily diagnosed.

**Principle 3. Strive for consistency.** Whether it’s creating a requirements model, developing a software design, generating source code, or creating test cases, the principle of consistency suggests that a familiar context makes software easier to use. As an example, consider the design of a user interface for a WebApp. Consistent placement of menu options, the use of a consistent color scheme, and the consistent use of recognizable icons all help to make the interface ergonomically sound.

**Principle 4. Focus on the transfer of information.** Software is about information transfer—from a database to an end user, from a legacy system to a WebApp, from an end user into a graphic user interface (GUI), from an operating system to an application, from one software component to another—the list is almost endless. In every case, information flows across an interface, and as a consequence, there are opportunities for error, or omission, or ambiguity. The implication of this principle is that you must pay special attention to the analysis, design, construction, and testing of interfaces.

**Principle 5. Build software that exhibits effective modularity.** Separation of concerns (Principle 1) establishes a philosophy for software. *Modularity* provides a mechanism for realizing the philosophy. Any complex system can be divided into modules (components), but good software engineering practice demands more. Modularity must be *effective*. That is, each module should focus exclusively on one well-constrained aspect of the system—it should be cohesive in its function and/or constrained in the content it represents. Additionally, modules should be interconnected in a relatively simple manner—each module should exhibit low coupling to other modules, to data sources, and to other environmental aspects.

**Principle 6. Look for patterns.** Brad Appleton [App00] suggests that:

The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions. Formally codifying these solutions and their relationships lets us successfully capture the



Use patterns  
(Chapter 12) to  
capture knowledge and  
experience for future  
generations of  
software engineers.

body of knowledge which defines our understanding of good architectures that meet the needs of their users.

**Principle 7. When possible, represent the problem and its solution from a number of different perspectives.** When a problem and its solution are examined from a number of different perspectives, it is more likely that greater insight will be achieved and that errors and omissions will be uncovered. For example, a requirements model can be represented using a data-oriented viewpoint, a function-oriented viewpoint, or a behavioral viewpoint (Chapters 6 and 7). Each provides a different view of the problem and its requirements.

**Principle 8. Remember that someone will maintain the software.** Over the long term, software will be corrected as defects are uncovered, adapted as its environment changes, and enhanced as stakeholders request more capabilities. These maintenance activities can be facilitated if solid software engineering practice is applied throughout the software process.

These principles are not all you'll need to build high-quality software, but they do establish a foundation for every software engineering method discussed in this book.

## 4.3 PRINCIPLES THAT GUIDE EACH FRAMEWORK ACTIVITY

### quote:

"The ideal engineer is a composite. . . . He is not a scientist, he is not a mathematician, he is not a sociologist or a writer; but he may use the knowledge and techniques of any or all of these disciplines in solving engineering problems."

N. W.  
Dougherty

In the sections that follow I consider principles that have a strong bearing on the success of each generic framework activity defined as part of the software process. In many cases, the principles that are discussed for each of the framework activities are a refinement of the principles presented in Section 4.2. They are simply core principles stated at a lower level of abstraction.

### 4.3.1 Communication Principles

Before customer requirements can be analyzed, modeled, or specified they must be gathered through the communication activity. A customer has a problem that may be amenable to a computer-based solution. You respond to the customer's request for help. Communication has begun. But the road from communication to understanding is often full of potholes.

Effective communication (among technical peers, with the customer and other stakeholders, and with project managers) is among the most challenging activities that you will confront. In this context, I discuss communication principles as they apply to customer communication. However, many of the principles apply equally to all forms of communication that occur within a software project.

**Principle 1. Listen.** Try to focus on the speaker's words, rather than formulating your response to those words. Ask for clarification if something is unclear, but avoid constant interruptions. Never become contentious in your words or actions (e.g., rolling your eyes or shaking your head) as a person is talking.



*Before communicating be sure you understand the point of view of the other party, know a bit about his or her needs, and then listen.*



Mark Twain



**Principle 2. Prepare before you communicate.** Spend the time to understand the problem before you meet with others. If necessary, do some research to understand business domain jargon. If you have responsibility for conducting a meeting, prepare an agenda in advance of the meeting.

**Principle 3. Someone should facilitate the activity.** Every communication meeting should have a leader (a facilitator) to keep the conversation moving in a productive direction, (2) to mediate any conflict that does occur, and (3) to ensure than other principles are followed.

**Principle 4. Face-to-face communication is best.** But it usually works better when some other representation of the relevant information is present. For example, a participant may create a drawing or a "strawman" document that serves as a focus for discussion.

**Principle 5. Take notes and document decisions.** Things have a way of falling into the cracks. Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.

**Principle 6. Strive for collaboration.** Collaboration and consensus occur when the collective knowledge of members of the team is used to describe product or system functions or features. Each small collaboration serves to build trust among team members and creates a common goal for the team.

**Principle 7. Stay focused; modularize your discussion.** The more people involved in any communication, the more likely that discussion will bounce from one topic to the next. The facilitator should keep the conversation modular, leaving one topic only after it has been resolved (however, see Principle 9).

**Principle 8. If something is unclear, draw a picture.** Verbal communication goes only so far. A sketch or drawing can often provide clarity when words fail to do the job.

**Principle 9. (a) Once you agree to something, move on. (b) If you can't agree to something, move on. (c) If a feature or function is unclear and cannot be clarified at the moment, move on.** Communication, like any software engineering activity, takes time. Rather than iterating endlessly, the people who participate should recognize that many topics require discussion (see Principle 2) and that "moving on" is sometimes the best way to achieve communication agility.

**Principle 10. Negotiation is not a contest or a game. It works best when both parties win.** There are many instances in which you and other stakeholders must negotiate functions and features, priorities, and delivery dates. If the team has collaborated well, all parties have a common goal. Still, negotiation will demand compromise from all parties.

## INFO

**The Difference Between Customers and End Users**

Software engineers communicate with many different stakeholders, but customers and end users have the most significant impact on the technical work that follows. In some cases the customer and the end user are one and the same, but for many projects, the customer and the end user are different people, working for different managers, in different business organizations.

A *customer* is the person or group who (1) originally requested the software to be built, (2) defines overall business objectives for the software, (3) provides basic

product requirements, and (4) coordinates funding for the project. In a product or system business, the customer is often the marketing department. In an information technology (IT) environment, the customer might be a business component or department.

An *end user* is the person or group who (1) will actually use the software that is built to achieve some business purpose and (2) will define operational details of the software so the business purpose can be achieved.

**SAFEHOME****Communication Mistakes**

team workspace

**The players:** Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member.

**The conversation:**

**Ed:** "What have you heard about this *SafeHome* project?"

**Vinod:** "The kick-off meeting is scheduled for next week."

**Jamie:** "I've already done a little bit of investigation, but it didn't go well."

**Ed:** "What do you mean?"

**Jamie:** "Well, I gave Lisa Perez a call. She's the marketing honcho on this thing."

**Vinod:** "And . . . ?"

**Jamie:** "I wanted her to tell me about *SafeHome* features and functions . . . that sort of thing. Instead, she began

asking me questions about security systems, surveillance systems . . . I'm no expert."

**Vinod:** "What does that tell you?"

(Jamie shrugs.)

**Vinod:** "That marketing will need us to act as consultants and that we'd better do some homework on this product area before our kick-off meeting. Doug said that he wanted us to 'collaborate' with our customer, so we'd better learn how to do that."

**Ed:** "Probably would have been better to stop by her office. Phone calls just don't work as well for this sort of thing."

**Jamie:** "You're both right. We've got to get our act together or our early communications will be a struggle."

**Vinod:** "I saw Doug reading a book on 'requirements engineering.' I'll bet that lists some principles of good communication. I'm going to borrow it from him."

**Jamie:** "Good idea . . . then you can teach us."

**Vinod (smiling):** "Yeah, right."

**4.3.2 Planning Principles**

The communication activity helps you to define your overall goals and objectives (subject, of course, to change as time passes). However, understanding these goals and objectives is not the same as defining a plan for getting there. The planning activity encompasses a set of management and technical practices that enable the software team to define a road map as it travels toward its strategic goal and tactical objectives.

**note:**

"In preparing for battle I have always found that plans are useless, but planning is indispensable."

**General Dwight D. Eisenhower**

**WebRef**

An excellent repository of planning and project management information can be found at [www.4pm.com/repository.htm](http://www.4pm.com/repository.htm).

Try as we might, it's impossible to predict exactly how a software project will evolve. There is no easy way to determine what unforeseen technical problems will be encountered, what important information will remain undiscovered until late in the project, what misunderstandings will occur, or what business issues will change. And yet, a good software team must plan its approach.

There are many different planning philosophies.<sup>2</sup> Some people are "minimalists," arguing that change often obviates the need for a detailed plan. Others are "traditionalists," arguing that the plan provides an effective road map and the more detail it has, the less likely the team will become lost. Still others are "agilists," arguing that a quick "planning game" may be necessary, but that the road map will emerge as "real work" on the software begins.

What to do? On many projects, overplanning is time consuming and fruitless (too many things change), but underplanning is a recipe for chaos. Like most things in life, planning should be conducted in moderation, enough to provide useful guidance for the team—no more, no less. Regardless of the rigor with which planning is conducted, the following principles always apply:

**Principle 1. Understand the scope of the project.** It's impossible to use a road map if you don't know where you're going. Scope provides the software team with a destination.

**Principle 2. Involve stakeholders in the planning activity.** Stakeholders define priorities and establish project constraints. To accommodate these realities, software engineers must often negotiate order of delivery, time lines, and other project-related issues.

**Principle 3. Recognize that planning is iterative.** A project plan is never engraved in stone. As work begins, it is very likely that things will change. As a consequence, the plan must be adjusted to accommodate these changes. In addition, iterative, incremental process models dictate replanning after the delivery of each software increment based on feedback received from users.

**Principle 4. Estimate based on what you know.** The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done. If information is vague or unreliable, estimates will be equally unreliable.

**Principle 5. Consider risk as you define the plan.** If you have identified risks that have high impact and high probability, contingency planning is necessary. In addition, the project plan (including the schedule) should be adjusted to accommodate the likelihood that one or more of these risks will occur.

**note:**

"Success is more a function of consistent common sense than it is of genius."

**An Wang**

---

<sup>2</sup> A detailed discussion of software project planning and management is presented in Part 4 of this book.

 **KEY POINT**

The term *granularity* refers to the detail with which some element of planning is represented or conducted.

**Principle 6. Be realistic.** People don't work 100 percent of every day.

Noise always enters into any human communication. Omissions and ambiguity are facts of life. Change will occur. Even the best software engineers make mistakes. These and other realities should be considered as a project plan is established.

**Principle 7. Adjust granularity as you define the plan.** *Granularity*

refers to the level of detail that is introduced as a project plan is developed. A "high-granularity" plan provides significant work task detail that is planned over relatively short time increments (so that tracking and control occur frequently). A "low-granularity" plan provides broader work tasks that are planned over longer time periods. In general, granularity moves from high to low as the project time line moves away from the current date. Over the next few weeks or months, the project can be planned in significant detail. Activities that won't occur for many months do not require high granularity (too much can change).

**Principle 8. Define how you intend to ensure quality.** The plan should identify how the software team intends to ensure quality. If technical reviews<sup>3</sup> are to be conducted, they should be scheduled. If pair programming (Chapter 3) is to be used during construction, it should be explicitly defined within the plan.

**Principle 9. Describe how you intend to accommodate change.** Even the best planning can be obviated by uncontrolled change. You should identify how changes are to be accommodated as software engineering work proceeds. For example, can the customer request a change at any time? If a change is requested, is the team obliged to implement it immediately? How is the impact and cost of the change assessed?

**Principle 10. Track the plan frequently and make adjustments as required.** Software projects fall behind schedule one day at a time. Therefore, it makes sense to track progress on a daily basis, looking for problem areas and situations in which scheduled work does not conform to actual work conducted. When slippage is encountered, the plan is adjusted accordingly.

To be most effective, everyone on the software team should participate in the planning activity. Only then will team members "sign up" to the plan.

#### 4.3.3 Modeling Principles

We create models to gain a better understanding of the actual entity to be built. When the entity is a physical thing (e.g., a building, a plane, a machine), we can build a model that is identical in form and shape but smaller in scale. However, when the

---

<sup>3</sup> Technical reviews are discussed in Chapter 15.

entity to be built is software, our model must take a different form. It must be capable of representing the information that software transforms, the architecture and functions that enable the transformation to occur, the features that users desire, and the behavior of the system as the transformation is taking place. Models must accomplish these objectives at different levels of abstraction—first depicting the software from the customer's viewpoint and later representing the software at a more technical level.

## KEY POINT

Requirements models represent customer requirements. Design models provide a concrete specification for the construction of the software.

In software engineering work, two classes of models can be created: requirements models and design models. *Requirements models* (also called *analysis models*) represent customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioral domain. *Design models* represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

In their book on agile modeling, Scott Ambler and Ron Jeffries [Amb02b] define a set of modeling principles<sup>4</sup> that are intended for those who use the agile process model (Chapter 3) but are appropriate for all software engineers who perform modeling actions and tasks:

**Principle 1. The primary goal of the software team is to build software, not create models.** Agility means getting software to the customer in the fastest possible time. Models that make this happen are worth creating, but models that slow the process down or provide little new insight should be avoided.

**Principle 2. Travel light—don't create more models than you need.**

Every model that is created must be kept up-to-date as changes occur. More importantly, every new model takes time that might otherwise be spent on construction (coding and testing). Therefore, create only those models that make it easier and faster to construct the software.

## ADVICE

The intent of any model is to communicate information. To accomplish this, use a consistent format. Assume that you won't be there to explain the model. It should stand on its own.

**Principle 3. Strive to produce the simplest model that will describe the problem or the software.** Don't overbuild the software [Amb02b]. By keeping models simple, the resultant software will also be simple. The result is software that is easier to integrate, easier to test, and easier to maintain (to change). In addition, simple models are easier for members of the software team to understand and critique, resulting in an ongoing form of feedback that optimizes the end result.

**Principle 4. Build models in a way that makes them amenable to change.**

Assume that your models will change, but in making this assumption don't

---

<sup>4</sup> The principles noted in this section have been abbreviated and rephrased for the purposes of this book.

get sloppy. For example, since requirements will change, there is a tendency to give requirements models short shrift. Why? Because you know that they'll change anyway. The problem with this attitude is that without a reasonably complete requirements model, you'll create a design (design model) that will invariably miss important functions and features.

**Principle 5. Be able to state an explicit purpose for each model that is created.** Every time you create a model, ask yourself why you're doing so. If you can't provide solid justification for the existence of the model, don't spend time on it.

**Principle 6. Adapt the models you develop to the system at hand.** It may be necessary to adapt model notation or rules to the application; for example, a video game application might require a different modeling technique than real-time, embedded software that controls an automobile engine.

**Principle 7. Try to build useful models, but forget about building perfect models.** When building requirements and design models, a software engineer reaches a point of diminishing returns. That is, the effort required to make the model absolutely complete and internally consistent is not worth the benefits of these properties. Am I suggesting that modeling should be sloppy or low quality? The answer is "no." But modeling should be conducted with an eye to the next software engineering steps. Iterating endlessly to make a model "perfect" does not serve the need for agility.

**Principle 8. Don't become dogmatic about the syntax of the model. If it communicates content successfully, representation is secondary.**

Although everyone on a software team should try to use consistent notation during modeling, the most important characteristic of the model is to communicate information that enables the next software engineering task. If a model does this successfully, incorrect syntax can be forgiven.

**Principle 9. If your instincts tell you a model isn't right even though it seems okay on paper, you probably have reason to be concerned.** If you are an experienced software engineer, trust your instincts. Software work teaches many lessons—some of them on a subconscious level. If something tells you that a design model is doomed to fail (even though you can't prove it explicitly), you have reason to spend additional time examining the model or developing a different one.

**Principle 10. Get feedback as soon as you can.** Every model should be reviewed by members of the software team. The intent of these reviews is to provide feedback that can be used to correct modeling mistakes, change misinterpretations, and add features or functions that were inadvertently omitted.

**Requirements modeling principles.** Over the past three decades, a large number of requirements modeling methods have been developed. Investigators have

identified requirements analysis problems and their causes and have developed a variety of modeling notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view. However, all analysis methods are related by a set of operational principles:

**Principle 1. *The information domain of a problem must be represented and understood.*** The *information domain* encompasses the data that flow into the system (from end users, other systems, or external devices), the data that flow out of the system (via the user interface, network interfaces, reports, graphics, and other means), and the data stores that collect and organize persistent data objects (i.e., data that are maintained permanently).

**Principle 2. *The functions that the software performs must be defined.*** Software functions provide direct benefit to end users and also provide internal support for those features that are user visible. Some functions transform data that flow into the system. In other cases, functions effect some level of control over internal software processing or external system elements. Functions can be described at many different levels of abstraction, ranging from a general statement of purpose to a detailed description of the processing elements that must be invoked.

**Principle 3. *The behavior of the software (as a consequence of external events) must be represented.*** The behavior of computer software is driven by its interaction with the external environment. Input provided by end users, control data provided by an external system, or monitoring data collected over a network all cause the software to behave in a specific way.

**Principle 4. *The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.*** Requirements modeling is the first step in software engineering problem solving. It allows you to better understand the problem and establishes a basis for the solution (design). Complex problems are difficult to solve in their entirety. For this reason, you should use a divide-and-conquer strategy. A large, complex problem is divided into subproblems until each subproblem is relatively easy to understand. This concept is called *partitioning* or *separation of concerns*, and it is a key strategy in requirements modeling.

**Principle 5. *The analysis task should move from essential information toward implementation detail.*** Requirements modeling begins by describing the problem from the end-user's perspective. The "essence" of the problem is described without any consideration of how a solution will be implemented. For example, a video game requires that the player "instruct" its protagonist on what direction to proceed as she moves into a dangerous maze. That is the essence of the problem. Implementation detail (normally described as part of the design model) indicates how the essence will be implemented. For the video game, voice input might be used. Alternatively,

## KEY POINT

Analysis modeling focuses on three attributes of software: information to be processed, function to be delivered, and behavior to be exhibited.

### Quote:

"The engineer's first problem in any design situation is to discover what the problem really is."

Author unknown

a keyboard command might be typed, a joystick (or mouse) might be pointed in a specific direction, or a motion-sensitive device might be waved in the air.

By applying these principles, a software engineer approaches a problem systematically. But how are these principles applied in practice? This question will be answered in Chapters 5 through 7.

**Design Modeling Principles.** The software design model is analogous to an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the system.

There is no shortage of methods for deriving the various elements of a software design. Some methods are data driven, allowing the data structure to dictate the program architecture and the resultant processing components. Others are pattern driven, using information about the problem domain (the requirements model) to develop architectural styles and processing patterns. Still others are object oriented, using problem domain objects as the driver for the creation of data structures and the methods that manipulate them. Yet all embrace a set of design principles that can be applied regardless of the method that is used:

**Principle 1. Design should be traceable to the requirements model.**

The requirements model describes the information domain of the problem, user-visible functions, system behavior, and a set of requirements classes that package business objects with the methods that service them. The design model translates this information into an architecture, a set of subsystems that implement major functions, and a set of components that are the realization of requirements classes. The elements of the design model should be traceable to the requirements model.

**Principle 2. Always consider the architecture of the system to be built.**

Software architecture (Chapter 9) is the skeleton of the system to be built. It affects interfaces, data structures, program control flow and behavior, the manner in which testing can be conducted, the maintainability of the resultant system, and much more. For all of these reasons, design should start with architectural considerations. Only after the architecture has been established should component-level issues be considered.

**Principle 3. Design of data is as important as design of processing functions.**

Data design is an essential element of architectural design. The manner in which data objects are realized within the design cannot be left to chance. A well-structured data design helps to simplify program flow, makes the design and implementation of software components easier, and makes overall processing more efficient.

**quote:**

"See first that the design is wise and just: that ascertained, pursue it resolutely; do not for one repulse forego the purpose that you resolved to effect."

William Shakespeare

**WebRef**

Insightful comments on the design process, along with a discussion of design aesthetics, can be found at [cs.wwc.edu/~abyan/Design/](http://cs.wwc.edu/~abyan/Design/).

**Quote:**

"The differences are not minor—they are rather like the differences between Salieri and Mozart. Study after study shows that the very best designers produce structures that are faster, smaller, simpler, clearer, and produced with less effort."

Frederick P.  
Brooks

**Principle 4. Interfaces (both internal and external) must be designed with care.** The manner in which data flows between the components of a system has much to do with processing efficiency, error propagation, and design simplicity. A well-designed interface makes integration easier and assists the tester in validating component functions.

**Principle 5. User interface design should be tuned to the needs of the end user. However, in every case, it should stress ease of use.** The user interface is the visible manifestation of the software. No matter how sophisticated its internal functions, no matter how comprehensive its data structures, no matter how well designed its architecture, a poor interface design often leads to the perception that the software is "bad."

**Principle 6. Component-level design should be functionally independent.** Functional independence is a measure of the "single-mindedness" of a software component. The functionality that is delivered by a component should be cohesive—that is, it should focus on one and only one function or subfunction.<sup>5</sup>

**Principle 7. Components should be loosely coupled to one another and to the external environment.** Coupling is achieved in many ways—via a component interface, by messaging, through global data. As the level of coupling increases, the likelihood of error propagation also increases and the overall maintainability of the software decreases. Therefore, component coupling should be kept as low as is reasonable.

**Principle 8. Design representations (models) should be easily understandable.** The purpose of design is to communicate information to practitioners who will generate code, to those who will test the software, and to others who may maintain the software in the future. If the design is difficult to understand, it will not serve as an effective communication medium.

**Principle 9. The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity.** Like almost all creative activities, design occurs iteratively. The first iterations work to refine the design and correct errors, but later iterations should strive to make the design as simple as is possible.

When these design principles are properly applied, you create a design that exhibits both external and internal quality factors [Mye78]. *External quality factors* are those properties of the software that can be readily observed by users (e.g., speed, reliability, correctness, usability). *Internal quality factors* are of importance to software engineers. They lead to a high-quality design from the technical perspective. To achieve internal quality factors, the designer must understand basic design concepts (Chapter 8).

---

<sup>5</sup> Additional discussion of cohesion can be found in Chapter 8.

 **Quote:**

"For much of my life, I have been a software voyeur, peeking furtively at other people's dirty code. Occasionally, I find a real jewel, a well-structured program written in a consistent style, free of kludges, developed so that each component is simple and organized, and designed so that the product is easy to change."

David Parnas

 **ADVICE**

Avoid developing an elegant program that solves the wrong problem. Pay particular attention to the first preparation principle.

### 4.3.4 Construction Principles

The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end user. In modern software engineering work, coding may be (1) the direct creation of programming language source code (e.g., Java), (2) the automatic generation of source code using an intermediate design-like representation of the component to be built, or (3) the automatic generation of executable code using a "fourth-generation programming language" (e.g., Visual C++).

The initial focus of testing is at the component level, often called *unit testing*. Other levels of testing include (1) *integration testing* (conducted as the system is constructed), *validation testing* that assesses whether requirements have been met for the complete system (or software increment), and (3) *acceptance testing* that is conducted by the customer in an effort to exercise all required features and functions. The following set of fundamental principles and concepts are applicable to coding and testing:

**Coding Principles.** The principles that guide the coding task are closely aligned with programming style, programming languages, and programming methods. However, there are a number of fundamental principles that can be stated:

**Preparation principles: Before you write one line of code, be sure you**

- Understand of the problem you're trying to solve.
- Understand basic design principles and concepts.
- Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
- Select a programming environment that provides tools that will make your work easier.
- Create a set of unit tests that will be applied once the component you code is completed.

**Programming principles: As you begin writing code, be sure you**

- Constrain your algorithms by following structured programming [Boh00] practice.
- Consider the use of pair programming.
- Select data structures that will meet the needs of the design.
- Understand the software architecture and create interfaces that are consistent with it.
- Keep conditional logic as simple as possible.
- Create nested loops in a way that makes them easily testable.
- Select meaningful variable names and follow other local coding standards.

- Write code that is self-documenting.
- Create a visual layout (e.g., indentation and blank lines) that aids understanding.

**Validation Principles:** *After you've completed your first coding pass, be sure you*

- Conduct a code walkthrough when appropriate.
- Perform unit tests and correct errors you've uncovered.
- Refactor the code.

**WebRef**

A wide variety of links to coding standards can be found at [www.literateprogramming.com/lpstyle.html](http://www.literateprogramming.com/lpstyle.html).

More books have been written about programming (coding) and the principles and concepts that guide it than about any other topic in the software process. Books on the subject include early works on programming style [Ker78], practical software construction [McC04], programming pearls [Ben99], the art of programming [Knu98], pragmatic programming issues [Hun99], and many, many other subjects. A comprehensive discussion of these principles and concepts is beyond the scope of this book. If you have further interest, examine one or more of the references noted.

**Testing Principles.** In a classic book on software testing, Glen Myers [Mye79] states a number of rules that can serve well as testing objectives:

-  **What are the objectives of software testing?**
- Testing is a process of executing a program with the intent of finding an error.
  - A good test case is one that has a high probability of finding an as-yet-undiscovered error.
  - A successful test is one that uncovers an as-yet-undiscovered error.

 **ADVICE**  
These objectives imply a dramatic change in viewpoint for some software developers. They move counter to the commonly held view that a successful test is one in which no errors are found. Your objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

If testing is conducted successfully (according to the objectives stated previously), it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to specification, and that behavioral and performance requirements appear to have been met. In addition, the data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole. But testing cannot show the absence of errors and defects; it can show only that software errors and defects are present. It is important to keep this (rather gloomy) statement in mind as testing is being conducted.

*In a broader software design context, recall that you begin "in the large" by focusing on software architecture and end "in the small" focusing on components. For testing, you simply reverse the focus and test your way out.*

Davis [Dav95b] suggests a set of testing principles<sup>6</sup> that have been adapted for use in this book:

**Principle 1. All tests should be traceable to customer requirements.<sup>7</sup>**

The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

**Principle 2. Tests should be planned long before testing begins.** Test planning (Chapter 17) can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

**Principle 3. The Pareto principle applies to software testing.** In this context the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

**Principle 4. Testing should begin "in the small" and progress toward testing "in the large."** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

**Principle 5. Exhaustive testing is not possible.** The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

#### 4.3.5 Deployment Principles

As I noted earlier in Part 1 of this book, the deployment activity encompasses three actions: delivery, support, and feedback. Because modern software process models are evolutionary or incremental in nature, deployment happens not once, but a number of times as software moves toward completion. Each delivery cycle provides the customer and end users with an operational software increment that provides usable functions and features. Each support cycle provides documentation and human assistance for all functions and features introduced during all deployment cycles to

---

<sup>6</sup> Only a small subset of Davis's testing principles are noted here. For more information, see [Dav95b].

<sup>7</sup> This principle refers to *functional tests*, i.e., tests that focus on requirements. *Structural tests* (tests that focus on architectural or logical detail) may not address specific requirements directly.

date. Each feedback cycle provides the software team with important guidance that results in modifications to the functions, features, and approach taken for the next increment.



*Be sure that your customer knows what to expect before a software increment is delivered. Otherwise, you can bet the customer will expect more than you deliver.*

The delivery of a software increment represents an important milestone for any software project. A number of key principles should be followed as the team prepares to deliver an increment:

**Principle 1. Customer expectations for the software must be managed.**

Too often, the customer expects more than the team has promised to deliver, and disappointment occurs immediately. This results in feedback that is not productive and ruins team morale. In her book on managing expectations, Naomi Karten [Kar94] states: "The starting point for managing expectations is to become more conscientious about what you communicate and how." She suggests that a software engineer must be careful about sending the customer conflicting messages (e.g., promising more than you can reasonably deliver in the time frame provided or delivering more than you promise for one software increment and then less than promised for the next).

**Principle 2. A complete delivery package should be assembled and tested.**

A CD-ROM or other media (including Web-based downloads) containing all executable software, support data files, support documents, and other relevant information should be assembled and thoroughly beta-tested with actual users. All installation scripts and other operational features should be thoroughly exercised in as many different computing configurations (i.e., hardware, operating systems, peripheral devices, networking arrangements) as possible.

**Principle 3. A support regime must be established before the software is delivered.**

An end user expects responsiveness and accurate information when a question or problem arises. If support is ad hoc, or worse, nonexistent, the customer will become dissatisfied immediately. Support should be planned, support materials should be prepared, and appropriate record-keeping mechanisms should be established so that the software team can conduct a categorical assessment of the kinds of support requested.

**Principle 4. Appropriate instructional materials must be provided to end users.**

The software team delivers more than the software itself. Appropriate training aids (if required) should be developed; troubleshooting guidelines should be provided, and when necessary, a "what's different about this software increment" description should be published.<sup>8</sup>

---

<sup>8</sup> During the communication activity, the software team should determine what types of help materials users want.

**Principle 5. Buggy software should be fixed first, delivered later.** Under time pressure, some software organizations deliver low-quality increments with a warning to the customer that bugs “will be fixed in the next release.” This is a mistake. There’s a saying in the software business: “Customers will forget you delivered a high-quality product a few days late, but they will never forget the problems that a low-quality product caused them. The software reminds them every day.”

The delivered software provides benefit for the end user, but it also provides useful feedback for the software team. As the increment is put into use, end users should be encouraged to comment on features and functions, ease of use, reliability, and any other characteristics that are appropriate.

## 4.4 SUMMARY

Software engineering practice encompasses principles, concepts, methods, and tools that software engineers apply throughout the software process. Every software engineering project is different. Yet, a set of generic principles apply to the process as a whole and to the practice of each framework activity regardless of the project or the product.

A set of core principles help in the application of a meaningful software process and the execution of effective software engineering methods. At the process level, core principles establish a philosophical foundation that guides a software team as it navigates through the software process. At the level of practice, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community.

Communication principles focus on the need to reduce noise and improve bandwidth as the conversation between developer and customer progresses. Both parties must collaborate for the best communication to occur.

Planning principles provide guidelines for constructing the best map for the journey to a completed system or product. The plan may be designed solely for a single software increment, or it may be defined for the entire project. Regardless, it must address what will be done, who will do it, and when the work will be completed.

Modeling encompasses both analysis and design, describing representations of the software that progressively become more detailed. The intent of the models is to solidify understanding of the work to be done and to provide technical guidance to those who will implement the software. Modeling principles serve as a foundation for the methods and notation that are used to create representations of the software.

Construction incorporates a coding and testing cycle in which source code for a component is generated and tested. Coding principles define generic actions that

should occur before code is written, while it is being created, and after it has been completed. Although there are many testing principles, only one is dominant: testing is a process of executing a program with the intent of finding an error.

Deployment occurs as each software increment is presented to the customer and encompasses delivery, support, and feedback. Key principles for delivery consider managing customer expectations and providing the customer with appropriate support information for the software. Support demands advance preparation. Feedback allows the customer to suggest changes that have business value and provide the developer with input for the next iterative software engineering cycle.

### PROBLEMS AND POINTS TO PONDER

- 4.1.** Since a focus on quality demands resources and time, is it possible to be agile and still maintain a quality focus?
- 4.2.** Of the eight core principles that guide process (discussed in Section 4.2.1), which do you believe is most important?
- 4.3.** Describe the concept of *separation of concerns* in your own words.
- 4.4.** An important communication principle states “prepare before you communicate.” How should this preparation manifest itself in the early work that you do? What work products might result as a consequence of early preparation?
- 4.5.** Do some research on “facilitation” for the communication activity (use the references provided or others) and prepare a set of guidelines that focus solely on facilitation.
- 4.6.** How does agile communication differ from traditional software engineering communication? How is it similar?
- 4.7.** Why is it necessary to “move on”?
- 4.8.** Do some research on “negotiation” for the communication activity and prepare a set of guidelines that focus solely on negotiation.
- 4.9.** Describe what *granularity* means in the context of a project schedule.
- 4.10.** Why are models important in software engineering work? Are they always necessary? Are there qualifiers to your answer about necessity?
- 4.11.** What three “domains” are considered during requirements modeling?
- 4.12.** Try to add one additional principle to those stated for coding in Section 4.3.4.
- 4.13.** What is a successful test?
- 4.14.** Do you agree or disagree with the following statement: “Since we deliver multiple increments to the customer, why should we be concerned about quality in the early increments—we can fix problems in later iterations.” Explain your answer.
- 4.15.** Why is feedback important to the software team?

### FURTHER READINGS AND INFORMATION SOURCES

Customer communication is a critically important activity in software engineering, yet few practitioners spend any time reading about it. Withall (*Software Requirements Patterns*, Microsoft Press, 2007) presents a variety of useful patterns that address communications problems. Sutliff

(*User-Centred Requirements Engineering*, Springer, 2002) focuses heavily on communications-related challenges. Books by Weigers (*Software Requirements*, 2d ed., Microsoft Press, 2003), Pardee (*To Satisfy and Delight Your Customer*, Dorset House, 1996), and Karten [Kar94] provide much insight into methods for effective customer interaction. Although their book does not focus on software, Hooks and Farry (*Customer Centered Products*, American Management Association, 2000) present useful generic guidelines for customer communication. Young (*Effective Requirements Practices*, Addison-Wesley, 2001) emphasizes a “joint team” of customers and developers who develop requirements collaboratively. Somerville and Kotonya (*Requirements Engineering: Processes and Techniques*, Wiley, 1998) discuss “elicitation” concepts and techniques and other requirements engineering principles.

Communication and planning concepts and principles are considered in many project management books. Useful project management offerings include books by Bechtold (*Essentials of Software Project Management*, 2d ed., Management Concepts, 2007), Wysocki (*Effective Project Management: Traditional, Adaptive, Extreme*, 4th ed., Wiley, 2006), Leach (*Lean Project Management: Eight Principles for Success*, BookSurge Publishing, 2006), Hughes (*Software Project Management*, McGraw-Hill, 2005), and Stellman and Greene (*Applied Software Project Management*, O'Reilly Media, Inc., 2005).

Davis [Dav95] has compiled an excellent collection of software engineering principles. In addition, virtually every book on software engineering contains a useful discussion of concepts and principles for analysis, design, and testing. Among the most widely used offerings (in addition to this book!) are:

- Abran, A., and J. Moore, *SWEBOk: Guide to the Software Engineering Body of Knowledge*, IEEE, 2002.
- Christensen, M., and R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.
- Jalote, P., *An Integrated Approach to Software Engineering*, Springer, 2006.
- Pfleeger, S., *Software Engineering: Theory and Practice*, 3d ed., Prentice-Hall, 2005.
- Schach, S., *Object-Oriented and Classical Software Engineering*, McGraw-Hill, 7th ed., 2006.
- Sommerville, I., *Software Engineering*, 8th ed., Addison-Wesley, 2006.

These books also present detailed discussion of modeling and construction principles.

Modeling principles are considered in many books dedicated to requirements analysis and/or software design. Books by Lieberman (*The Art of Software Modeling*, Auerbach, 2007), Rosenberg and Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Roques (*UML in Practice*, Wiley, 2004), Penker and Eriksson (*Business Modeling with UML: Business Patterns at Work*, Wiley, 2001) discuss modeling principles and methods.

Norman's (*The Design of Everyday Things*, Currency/Doubleday, 1990) is must reading for every software engineer who intends to do design work. Winograd and his colleagues (*Bringing Design to Software*, Addison-Wesley, 1996) have edited an excellent collection of essays that address practical issues for software design. Constantine and Lockwood (*Software for Use*, Addison-Wesley, 1999) present the concepts associated with “user centered design.” Tognazzini (*Tog on Software Design*, Addison-Wesley, 1995) presents a worthwhile philosophical discussion of the nature of design and the impact of decisions on quality and a team's ability to produce software that provides great value to its customer. Stahl and his colleagues (*Model-Driven Software Development: Technology, Engineering*, Wiley, 2006) discuss the principles of model-driven development.

Hundreds of books address one or more elements of the construction activity. Kernighan and Plauger [Ker78] have written a classic text on programming style, McConnell [McC93] presents pragmatic guidelines for practical software construction, Bentley [Ben99] suggests a wide variety of programming pearls, Knuth [Knu99] has written a classic three-volume series on the art of programming, and Hunt [Hun99] suggests pragmatic programming guidelines.

Myers and his colleagues (*The Art of Software Testing*, 2d ed., Wiley, 2004) have developed a major revision of his classic text and discuss many important testing principles. Books by Perry

(*Effective Methods for Software Testing*, 3d ed., Wiley, 2006), Whittaker (*How to Break Software*, Addison-Wesley, 2002), Kaner and his colleagues (*Lessons Learned in Software Testing*, Wiley, 2001), and Marick (*The Craft of Software Testing*, Prentice-Hall, 1997) each present important testing concepts and principles and much pragmatic guidance.

A wide variety of information sources on software engineering practice are available on the Internet. An up-to-date list of World Wide Web references that are relevant to software engineering practice can be found at the SEPA website: [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm).

# UNDERSTANDING REQUIREMENTS

## KEY CONCEPTS

analysis	
model .....	138
analysis	
patterns .....	142
collaboration ..	126
elaboration .....	122
elicitation .....	121
inception .....	121
negotiation .....	122
quality function	
deployment .....	131

**U**nderstanding the requirements of a problem is among the most difficult tasks that face a software engineer. When you first think about it, developing a clear understanding of requirements doesn't seem that hard. After all, doesn't the customer know what is required? Shouldn't the end users have a good understanding of the features and functions that will provide benefit? Surprisingly, in many instances the answer to these questions is "no." And even if customers and end-users are explicit in their needs, those needs will change throughout the project.

In the forward to a book by Ralph Young [You01] on effective requirements practices, I wrote:

It's your worst nightmare. A customer walks into your office, sits down, looks you straight in the eye, and says, "I know you think you understand what I said, but what you don't understand is what I said is not what I meant." Invariably, this happens late

## QUICK LOOK

**What is it?** Before you begin any technical work, it's a good idea to apply a set of requirements engineering tasks. These tasks lead to an understanding of what the business impact of the software will be, what the customer wants, and how end users will interact with the software.

**Who does it?** Software engineers (sometimes referred to as system engineers or "analysts" in the IT world) and other project stakeholders (managers, customers, end users) all participate in requirements engineering.

**Why is it important?** Designing and building an elegant computer program that solves the wrong problem serves no one's needs. That's why it's important to understand what the customer wants before you begin to design and build a computer-based system.

**What are the steps?** Requirements engineering begins with inception—a task that defines the scope and nature of the problem to be solved. It moves onwards to elicitation—a task that helps stakeholders define what is required, and then

elaboration—where basic requirements are refined and modified. As stakeholders define the problem, negotiation occurs—what are the priorities, what is essential, when is it required? Finally, the problem is specified in some manner and then reviewed or validated to ensure that your understanding of the problem and the stakeholders' understanding of the problem coincide.

**What is the work product?** The intent of requirements engineering is to provide all parties with a written understanding of the problem. This can be achieved through a number of work products: usage scenarios, functions and features lists, requirements models, or a specification.

**How do I ensure that I've done it right?** Requirements engineering work products are reviewed with stakeholders to ensure that what you have learned is what they really meant. A word of warning: even after all parties agree, things will change, and they will continue to change throughout the project.

requirements engineering	...120
requirements gathering	....128
requirements management	..124
specification	...122
stakeholders	..125
use cases	....133
validating requirements	..144
validation	....123
viewpoints	....126
work products	....133

in the project, after deadline commitments have been made, reputations are on the line, and serious money is at stake.

All of us who have worked in the systems and software business for more than a few years have lived this nightmare, and yet, few of us have learned to make it go away. We struggle when we try to elicit requirements from our customers. We have trouble understanding the information that we do acquire. We often record requirements in a disorganized manner, and we spend far too little time verifying what we do record. We allow change to control us, rather than establishing mechanisms to control change. In short, we fail to establish a solid foundation for the system or software. Each of these problems is challenging. When they are combined, the outlook is daunting for even the most experienced managers and practitioners. But solutions do exist.

It's reasonable to argue that the techniques I'll discuss in this chapter are not a true "solution" to the challenges just noted. But they do provide a solid approach for addressing these challenges.

## 5.1 REQUIREMENTS ENGINEERING

### note:

"The hardest single part of building a software system is deciding what to build. No part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."

Fred Brooks

### KEY POINT

Requirements engineering establishes a solid base for design and construction. Without it, the resulting software has a high probability of not meeting customer's needs.

Designing and building computer software is challenging, creative, and just plain fun. In fact, building software is so compelling that many software developers want to jump right in before they have a clear understanding of what is needed. They argue that things will become clear as they build, that project stakeholders will be able to understand need only after examining early iterations of the software, that things change so rapidly that any attempt to understand requirements in detail is a waste of time, that the bottom line is producing a working program and all else is secondary. What makes these arguments seductive is that they contain elements of truth.<sup>1</sup> But each is flawed and can lead to a failed software project.

The broad spectrum of tasks and techniques that lead to an understanding of requirements is called *requirements engineering*. From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. It must be adapted to the needs of the process, the project, the product, and the people doing the work.

Requirements engineering builds a bridge to design and construction. But where does the bridge originate? One could argue that it begins at the feet of the project stakeholders (e.g., managers, customers, end users), where business need is defined, user scenarios are described, functions and features are delineated, and project constraints are identified. Others might suggest that it begins with a broader system definition, where software is but one component of the larger system domain. But regardless of the starting point, the journey across the bridge takes you

<sup>1</sup> This is particularly true for small projects (less than one month) and smaller, relatively simple software efforts. As software grows in size and complexity, these arguments begin to break down.



*Expect to do a bit of design during requirements work and a bit of requirements work during design.*

high above the project, allowing you to examine the context of the software work to be performed; the specific needs that design and construction must address; the priorities that guide the order in which work is to be completed; and the information, functions, and behaviors that will have a profound impact on the resultant design.

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system [Tha97]. It encompasses seven distinct tasks: inception, elicitation, elaboration, negotiation, specification, validation, and management. It is important to note that some of these tasks occur in parallel and all are adapted to the needs of the project.

### quote:

*"The seeds of major software disasters are usually sown in the first three months of commencing the software project."*

**Caper Jones**

**Inception.** How does a software project get started? Is there a single event that becomes the catalyst for a new computer-based system or product, or does the need evolve over time? There are no definitive answers to these questions. In some cases, a casual conversation is all that is needed to precipitate a major software engineering effort. But in general, most projects begin when a business need is identified or a potential new market or service is discovered. Stakeholders from the business community (e.g., business managers, marketing people, product managers) define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's scope. All of this information is subject to change, but it is sufficient to precipitate discussions with the software engineering organization.<sup>2</sup>

At project inception,<sup>3</sup> you establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

**Elicitation.** It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. But it isn't simple—it's very hard.

Christel and Kang [Cri92] identify a number of problems that are encountered as elicitation occurs.

- **Problems of scope.** The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.

**?** Why is it difficult to gain a clear understanding of what the customer wants?

---

2 If a computer-based system is to be developed, discussions begin within the context of a system engineering process. For a detailed discussion of system engineering, visit the website that accompanies this book.

3 Recall that the Unified Process (Chapter 2) defines a more comprehensive "inception phase" that encompasses the inception, elicitation, and elaboration tasks discussed in this chapter.

- **Problems of understanding.** The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.
- **Problems of volatility.** The requirements change over time.

To help overcome these problems, you must approach requirements gathering in an organized manner.



*Elaboration is a good thing, but you have to know when to stop. The key is to describe the problem in a way that establishes a firm base for design. If you work beyond that point, you're doing design.*

**Elaboration.** The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model (Chapters 6 and 7) that identifies various aspects of software function, behavior, and information.

Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system. Each user scenario is parsed to extract analysis classes—business domain entities that are visible to the end user. The attributes of each analysis class are defined, and the services<sup>4</sup> that are required by each class are identified. The relationships and collaboration between classes are identified, and a variety of supplementary diagrams are produced.



*There should be no winner and no loser in an effective negotiation. Both sides win, because a "deal" that both can live with is solidified.*

**Negotiation.** It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources. It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs."

You have to reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

**Specification.** In the context of computer-based systems (and software), the term *specification* means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

Some suggest that a "standard template" [Som97] should be developed and used for a specification, arguing that this leads to requirements that are presented in a

---

<sup>4</sup> A service manipulates the data encapsulated by the class. The terms *operation* and *method* are also used. If you are unfamiliar with object-oriented concepts, a basic introduction is presented in Appendix 2.

## KEY POINT

The formality and format of a specification varies with the size and the complexity of the software to be built.

consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.



### Software Requirements Specification Template

**INFO**

A software requirements specification (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Karl Wiegers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at [www.processimpact.com/process\\_assets/srs\\_template.doc](http://www.processimpact.com/process_assets/srs_template.doc)) that can serve as a guideline for those who must create a complete SRS. A topic outline follows:

#### Table of Contents

#### Revision History

#### 1. Introduction

- 1.1 Purpose
- 1.2 Document Conventions
- 1.3 Intended Audience and Reading Suggestions
- 1.4 Project Scope
- 1.5 References

#### 2. Overall Description

- 2.1 Product Perspective

#### 2.2 Product Features

- 2.3 User Classes and Characteristics
- 2.4 Operating Environment
- 2.5 Design and Implementation Constraints
- 2.6 User Documentation
- 2.7 Assumptions and Dependencies

#### 3. System Features

- 3.1 System Feature 1
- 3.2 System Feature 2 (and so on)

#### 4. External Interface Requirements

- 4.1 User Interfaces
- 4.2 Hardware Interfaces
- 4.3 Software Interfaces
- 4.4 Communications Interfaces

#### 5. Other Nonfunctional Requirements

- 5.1 Performance Requirements
- 5.2 Safety Requirements
- 5.3 Security Requirements
- 5.4 Software Quality Attributes

#### 6. Other Requirements

##### Appendix A: Glossary

##### Appendix B: Analysis Models

##### Appendix C: Issues List

A detailed description of each SRS topic can be obtained by downloading the SRS template at the URL noted earlier in this sidebar.

**Validation.** The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification<sup>5</sup> to ensure that all software requirements have been

<sup>5</sup> Recall that the nature of the specification will vary with each project. In some cases, the “specification” is a collection of user scenarios and little else. In others, the specification may be a document that contains scenarios, models, and written descriptions.

stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.



*A key concern during requirements validation is consistency. Use the analysis model to ensure that requirements have been consistently stated.*

The primary requirements validation mechanism is the technical review (Chapter 15). The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements.

## INFO



### Requirements Validation Checklist

It is often useful to examine each requirement against a set of checklist questions. Here is a small subset of those that might be asked:

- Are requirements stated clearly? Can they be misinterpreted?
- Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?
- Is the requirement bounded in quantitative terms?
- What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?

- Does the requirement violate any system domain constraints?
- Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?
- Is the requirement traceable to any system model that has been created?
- Is the requirement traceable to overall system/product objectives?
- Is the specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
- Has an index for the specification been created?
- Have requirements associated with performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

**Requirements management.** Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.<sup>6</sup> Many of these activities are identical to the software configuration management (SCM) techniques discussed in Chapter 22.

<sup>6</sup> Formal requirements management is initiated only for large projects that have hundreds of identifiable requirements. For small projects, this requirements engineering action is considerably less formal.

## SOFTWARE TOOLS



### Requirements Engineering

**Objective:** Requirements engineering tools assist in requirements gathering, requirements modeling, requirements management, and requirements validation.

**Mechanics:** Tool mechanics vary. In general, requirements engineering tools build a variety of graphical (e.g., UML) models that depict the informational, functional, and behavioral aspects of a system. These models form the basis for all other activities in the software process.

#### Representative Tools:

A reasonably comprehensive (and up-to-date) listing of requirements engineering tools can be found at the Volvere Requirements resources site at [www.volere.co.uk/tools.htm](http://www.volere.co.uk/tools.htm). Requirements modeling tools are discussed in

Chapters 6 and 7. Tools noted below focus on requirement management.

*EasyRM*, developed by Cybernetic Intelligence GmbH ([www.easy-rm.com](http://www.easy-rm.com)), builds a project-specific dictionary/glossary that contains detailed requirements descriptions and attributes.

*Rational RequisitePro*, developed by Rational Software ([www-306.ibm.com/software/awdtools/reapro/](http://www-306.ibm.com/software/awdtools/reapro/)), allows users to build a requirements database; represent relationships among requirements; and organize, prioritize, and trace requirements.

Many additional requirements management tools can be found at the Volvere site noted earlier and at [www.jiludwig.com/Requirements\\_Management\\_Tools.html](http://www.jiludwig.com/Requirements_Management_Tools.html).

## 5.2 ESTABLISHING THE GROUNDWORK

In an ideal setting, stakeholders and software engineers work together on the same team.<sup>8</sup> In such cases, requirements engineering is simply a matter of conducting meaningful conversations with colleagues who are well-known members of the team. But reality is often quite different.

Customer(s) or end users may be located in a different city or country, may have only a vague idea of what is required, may have conflicting opinions about the system to be built, may have limited technical knowledge, and may have limited time to interact with the requirements engineer. None of these things are desirable, but all are fairly common, and you are often forced to work within the constraints imposed by this situation.

In the sections that follow, I discuss the steps required to establish the groundwork for an understanding of software requirements—to get the project started in a way that will keep it moving forward toward a successful solution.

### KEY POINT

A stakeholder is anyone who has a direct interest in or benefits from the system that is to be developed.

#### 5.2.1 Identifying Stakeholders

Sommerville and Sawyer [Som97] define a stakeholder as “anyone who benefits in a direct or indirect way from the system which is being developed.” I have already

<sup>7</sup> Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

<sup>8</sup> This approach is strongly recommended for projects that adopt an agile software development philosophy.

identified the usual suspects: business operations managers, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers, support and maintenance engineers, and others. Each stakeholder has a different view of the system, achieves different benefits when the system is successfully developed, and is open to different risks if the development effort should fail.

At inception, you should create a list of people who will contribute input as requirements are elicited (Section 5.3). The initial list will grow as stakeholders are contacted because every stakeholder will be asked: “Whom else do you think I should talk to?”

### 5.2.2 Recognizing Multiple Viewpoints



#### note:

“Put three stakeholders in a room and ask them what kind of system they want. You’re likely to get four or more different opinions.”

Author unknown

Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. For example, the marketing group is interested in functions and features that will excite the potential market, making the new system easy to sell. Business managers are interested in a feature set that can be built within budget and that will be ready to meet defined market windows. End users may want features that are familiar to them and that are easy to learn and use. Software engineers may be concerned with functions that are invisible to nontechnical stakeholders but that enable an infrastructure that supports more marketable functions and features. Support engineers may focus on the maintainability of the software.

Each of these constituencies (and others) will contribute information to the requirements engineering process. As information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another. You should categorize all stakeholder information (including inconsistent and conflicting requirements) in a way that will allow decision makers to choose an internally consistent set of requirements for the system.

### 5.2.3 Working toward Collaboration

If five stakeholders are involved in a software project, you may have five (or more) different opinions about the proper set of requirements. Throughout earlier chapters, I have noted that customers (and other stakeholders) must collaborate among themselves (avoiding petty turf battles) and with software engineering practitioners if a successful system is to result. But how is this collaboration accomplished?

The job of a requirements engineer is to identify areas of commonality (i.e., requirements on which all stakeholders agree) and areas of conflict or inconsistency (i.e., requirements that are desired by one stakeholder but conflict with the needs of another stakeholder). It is, of course, the latter category that presents a challenge.

**INFO****Using "Priority Points"**

One way of resolving conflicting requirements and at the same time better understanding the relative importance of all requirements is to use a "voting" scheme based on *priority points*. All stakeholders are provided with some number of priority points that can be "spent" on any number of requirements. A list of requirements is presented, and each stakeholder indicates the relative importance of

each (from his or her viewpoint) by spending one or more priority points on it. Points spent cannot be reused. Once a stakeholder's priority points are exhausted, no further action on requirements can be taken by that person. Overall points spent on each requirement by all stakeholders provide an indication of the overall importance of each requirement.

Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion" (e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

#### **5.2.4 Asking the First Questions**

Questions asked at the inception of the project should be "context free" [Gau89]. The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:

**Q**uote:

"It is better to know some of the questions than all of the answers."

James Thurber

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

The next set of questions enables you to gain a better understanding of the problem and allows the customer to voice his or her perceptions about a solution:

What  
questions  
will help you gain  
a preliminary  
understanding of  
the problem?

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication activity itself. Gause and Weinberg [Gau89] call these “meta-questions” and propose the following (abbreviated) list:

- Are you the right person to answer these questions? Are your answers “official”?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

 **quote:**  
“He who asks a question is a fool for five minutes; he who does not ask a question is a fool forever.”

**Chinese proverb**

These questions (and others) will help to “break the ice” and initiate the communication that is essential to successful elicitation. But a question-and-answer meeting format is not an approach that has been overwhelmingly successful. In fact, the Q&A session should be used for the first encounter only and then replaced by a requirements elicitation format that combines elements of problem solving, negotiation, and specification. An approach of this type is presented in Section 5.3.

## 5.3 ELICITING REQUIREMENTS

Requirements elicitation (also called *requirements gathering*) combines elements of problem solving, elaboration, negotiation, and specification. In order to encourage a collaborative, team-oriented approach to requirements gathering, stakeholders work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements [Zah90].<sup>9</sup>

### 5.3.1 Collaborative Requirements Gathering

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

 **What are the basic guidelines for conducting a collaborative requirements gathering meeting?**

- Meetings are conducted and attended by both software engineers and other stakeholders.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A “facilitator” (can be a customer, a developer, or an outsider) controls the meeting.
- A “definition mechanism” (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

<sup>9</sup> This approach is sometimes called a *facilitated application specification technique* (FAST).

**note:**

"We spend a lot of time—the majority of project effort—not implementing or testing, but trying to decide what to build."

Brian Lawrence

**WebRef**

*Joint Application Development (JAD)* is a popular technique for requirements gathering. A good description can be found at [www.carolla.com/  
wp-jad.htm](http://www.carolla.com/wp-jad.htm).



If a system or product will serve many users, be absolutely certain that requirements are elicited from a representative cross section of users. If only one user defines all requirements, acceptance risk is high.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal. To better understand the flow of events as they occur, I present a brief scenario that outlines the sequence of events that lead up to the requirements gathering meeting, occur during the meeting, and follow the meeting.

During inception (Section 5.2) basic questions and answers establish the scope of the problem and the overall perception of a solution. Out of these initial meetings, the developer and customers write a one- or two-page "product request."

A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate. The product request is distributed to all attendees before the meeting date.

As an example,<sup>10</sup> consider an excerpt from a product request written by a marketing person involved in the *SafeHome* project. This person writes the following narrative about the home security function that is to be part of *SafeHome*:

Our research indicates that the market for home management systems is growing at a rate of 40 percent per year. The first *SafeHome* function we bring to market should be the home security function. Most people are familiar with "alarm systems" so this would be an easy sell.

The home security function would protect against and/or recognize a variety of undesirable "situations" such as illegal entry, fire, flooding, carbon monoxide levels, and others. It'll use our wireless sensors to detect each situation. It can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected.

In reality, others would contribute to this narrative during the requirements gathering meeting and considerably more information would be available. But even with additional information, ambiguity would be present, omissions would likely exist, and errors might occur. For now, the preceding "functional description" will suffice.

While reviewing the product request in the days before the meeting, each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects. Finally, lists of constraints (e.g., cost, size, business rules) and performance criteria (e.g., speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system.

<sup>10</sup> This example (with extensions and variations) is used to illustrate important software engineering methods in many of the chapters that follow. As an exercise, it would be worthwhile to conduct your own requirements gathering meeting and develop a set of lists for it.

**note:**

"Facts do not cease to exist because they are ignored."

Aldous Huxley



*Avoid the impulse to shoot down a customer's idea as "too costly" or "impractical." The idea here is to negotiate a list that is acceptable to all. To do this, you must keep an open mind.*

Objects described for *SafeHome* might include the control panel, smoke detectors, window and door sensors, motion detectors, an alarm, an event (a sensor has been activated), a display, a PC, telephone numbers, a telephone call, and so on. The list of services might include *configuring* the system, *setting* the alarm, *monitoring* the sensors, *dialing* the phone, *programming* the control panel, and *reading* the display (note that services act on objects). In a similar fashion, each attendee will develop lists of constraints (e.g., the system must recognize when sensors are not operating, must be user-friendly, must interface directly to a standard phone line) and performance criteria (e.g., a sensor event should be recognized within one second, and an event priority scheme should be implemented).

The lists of objects can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive-backed sheets, or written on a wall board. Alternatively, the lists may have been posted on an electronic bulletin board, at an internal website, or posed in a chat room environment for review prior to the meeting. Ideally, each listed entry should be capable of being manipulated separately so that lists can be combined, entries can be modified, and additions can be made. At this stage, critique and debate are strictly prohibited.

After individual lists are presented in one topic area, the group creates a combined list by eliminating redundant entries, adding any new ideas that come up during the discussion, but not deleting anything. After you create combined lists for all topic areas, discussion—coordinated by the facilitator—ensues. The combined list is shortened, lengthened, or reworded to properly reflect the product/system to be developed. The objective is to develop a consensus list of objects, services, constraints, and performance for the system to be built.

In many cases, an object or service described on a list will require further explanation. To accomplish this, stakeholders develop *mini-specifications* for entries on the lists.<sup>11</sup> Each mini-specification is an elaboration of an object or service. For example, the mini-spec for the *SafeHome* object **Control Panel** might be:

The control panel is a wall-mounted unit that is approximately 9 × 5 inches in size. The control panel has wireless connectivity to sensors and a PC. User interaction occurs through a keypad containing 12 keys. A 3 × 3 inch LCD color display provides user feedback. Software provides interactive prompts, echo, and similar functions.

The mini-specs are presented to all stakeholders for discussion. Additions, deletions, and further elaboration are made. In some cases, the development of mini-specs will uncover new objects, services, constraints, or performance requirements that will be added to the original lists. During all discussions, the team may raise an issue that cannot be resolved during the meeting. An *issues list* is maintained so that these ideas will be acted on later.

---

<sup>11</sup> Rather than creating a mini-specification, many software teams elect to develop user scenarios called *use cases*. These are considered in detail in Section 5.4 and in Chapter 6.

## SAFEHOME



### Conducting a Requirements Gathering Meeting

**The scene:** A meeting room. The first requirements gathering meeting is in progress.

**The players:** Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

#### The conversation:

**Facilitator (pointing at whiteboard):** So that's the current list of objects and services for the home security function.

**Marketing person:** That about covers it from our point of view.

**Vinod:** Didn't someone mention that they wanted all SafeHome functionality to be accessible via the Internet? That would include the home security function, no?

**Marketing person:** Yes, that's right . . . we'll have to add that functionality and the appropriate objects.

**Facilitator:** Does that also add some constraints?

**Jamie:** It does, both technical and legal.

**Production rep:** Meaning?

**Jamie:** We better make sure an outsider can't hack into the system, disarm it, and rob the place or worse. Heavy liability on our part.

**Doug:** Very true.

**Marketing:** But we still need that . . . just be sure to stop an outsider from getting in.

**Ed:** That's easier said than done and . . .

**Facilitator (interrupting):** I don't want to debate this issue now. Let's note it as an action item and proceed.

(Doug, serving as the recorder for the meeting, makes an appropriate note.)

**Facilitator:** I have a feeling there's still more to consider here.

(The group spends the next 20 minutes refining and expanding the details of the home security function.)

### 5.3.2 Quality Function Deployment



QFD defines requirements in a way that maximizes customer satisfaction.



Everyone wants to implement lots of exciting requirements, but be careful. That's how "requirements creep" sets in. On the other hand, exciting requirements lead to a breakthrough product!

*Quality function deployment* (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD "concentrates on maximizing customer satisfaction from the software engineering process" [Zul92]. To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process. QFD identifies three types of requirements [Zul92]:

**Normal requirements.** The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

**Expected requirements.** These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

**Exciting requirements.** These features go beyond the customer's expectations and prove to be very satisfying when present. For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product.

### WebRef

Useful information on QFD can be obtained at [www.qfdi.org](http://www.qfdi.org).

Although QFD concepts can be applied across the entire software process [Par96a], specific QFD techniques are applicable to the requirements elicitation activity. QFD uses customer interviews and observation, surveys, and examination of historical data (e.g., problem reports) as raw data for the requirements gathering activity. These data are then translated into a table of requirements—called the *customer voice table*—that is reviewed with the customer and other stakeholders. A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements [Aka04].

### 5.3.3 Usage Scenarios

As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called *use cases* [Jac92], provide a description of how the system will be used. Use cases are discussed in greater detail in Section 5.4.

## SAFEHOME



### Developing a Preliminary User Scenario

**The scene:** A meeting room, continuing the first requirements gathering meeting.

**The players:** Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

#### The conversation:

**Facilitator:** We've been talking about security for access to *SafeHome* functionality that will be accessible via the Internet. I'd like to try something. Let's develop a usage scenario for access to the home security function.

**Jamie:** How?

**Facilitator:** We can do it a couple of different ways, but for now, I'd like to keep things really informal. Tell us (he points at a marketing person) how you envision accessing the system.

**Marketing person:** Um . . . well, this is the kind of thing I'd do if I was away from home and I had to let someone into the house, say a housekeeper or repair guy, who didn't have the security code.

**Facilitator (smiling):** That's the reason you'd do it . . . tell me how you'd actually do this.

**Marketing person:** Um . . . the first thing I'd need is a PC. I'd log on to a website we'd maintain for all users of *SafeHome*. I'd provide my user id and . . .

**Vinod (interrupting):** The Web page would have to be secure, encrypted, to guarantee that we're safe and . . .

**Facilitator (interrupting):** That's good information, Vinod, but it's technical. Let's just focus on how the end user will use this capability. OK?

**Vinod:** No problem.

**Marketing person:** So as I was saying, I'd log on to a website and provide my user ID and two levels of passwords.

**Jamie:** What if I forget my password?

**Facilitator (interrupting):** Good point, Jamie, but let's not address that now. We'll make a note of that and call it an exception. I'm sure there'll be others.

**Marketing person:** After I enter the passwords, a screen representing all *SafeHome* functions will appear. I'd select the home security function. The system might request that I verify who I am, say, by asking for my address or phone number or something. It would then display a picture of the security system control panel

along with a list of functions that I can perform—arm the system, disarm the system, disarm one or more sensors. I suppose it might also allow me to reconfigure security zones and other things like that, but I'm not sure.

(As the marketing person continues talking, Doug takes copious notes; these form the basis for the first informal usage scenario. Alternatively, the marketing person could have been asked to write the scenario, but this would be done outside the meeting.)

### 5.3.4 Elicitation Work Products

The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include

What information is produced as a consequence of requirements gathering?

- A statement of need and feasibility.
- A bounded statement of scope for the system or product.
- A list of customers, users, and other stakeholders who participated in requirements elicitation.
- A description of the system's technical environment.
- A list of requirements (preferably organized by function) and the domain constraints that apply to each.
- A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
- Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in requirements elicitation.

## 5.4 DEVELOPING USE CASES

In a book that discusses how to write effective use cases, Alistair Cockburn [Coc01b] notes that “a use case captures a contract … [that] describes the system’s behavior under various conditions as the system responds to a request from one of its stakeholders . . .” In essence, a use case tells a stylized story about how an end user (playing one of a number of possible roles) interacts with the system under a specific set of circumstances. The story may be narrative text, an outline of tasks or interactions, a template-based description, or a diagrammatic representation. Regardless of its form, a use case depicts the software or system from the end user’s point of view.



Use cases are defined from an actor's point of view. An actor is a role that people (users) or devices play as they interact with the software.

The first step in writing a use case is to define the set of “actors” that will be involved in the story. *Actors* are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described. Actors represent the roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself. Every actor has one or more goals when using the system.

It is important to note that an actor and an end user are not necessarily the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case. As an example, consider a machine operator (a user) who interacts with the control computer for a manufacturing cell that contains a number of robots and numerically controlled machines. After careful review of requirements, the software for the control computer requires four different modes (roles) for interaction: programming mode, test mode, monitoring mode, and troubleshooting mode. Therefore, four actors can be defined: programmer, tester, monitor, and troubleshooter. In some cases, the machine operator can play all of these roles. In others, different people may play the role of each actor.

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors [Jac92] during the first iteration and secondary actors as more is learned about the system. *Primary actors* interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software. *Secondary actors* support the system so that primary actors can do their work.

Once actors have been identified, use cases can be developed. Jacobson [Jac92] suggests a number of questions<sup>12</sup> that should be answered by a use case:

### What do I need to know in order to develop an effective use case?

- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

---

<sup>12</sup> Jacobson's questions have been extended to provide a more complete view of use-case content.

Recalling basic *SafeHome* requirements, we define four actors: **homeowner** (a user), **setup manager** (likely the same person as **homeowner**, but playing a different role), **sensors** (devices attached to the system), and the **monitoring and response subsystem** (the central station that monitors the *SafeHome* home security function). For the purposes of this example, we consider only the **homeowner** actor. The **homeowner** actor interacts with the home security function in a number of different ways using either the alarm control panel or a PC:

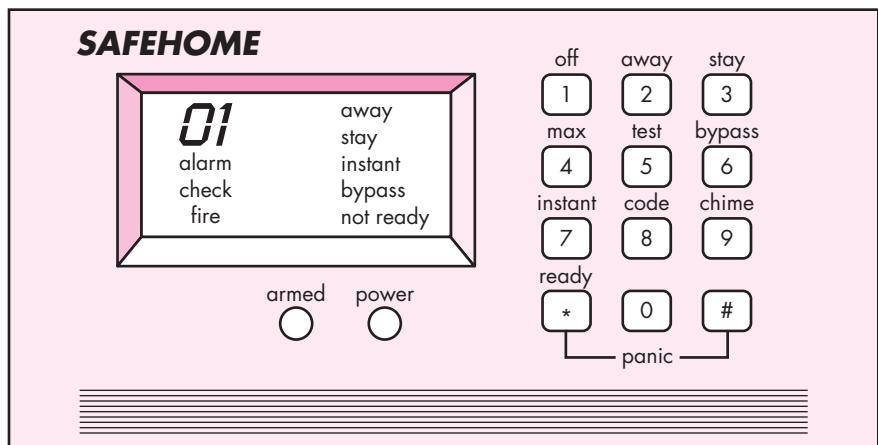
- Enters a password to allow all other interactions.
- Inquires about the status of a security zone.
- Inquires about the status of a sensor.
- Presses the panic button in an emergency.
- Activates/deactivates the security system.

Considering the situation in which the homeowner uses the control panel, the basic use case for system activation follows:<sup>13</sup>

1. The homeowner observes the *SafeHome* control panel (Figure 5.1) to determine if the system is ready for input. If the system is not ready, a *not ready* message is displayed on the LCD display, and the homeowner must physically close windows or doors so that the *not ready* message disappears. [A *not ready* message implies that a sensor is open; i.e., that a door or window is open.]

**FIGURE 5.1**

*SafeHome*  
control panel



<sup>13</sup> Note that this use case differs from the situation in which the system is accessed via the Internet. In this case, interaction occurs via the control panel, not the graphical user interface (GUI) provided when a PC is used.

2. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.
3. The homeowner selects and keys in *stay* or *away* (see Figure 5.1) to activate the system. *Stay* activates only perimeter sensors (inside motion detecting sensors are deactivated). *Away* activates all sensors.
4. When activation occurs, a red alarm light can be observed by the homeowner.

The basic use case presents a high-level story that describes the interaction between the actor and the system.



*Use cases are often written informally. However, use the template shown here to ensure that you've addressed all key issues.*

In many instances, use cases are further elaborated to provide considerably more detail about the interaction. For example, Cockburn [Coc01b] suggests the following template for detailed descriptions of use cases:

<b>Use case:</b>	<i>InitiateMonitoring</i>
<b>Primary actor:</b>	Homeowner.
<b>Goal in context:</b>	To set the system to monitor sensors when the homeowner leaves the house or remains inside.
<b>Preconditions:</b>	System has been programmed for a password and to recognize various sensors.
<b>Trigger:</b>	The homeowner decides to "set" the system, i.e., to turn on the alarm functions.
<b>Scenario:</b>	<ol style="list-style-type: none"> <li>1. Homeowner: observes control panel</li> <li>2. Homeowner: enters password</li> <li>3. Homeowner: selects "stay" or "away"</li> <li>4. Homeowner: observes red alarm light to indicate that <i>SafeHome</i> has been armed</li> </ol>
<b>Exceptions:</b>	<ol style="list-style-type: none"> <li>1. Control panel is <i>not ready</i>: homeowner checks all sensors to determine which are open; closes them.</li> <li>2. Password is incorrect (control panel beeps once): homeowner reenters correct password.</li> <li>3. Password not recognized: monitoring and response subsystem must be contacted to reprogram password.</li> <li>4. <i>Stay</i> is selected: control panel beeps twice and a <i>stay</i> light is lit; perimeter sensors are activated.</li> <li>5. <i>Away</i> is selected: control panel beeps three times and an <i>away</i> light is lit; all sensors are activated.</li> </ol>
<b>Priority:</b>	Essential, must be implemented
<b>When available:</b>	First increment

**Frequency of use:** Many times per day

**Channel to actor:** Via control panel interface

**Secondary actors:** Support technician, sensors

#### Channels to secondary actors:

Support technician: phone line

Sensors: hardwired and radio frequency interfaces

#### Open issues:

1. Should there be a way to activate the system without the use of a password or with an abbreviated password?
2. Should the control panel display additional text messages?
3. How much time does the homeowner have to enter the password from the time the first key is pressed?
4. Is there a way to deactivate the system before it actually activates?

Use cases for other **homeowner** interactions would be developed in a similar manner.

It is important to review each use case with care. If some element of the interaction is ambiguous, it is likely that a review of the use case will indicate a problem.

## SAFEHOME



### Developing a High-Level Use-Case Diagram

**The scene:** A meeting room, continuing the requirements gathering meeting

**The players:** Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

#### The conversation:

**Facilitator:** We've spent a fair amount of time talking about *SafeHome* home security functionality. During the break I sketched a use case diagram to summarize the important scenarios that are part of this function. Take a look.

(All attendees look at Figure 5.2.)

**Jamie:** I'm just beginning to learn UML notation.<sup>14</sup> So the home security function is represented by the big box with the ovals inside it? And the ovals represent use cases that we've written in text?

**Facilitator:** Yep. And the stick figures represent actors—the people or things that interact with the system as described by the use case . . . oh, I use the labeled square to represent an actor that's not a person . . . in this case, sensors.

**Doug:** Is that legal in UML?

**Facilitator:** Legality isn't the issue. The point is to communicate information. I view the use of a humanlike stick figure for representing a device to be misleading. So I've adapted things a bit. I don't think it creates a problem.

**Vinod:** Okay, so we have use-case narratives for each of the ovals. Do we need to develop the more detailed template-based narratives I've read about?

**Facilitator:** Probably, but that can wait until we've considered other *SafeHome* functions.

**Marketing person:** Wait, I've been looking at this diagram and all of a sudden I realize we missed something.

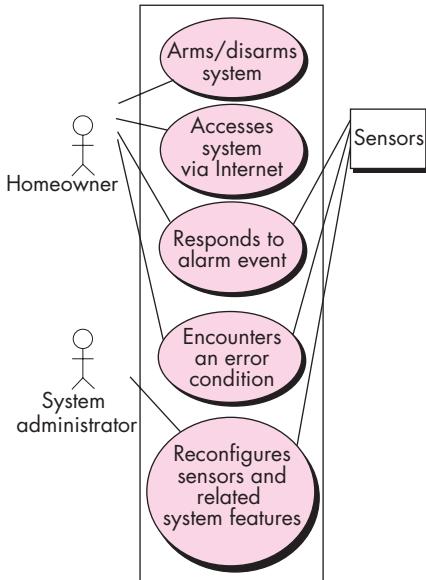
**Facilitator:** Oh really. Tell me what we've missed.

(The meeting continues.)

14 A brief UML tutorial is presented in Appendix 1 for those who are unfamiliar with the notation.

**FIGURE 5.2**

UML use case diagram for *SafeHome* home security function



### Use-Case Development

**Objective:** Assist in the development of use cases by providing automated templates and mechanisms for assessing clarity and consistency.

**Mechanics:** Tool mechanics vary. In general, use-case tools provide fill-in-the-blank templates for creating effective use cases. Most use-case functionality is embedded into a set of broader requirements engineering functions.

### SOFTWARE TOOLS

#### Representative Tools:<sup>15</sup>

The vast majority of UML-based analysis modeling tools provide both text and graphical support for use-case development and modeling.

#### Objects by Design

([www.objectsbydesign.com/tools/umltools\\_byCompany.html](http://www.objectsbydesign.com/tools/umltools_byCompany.html)) provides comprehensive links to tools of this type.

## 5.5 BUILDING THE REQUIREMENTS MODEL<sup>16</sup>

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as you learn more about the system to be built, and other stakeholders understand more about what they really require. For that reason, the analysis model is a snapshot of requirements at any given time. You should expect it to change.

<sup>15</sup> Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

<sup>16</sup> Throughout this book, I use the terms *analysis model* and *requirements model* synonymously. Both refer to representations of the information, functional, and behavioral domains that describe problem requirements.

As the requirements model evolves, certain elements will become relatively stable, providing a solid foundation for the design tasks that follow. However, other elements of the model may be more volatile, indicating that stakeholders do not yet fully understand requirements for the system. The analysis model and the methods that are used to build it are presented in detail in Chapters 6 and 7. I present a brief overview in the sections that follow.

### 5.5.1 Elements of the Requirements Model

There are many different ways to look at the requirements for a computer-based system. Some software people argue that it's best to select one mode of representation (e.g., the use case) and apply it to the exclusion of all other modes. Other practitioners believe that it's worthwhile to use a number of different modes of representation to depict the requirements model. Different modes of representation force you to consider requirements from different viewpoints—an approach that has a higher probability of uncovering omissions, inconsistencies, and ambiguity.

The specific elements of the requirements model are dictated by the analysis modeling method (Chapters 6 and 7) that is to be used. However, a set of generic elements is common to most requirements models.



*It is always a good idea to get stakeholders involved. One of the best ways to do this is to have each stakeholder write use cases that describe how the software will be used.*



*One way to isolate classes is to look for descriptive nouns in a use-case script. At least some of the nouns will be candidate classes. More on this in the Chapter 8.*

**Scenario-based elements.** The system is described from the user's point of view using a scenario-based approach. For example, basic use cases (Section 5.4) and their corresponding use-case diagrams (Figure 5.2) evolve into more elaborate template-based use cases. Scenario-based elements of the requirements model are often the first part of the model that is developed. As such, they serve as input for the creation of other modeling elements. Figure 5.3 depicts a UML activity diagram<sup>17</sup> for eliciting requirements and representing them using use cases. Three levels of elaboration are shown, culminating in a scenario-based representation.

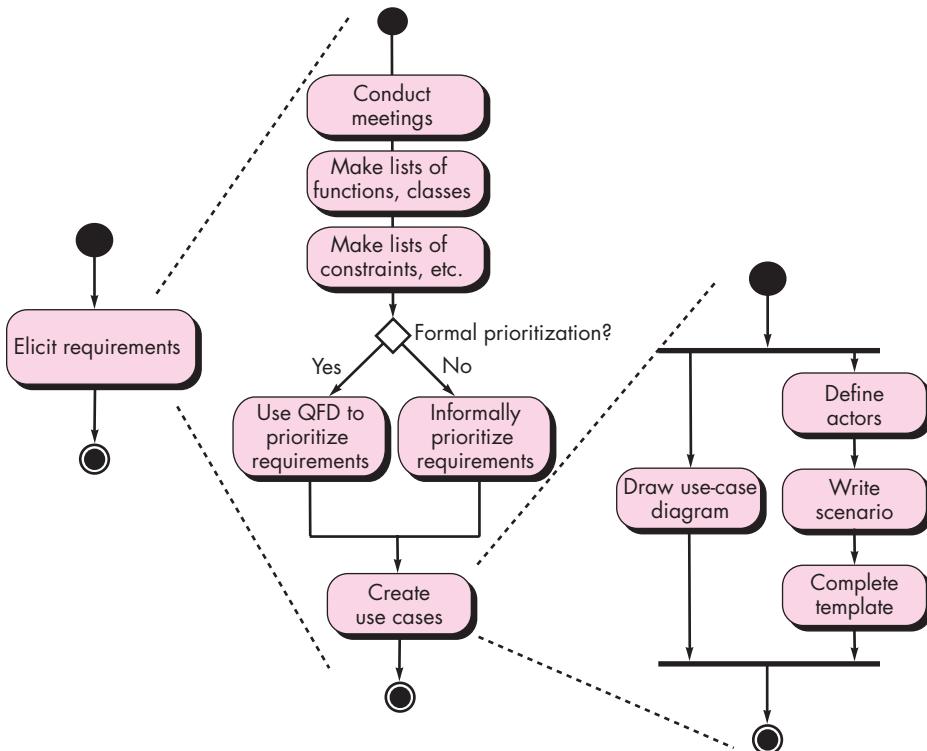
**Class-based elements.** Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors. For example, a UML class diagram can be used to depict a **Sensor** class for the *SafeHome* security function (Figure 5.4). Note that the diagram lists the attributes of sensors (e.g., name, type) and the operations (e.g., *identify*, *enable*) that can be applied to modify these attributes. In addition to class diagrams, other analysis modeling elements depict the manner in which classes collaborate with one another and the relationships and interactions between classes. These are discussed in more detail in Chapter 7.

**Behavioral elements.** The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior.

<sup>17</sup> A brief UML tutorial is presented in Appendix 1 for those who are unfamiliar with the notation.

**FIGURE 5.3**

UML activity diagrams for eliciting requirements

**FIGURE 5.4**

Class diagram for sensor



## KEY POINT

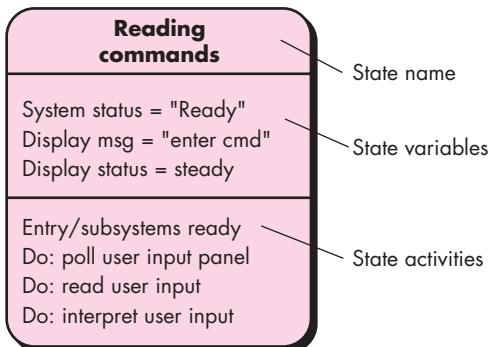
A state is an externally observable mode of behavior. External stimuli cause transitions between states.

The *state diagram* is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. A *state* is any externally observable mode of behavior. In addition, the state diagram indicates actions (e.g., process activation) taken as a consequence of a particular event.

To illustrate the use of a state diagram, consider software embedded within the *SafeHome* control panel that is responsible for reading user input. A simplified UML state diagram is shown in Figure 5.5.

**FIGURE 5.5**

UML state  
diagram  
notation



In addition to behavioral representations of the system as a whole, the behavior of individual classes can also be modeled. Further discussion of behavioral modeling is presented in Chapter 7.

## SAFEHOME



### Preliminary Behavioral Modeling

**The scene:** A meeting room, continuing the requirements meeting.

**The players:** Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

#### The conversation:

**Facilitator:** We've just about finished talking about SafeHome home security functionality. But before we do, I want to discuss the behavior of the function.

**Marketing person:** I don't understand what you mean by behavior.

**Ed (smiling):** That's when you give the product a "timeout" if it misbehaves.

**Facilitator:** Not exactly. Let me explain.

(The facilitator explains the basics of behavioral modeling to the requirements gathering team.)

**Marketing person:** This seems a little technical. I'm not sure I can help here.

**Facilitator:** Sure you can. What behavior do you observe from the user's point of view?

**Marketing person:** Uh . . . well, the system will be monitoring the sensors. It'll be reading commands from the homeowner. It'll be displaying its status.

**Facilitator:** See, you can do it.

**Jamie:** It'll also be polling the PC to determine if there is any input from it, for example, Internet-based access or configuration information.

**Vinod:** Yeah, in fact, configuring the system is a state in its own right.

**Doug:** You guys are rolling. Let's give this a bit more thought . . . is there a way to diagram this stuff?

**Facilitator:** There is, but let's postpone that until after the meeting.

**Flow-oriented elements.** Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms. Input may be a control signal transmitted by a transducer, a series of numbers typed by a human operator, a

packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. The transform(s) may comprise a single logical comparison, a complex numerical algorithm, or a rule-inference approach of an expert system. Output may light a single LED or produce a 200-page report. In effect, we can create a flow model for any computer-based system, regardless of size and complexity. A more detailed discussion of flow modeling is presented in Chapter 7.

### 5.5.2 Analysis Patterns

Anyone who has done requirements engineering on more than a few software projects begins to notice that certain problems reoccur across all projects within a specific application domain.<sup>18</sup> These *analysis patterns* [Fow97] suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

Geyer-Schulz and Hahsler [Gey01] suggest two benefits that can be associated with the use of analysis patterns:

First, analysis patterns speed up the development of abstract analysis models that capture the main requirements of the concrete problem by providing reusable analysis models with examples as well as a description of advantages and limitations. Second, analysis patterns facilitate the transformation of the analysis model into a design model by suggesting design patterns and reliable solutions for common problems.

Analysis patterns are integrated into the analysis model by reference to the pattern name. They are also stored in a repository so that requirements engineers can use search facilities to find and apply them. Information about an analysis pattern (and other types of patterns) is presented in a standard template [Gey01]<sup>19</sup> that is discussed in more detail in Chapter 12. Examples of analysis patterns and further discussion of this topic are presented in Chapter 7.

## 5.6 NEGOTIATING REQUIREMENTS

 **note:**

"A compromise is the art of dividing a cake in such a way that everyone believes he has the biggest piece."

**Ludwig Erhard**

In an ideal requirements engineering context, the inception, elicitation, and elaboration tasks determine customer requirements in sufficient detail to proceed to subsequent software engineering activities. Unfortunately, this rarely happens. In reality, you may have to enter into a negotiation with one or more stakeholders. In most cases, stakeholders are asked to balance functionality, performance, and other product or system characteristics against cost and time-to-market. The intent of this negotiation is to develop a project plan that meets stakeholder needs while at the

<sup>18</sup> In some cases, problems reoccur regardless of the application domain. For example, the features and functions used to solve user interface problems are common regardless of the application domain under consideration.

<sup>19</sup> A variety of patterns templates have been proposed in the literature. If you have interest, see [Fow97], [Gam95], [Yac03], and [Bus07] among many sources.

same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team.

### WebRef

A brief paper on negotiation for software requirements can be downloaded from [www.alexander-egyed.com/publications/Software\\_Requirements\\_Negotiation-Some\\_Lessons\\_Learned.html](http://www.alexander-egyed.com/publications/Software_Requirements_Negotiation-Some_Lessons_Learned.html).

The best negotiations strive for a “win-win” result.<sup>20</sup> That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you (as a member of the software team) win by working to realistic and achievable budgets and deadlines.

Boehm [Boe98] defines a set of negotiation activities at the beginning of each software process iteration. Rather than a single customer communication activity, the following activities are defined:

1. Identification of the system or subsystem’s key stakeholders.
2. Determination of the stakeholders’ “win conditions.”
3. Negotiation of the stakeholders’ win conditions to reconcile them into a set of win-win conditions for all concerned (including the software team).

Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to subsequent software engineering activities.

### INFO



#### *The Art of Negotiation*

Learning how to negotiate effectively can serve you well throughout your personal and technical life. The following guidelines are well worth considering:

1. *Recognize that it's not a competition.* To be successful, both parties have to feel they've won or achieved something. Both will have to compromise.
2. *Map out a strategy.* Decide what you'd like to achieve; what the other party wants to achieve, and how you'll go about making both happen.
3. *Listen actively.* Don't work on formulating your response while the other party is talking. Listen

to her. It's likely you'll gain knowledge that will help you to better negotiate your position.

4. *Focus on the other party's interests.* Don't take hard positions if you want to avoid conflict.
5. *Don't let it get personal.* Focus on the problem that needs to be solved.
6. *Be creative.* Don't be afraid to think out of the box if you're at an impasse.
7. *Be ready to commit.* Once an agreement has been reached, don't waffle; commit to it and move on.

### SAFEHOME



#### *The Start of a Negotiation*

**The scene:** Lisa Perez's office, after the first requirements gathering meeting.

**The players:** Doug Miller, software engineering manager and Lisa Perez, marketing manager.

#### **The conversation:**

**Lisa:** So, I hear the first meeting went really well.

**Doug:** Actually, it did. You sent some good people to the meeting . . . they really contributed.

<sup>20</sup> Dozens of books have been written on negotiating skills (e.g., [Lew06], [Rai06], [Fis06]). It is one of the more important skills that you can learn. Read one.

**Lisa (smiling):** Yeah, they actually told me they got into it and it wasn't a "propeller head activity."

**Doug (laughing):** I'll be sure to take off my techie beanie the next time I visit . . . Look, Lisa, I think we may have a problem with getting all of the functionality for the home security system out by the dates your management is talking about. It's early, I know, but I've already been doing a little back-of-the-envelope planning and . . .

**Lisa (frowning):** We've got to have it by that date, Doug. What functionality are you talking about?

**Doug:** I figure we can get full home security functionality out by the drop-dead date, but we'll have to delay Internet access 'til the second release.

**Lisa:** Doug, it's the Internet access that gives *SafeHome* "gee whiz" appeal. We're going to build our entire marketing campaign around it. We've gotta have it!

**Doug:** I understand your situation, I really do. The problem is that in order to give you Internet access, we'll have to have a fully secure website up and running. That takes time and people. We'll also have to build a lot of additional functionality into the first release . . . I don't think we can do it with the resources we've got.

**Lisa (still frowning):** I see, but you've got to figure out a way to get it done. It's pivotal to home security functions and to other functions as well . . . those can wait until the next releases . . . I'll agree to that.

Lisa and Doug appear to be at an impasse, and yet they must negotiate a solution to this problem. Can they both "win" here? Playing the role of a mediator, what would you suggest?

## 5.7 VALIDATING REQUIREMENTS

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. The requirements represented by the model are prioritized by the stakeholders and grouped within requirements packages that will be implemented as software increments. A review of the requirements model addresses the following questions:



- Is each requirement consistent with the overall objectives for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?

- Has the requirements model been “partitioned” in a way that exposes progressively more detailed information about the system?
- Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

These and other questions should be asked and answered to ensure that the requirements model is an accurate reflection of stakeholder needs and that it provides a solid foundation for design.

## 5.8 SUMMARY

Requirements engineering tasks are conducted to establish a solid foundation for design and construction. Requirements engineering occurs during the communication and modeling activities that have been defined for the generic software process. Seven distinct requirements engineering functions—inception, elicitation, elaboration, negotiation, specification, validation, and management—are conducted by members of the software team.

At project inception, stakeholders establish basic problem requirements, define overriding project constraints, and address major features and functions that must be present for the system to meet its objectives. This information is refined and expanded during elicitation—a requirements gathering activity that makes use of facilitated meetings, QFD, and the development of usage scenarios.

Elaboration further expands requirements in a model—a collection of scenario-based, class-based, behavioral, and flow-oriented elements. The model may reference analysis patterns, solutions for analysis problems that have been seen to reoccur across different applications.

As requirements are identified and the requirements model is being created, the software team and other project stakeholders negotiate the priority, availability, and relative cost of each requirement. The intent of this negotiation is to develop a realistic project plan. In addition, each requirement and the requirements model as a whole are validated against customer need to ensure that the right system is to be built.

## PROBLEMS AND POINTS TO PONDER

- 5.1.** Why is it that many software developers don’t pay enough attention to requirements engineering? Are there ever circumstances where you can skip it?
- 5.2.** You have been given the responsibility to elicit requirements from a customer who tells you he is too busy to meet with you. What should you do?
- 5.3.** Discuss some of the problems that occur when requirements must be elicited from three or four different customers.
- 5.4.** Why do we say that the requirements model represents a snapshot of a system in time?

**5.5.** Let's assume that you've convinced the customer (you're a very good salesperson) to agree to every demand that you have as a developer. Does that make you a master negotiator? Why?

**5.6.** Develop at least three additional "context-free questions" that you might ask a stakeholder during inception.

**5.7.** Develop a requirements gathering "kit." The kit should include a set of guidelines for conducting a requirements gathering meeting and materials that can be used to facilitate the creation of lists and any other items that might help in defining requirements.

**5.8.** Your instructor will divide the class into groups of four to six students. Half of the group will play the role of the marketing department and half will take on the role of software engineering. Your job is to define requirements for the *SafeHome* security function described in this chapter. Conduct a requirements gathering meeting using the guidelines presented in this chapter.

**5.9.** Develop a complete use case for one of the following activities:

- a. Making a withdrawal at an ATM
- b. Using your charge card for a meal at a restaurant
- c. Buying a stock using an on-line brokerage account
- d. Searching for books (on a specific topic) using an on-line bookstore
- e. An activity specified by your instructor.

**5.10.** What do use case "exceptions" represent?

**5.11.** Describe what an *analysis pattern* is in your own words.

**5.12.** Using the template presented in Section 5.5.2, suggest one or more analysis pattern for the following application domains:

- a. Accounting software
- b. E-mail software
- c. Internet browsers
- d. Word-processing software
- e. Website creation software
- f. An application domain specified by your instructor

**5.13.** What does *win-win* mean in the context of negotiation during the requirements engineering activity?

**5.14.** What do you think happens when requirement validation uncovers an error? Who is involved in correcting the error?

## FURTHER READINGS AND INFORMATION SOURCES

Because it is pivotal to the successful creation of any complex computer-based system, requirements engineering is discussed in a wide array of books. Hood and his colleagues (*Requirements Management*, Springer, 2007) discuss a variety of requirements engineering issues that span both systems and software engineering. Young (*The Requirements Engineering Handbook*, Artech House Publishers, 2007) presents an in-depth discussion of requirements engineering tasks. Wiegers (*More About Software Requirements*, Microsoft Press, 2006) provides many practical techniques for requirements gathering and management. Hull and her colleagues (*Requirements Engineering*, 2d ed., Springer-Verlag, 2004), Bray (*An Introduction to Requirements Engineering*, Addison-Wesley, 2002), Arlow (*Requirements Engineering*, Addison-Wesley, 2001), Gilb (*Requirements Engineering*, Addison-Wesley, 2000), Graham (*Requirements Engineering and Rapid Development*, Addison-Wesley, 1999), and Sommerville and Kotonya (*Requirement Engineering: Processes and Techniques*, Wiley, 1998) are but a few of many books dedicated to the subject. Gottesdiener (*Requirements by Collaboration: Workshops for Defining*

*Needs*, Addison-Wesley, 2002) provides useful guidance for those who must establish a collaborative requirements gathering environment with stakeholders.

Lauesen (*Software Requirements: Styles and Techniques*, Addison-Wesley, 2002) presents a comprehensive survey of requirement analysis methods and notation. Weigers (*Software Requirements*, Microsoft Press, 1999) and Leffingwell and his colleagues (*Managing Software Requirements: A Use Case Approach*, 2d ed., Addison-Wesley, 2003) present a useful collection of requirement best practices and suggest pragmatic guidelines for most aspects of the requirements engineering process.

A patterns-based view of requirements engineering is described by Withall (*Software Requirement Patterns*, Microsoft Press, 2007). Ploesch (*Assertions, Scenarios and Prototypes*, Springer-Verlag, 2003) discusses advanced techniques for developing software requirements. Windle and Abreo (*Software Requirements Using the Unified Process*, Prentice-Hall, 2002) discuss requirements engineering within the context of the Unified Process and UML notation. Alexander and Steven (*Writing Better Requirements*, Addison-Wesley, 2002) present a brief set of guidelines for writing clear requirements, representing them as scenarios, and reviewing the end result.

Use-case modeling is often the driver for the creation of all other aspects of the analysis model. The subject is discussed at length by Rosenberg and Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality*, Addison-Wesley, 2005), Alexander and Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*, Wiley, 2004), Leffingwell and his colleagues (*Managing Software Requirements: A Use Case Approach*, 2d ed., Addison-Wesley, 2003) present a useful collection of requirement best practices. Bittner and Spence (*Use Case Modeling*, Addison-Wesley, 2002), Cockburn [Coc01], Armour and Miller (*Advanced Use Case Modeling: Software Systems*, Addison-Wesley, 2000), and Kulak and his colleagues (*Use Cases: Requirements in Context*, Addison-Wesley, 2000) discuss requirements gathering with an emphasis on use-case modeling.

A wide variety of information sources on requirements engineering and analysis is available on the Internet. An up-to-date list of World Wide Web references that are relevant to requirements engineering and analysis can be found at the SEPA website: [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm).

# REQUIREMENTS MODELING: SCENARIOS, INFORMATION, AND ANALYSIS CLASSES

## KEY CONCEPTS

activity diagram .....	161
analysis classes .....	167
analysis packages .....	182
associations .....	180
class-based modeling .....	167
CRC modeling .....	173
data modeling .....	164
domain analysis .....	151
grammatical parse .....	167

## QUICK LOOK

**What is it?** The written word is a wonderful vehicle for communication, but it is not necessarily the best way to represent the requirements for computer software. Requirements modeling uses a combination of text and diagrammatic forms to depict requirements in a way that is relatively easy to understand, and more important, straightforward to review for correctness, completeness, and consistency.

**Who does it?** A software engineer (sometimes called an “analyst”) builds the model using requirements elicited from the customer.

**Why is it important?** To validate software requirements, you need to examine them from a number of different points of view. In this chapter you’ll consider requirements modeling from three different perspectives: scenario-based models, data (information) models, and class-based models. Each represents requirements in a different “dimension,” thereby increasing the probability that errors will be found, that inconsistency will surface, and that omissions will be uncovered.

**A**t a technical level, software engineering begins with a series of modeling tasks that lead to a specification of requirements and a design representation for the software to be built. The requirements model<sup>1</sup>—actually a set of models—is the first technical representation of a system.

In a seminal book on requirements modeling methods, Tom DeMarco [DeM79] describes the process in this way:

Looking back over the recognized problems and failings of the analysis phase, I suggest that we need to make the following additions to our set of analysis phase goals.

The products of analysis must be highly maintainable. This applies particularly to the

**What are the steps?** Scenario-based modeling represents the system from the user’s point of view. Data modeling represents the information space and depicts the data objects that the software will manipulate and the relationships among them. Class-based modeling defines objects, attributes, and relationships. Once preliminary models are created, they are refined and analyzed to assess their clarity, completeness, and consistency. In Chapter 7, we extend the modeling dimensions noted here with additional representations, providing a more robust view of requirements.

**What is the work product?** A wide array of text-based and diagrammatic forms may be chosen for the requirements model. Each of these representations provides a view of one or more of the model elements.

**How do I ensure that I’ve done it right?** Requirements modeling work products must be reviewed for correctness, completeness, and consistency. They must reflect the needs of all stakeholders and establish a foundation from which design can be conducted.

<sup>1</sup> In past editions of this book, I used the term *analysis model*, rather than *requirements model*. In this edition, I’ve decided to use both phrases to represent the modeling activity that defines various aspects of the problem to be solved. *Analysis* is the action that occurs as *requirements* are derived.

<b>requirements modeling</b>	.....153
<b>scenario-based modeling</b>	.....154
<b>swimlane diagram</b>	.....162
<b>UML models</b>	.....161
<b>use cases</b>	.....156

Target Document [software requirements specification]. Problems of size must be dealt with using an effective method of partitioning. The Victorian novel specification is out. Graphics have to be used whenever possible. We have to differentiate between logical [essential] and physical [implementation] considerations. . . At the very least, we need. . . Something to help us partition our requirements and document that partitioning before specification. . . Some means of keeping track of and evaluating interfaces. . . New tools to describe logic and policy, something better than narrative text.

Although DeMarco wrote about the attributes of analysis modeling more than a quarter century ago, his comments still apply to modern requirements modeling methods and notation.

## 6.1 REQUIREMENTS ANALYSIS

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you (regardless of whether you're called a *software engineer*, an *analyst*, or a *modeler*) to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering (Chapter 5).

**note:**

"Any one 'view' of requirements is insufficient to understand or describe the desired behavior of a complex system."

Alan M. Davis

The requirements modeling action results in one or more of the following types of models:

- *Scenario-based models* of requirements from the point of view of various system "actors"
- *Data models* that depict the information domain for the problem
- *Class-oriented models* that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements
- *Flow-oriented models* that represent the functional elements of the system and how they transform data as it moves through the system
- *Behavioral models* that depict how the software behaves as a consequence of external "events"

**KEY POINT**

The analysis model and requirements specification provide a means for assessing quality once the software is built.

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model (and the software requirements specification) provides the developer and the customer with the means to assess quality once software is built.

In this chapter, I focus on *scenario-based modeling*—a technique that is growing increasingly popular throughout the software engineering community; *data modeling*—a more specialized technique that is particularly appropriate when an application must create or manipulate a complex information space; and *class*

**FIGURE 6.1**

The requirements model as a bridge between the system description and the design model



*modeling*—a representation of the object-oriented classes and the resultant collaborations that allow a system to function. Flow-oriented models, behavioral models, pattern-based modeling, and WebApp models are discussed in Chapter 7.

### QUOTE

"Requirements are not architecture. Requirements are not design, nor are they the user interface. Requirements are need."

Andrew Hunt  
and David Thomas

### KEY POINT

The analysis model should describe what the customer wants, establish a basis for design, and establish a target for validation.

### 6.1.1 Overall Objectives and Philosophy

Throughout requirements modeling, your primary focus is on *what*, not *how*. What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined, and what constraints apply?<sup>2</sup>

In earlier chapters, I noted that complete specification of requirements may not be possible at this stage. The customer may be unsure of precisely what is required for certain aspects of the system. The developer may be unsure that a specific approach will properly accomplish function and performance. These realities mitigate in favor of an iterative approach to requirements analysis and modeling. The analyst should model what is known and use that model as the basis for design of the software increment.<sup>3</sup>

The requirements model must achieve three primary objectives: (1) to describe what the customer requires, (2) to establish a basis for the creation of a software design, and (3) to define a set of requirements that can be validated once the software is built. The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design (Chapters 8 through 13) that describes the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure 6.1.

- 2 It should be noted that as customers become more technologically sophisticated, there is a trend toward the specification of *how* as well as *what*. However, the primary focus should remain on *what*.
- 3 Alternatively, the software team may choose to create a prototype (Chapter 2) in an effort to better understand requirements for the system.

It is important to note that all elements of the requirements model will be directly traceable to parts of the design model. A clear division of analysis and design tasks between these two important modeling activities is not always possible. Some design invariably occurs as part of analysis, and some analysis will be conducted during design.

### 6.1.2 Analysis Rules of Thumb

Arlow and Neustadt [Arl02] suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

**?** Are there basic guidelines that can help us as we do requirements analysis work?

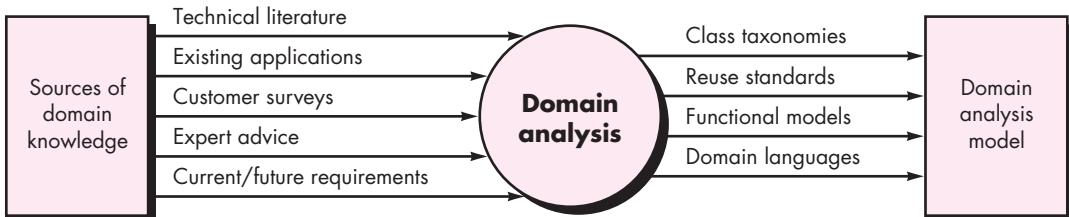
**quote:**  
"Problems worthy of attack, prove their worth by hitting back."  
**Piet Hein**

- *The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high. "Don't get bogged down in details"* [Arl02] that try to explain how the system will work.
- *Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.*
- *Delay consideration of infrastructure and other nonfunctional models until design.* That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.
- *Minimize coupling throughout the system.* It is important to represent relationships between classes and functions. However, if the level of "interconnect-edness" is extremely high, effort should be made to reduce it.
- *Be certain that the requirements model provides value to all stakeholders.* Each constituency has its own use for the model. For example, business stakeholders should use the model to validate requirements; designers should use the model as a basis for design; QA people should use the model to help plan acceptance tests.
- *Keep the model as simple as it can be.* Don't create additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

### 6.1.3 Domain Analysis

**WebRef**  
Many useful resources for domain analysis can be found at [www.iturls.com/English/Software\\_Engineering/SE\\_mod5.asp](http://www.iturls.com/English/Software_Engineering/SE_mod5.asp).

In the discussion of requirements engineering (Chapter 5), I noted that analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems, the creation of the analysis model is expedited. More important, the likelihood of applying design patterns and executable software components grows dramatically. This improves time-to-market and reduces development costs.

**FIGURE 6.2** Input and output for domain analysis

But how are analysis patterns and classes recognized in the first place? Who defines them, categorizes them, and readies them for use on subsequent projects? The answers to these questions lie in *domain analysis*. Firesmith [Fir93] describes domain analysis in the following way:

### KEY POINT

Domain analysis doesn't look at a specific application, but rather at the domain in which the application resides. The intent is to identify common problem solving elements that are applicable to all applications within the domain.

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain. . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks.

The “specific application domain” can range from avionics to banking, from multimedia video games to software embedded within medical devices. The goal of domain analysis is straightforward: to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.<sup>4</sup>

Using terminology that was introduced earlier in this book, domain analysis may be viewed as an umbrella activity for the software process. By this I mean that domain analysis is an ongoing software engineering activity that is not connected to any one software project. In a way, the role of a domain analyst is similar to the role of a master toolsmith in a heavy manufacturing environment. The job of the toolsmith is to design and build tools that may be used by many people doing similar but not necessarily the same jobs. The role of the domain analyst<sup>5</sup> is to discover and define analysis patterns, analysis classes, and related information that may be used by many people working on similar but not necessarily the same applications.

Figure 6.2 [Ara89] illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

<sup>4</sup> A complementary view of domain analysis “involves modeling the domain so that software engineers and other stakeholders can better learn about it . . . not all domain classes necessarily result in the development of reusable classes . . .” [Let03a].

<sup>5</sup> Do not make the assumption that because a domain analyst is at work, a software engineer need not understand the application domain. Every member of a software team should have some understanding of the domain in which the software is to be placed.

## SAFEHOME



### Domain Analysis

**The scene:** Doug Miller's office, after a meeting with marketing.

**The players:** Doug Miller, software engineering manager, and Vinod Raman, a member of the software engineering team.

#### The conversation:

**Doug:** I need you for a special project, Vinod. I'm going to pull you out of the requirements gathering meetings.

**Vinod (frowning):** Too bad. That format actually works . . . I was getting something out of it. What's up?

**Doug:** Jamie and Ed will cover for you. Anyway, marketing insists that we deliver the Internet capability along with the home security function in the first release of *SafeHome*. We're under the gun on this . . . not enough time or people, so we've got to solve both problems—the PC interface and the Web interface—at once.

**Vinod (looking confused):** I didn't know the plan was set . . . we're not even finished with requirements gathering.

**Doug (a wan smile):** I know, but the time lines are so short that I decided to begin strategizing with marketing right now . . . anyhow, we'll revisit any tentative plan once we have the info from all of the requirements gathering meetings.

**Vinod:** Okay, what's up? What do you want me to do?

**Doug:** Do you know what "domain analysis" is?

**Vinod:** Sort of. You look for similar patterns in Apps that do the same kinds of things as the App you're building. If possible, you then steal the patterns and reuse them in your work.

**Doug:** Not sure I like the word *steal*, but basically you have it right. What I'd like you to do is to begin researching existing user interfaces for systems that control something like *SafeHome*. I want you to propose a set of patterns and analysis classes that can be common to both the PC-based interface that'll sit in the house and the browser-based interface that is accessible via the Internet.

**Vinod:** We can save time by making them the same . . . why don't we just do that?

**Doug:** Ah . . . it's nice to have people who think like you do. That's the whole point—we can save time and effort if both interfaces are nearly identical, implemented with the same code, blah, blah, that marketing insists on.

**Vinod:** So you want, what—classes, analysis patterns, design patterns?

**Doug:** All of 'em. Nothing formal at this point. I just want to get a head start on our internal analysis and design work.

**Vinod:** I'll go to our class library and see what we've got. I'll also use a patterns template I saw in a book I was reading a few months back.

**Doug:** Good. Go to work.

#### note:

"...analysis is frustrating, full of complex interpersonal relationships, indefinite, and difficult. In a word, it is fascinating. Once you're hooked, the old easy pleasures of system building are never again enough to satisfy you."

Tom DeMarco

### 6.1.4 Requirements Modeling Approaches

One view of requirements modeling, called *structured analysis*, considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

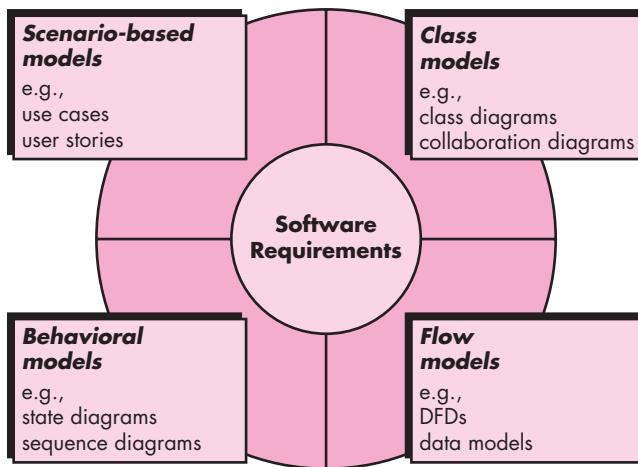
A second approach to analysis modeling, called *object-oriented analysis*, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process (Chapter 2) are predominantly object oriented.

Although the requirements model proposed in this book combines features of both approaches, software teams often choose one approach and exclude all representations from the other. The question is not which is best, but rather, what

**FIGURE 6.3**

Elements of the analysis model

What different points of view can be used to describe the requirements model?



### **quote:**

"Why should we build models? Why not just build the system itself? The answer is that we can construct models in such a way as to highlight, or emphasize, certain critical features of a system, while simultaneously de-emphasizing other aspects of the system."

**Ed Yourdon**

combination of representations will provide stakeholders with the best model of software requirements and the most effective bridge to software design.

Each element of the requirements model (Figure 6.3) presents the problem from a different point of view. Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used. Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally, flow-oriented elements represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

Analysis modeling leads to the derivation of each of these modeling elements. However, the specific content of each element (i.e., the diagrams that are used to construct the element and the model) may differ from project to project. As we have noted a number of times in this book, the software team must work to keep it simple. Only those modeling elements that add value to the model should be used.

## **6.2 SCENARIO-BASED MODELING**

Although the success of a computer-based system or product is measured in many ways, user satisfaction resides at the top of the list. If you understand how end users (and other actors) want to interact with a system, your software team will be better able to properly characterize requirements and build meaningful analysis and design

models. Hence, requirements modeling with UML<sup>6</sup> begins with the creation of scenarios in the form of use cases, activity diagrams, and swimlane diagrams.

### **Quote:**

“[Use cases] are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use cases).”

Ivar Jacobson



In some situations, use cases become the dominant requirements engineering mechanism. However, this does not mean that you should discard other modeling methods when they are appropriate.

### **6.2.1 Creating a Preliminary Use Case**

Alistair Cockburn characterizes a use case as a “contract for behavior” [Coc01b]. As we discussed in Chapter 5, the “contract” defines the way in which an actor<sup>7</sup> uses a computer-based system to accomplish some goal. In essence, a use case captures the interactions that occur between producers and consumers of information and the system itself. In this section, I examine how use cases are developed as part of the requirements modeling activity.<sup>8</sup>

In Chapter 5, I noted that a use case describes a specific usage scenario in straightforward language from the point of view of a defined actor. But how do you know (1) what to write about, (2) how much to write about it, (3) how detailed to make your description, and (4) how to organize the description? These are the questions that must be answered if use cases are to provide value as a requirements modeling tool.

**What to write about?** The first two requirements engineering tasks—inception and elicitation—provide you with the information you’ll need to begin writing use cases. Requirements gathering meetings, QFD, and other requirements engineering mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, establish priorities, outline all known functional requirements, and describe the things (objects) that will be manipulated by the system.

To begin developing a set of use cases, list the functions or activities performed by a specific actor. You can obtain these from a list of required system functions, through conversations with stakeholders, or by an evaluation of activity diagrams (Section 6.3.1) developed as part of requirements modeling.

## SAFEHOME



### **Developing Another Preliminary User Scenario**

**The scene:** A meeting room, during the second requirements gathering meeting.

**The players:** Jamie Lazar, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

### **The conversation:**

**Facilitator:** It’s time that we begin talking about the SafeHome surveillance function. Let’s develop a user scenario for access to the surveillance function.

**Jamie:** Who plays the role of the actor on this?

- 
- 6 UML will be used as the modeling notation throughout this book. Appendix 1 provides a brief tutorial for those readers who may be unfamiliar with basic UML notation.
  - 7 An actor is not a specific person, but rather a role that a person (or a device) plays within a specific context. An actor “calls on the system to deliver one of its services” [Coc01b].
  - 8 Use cases are a particularly important part of analysis modeling for user interfaces. Interface analysis is discussed in detail in Chapter 11.

**Facilitator:** I think Meredith (a marketing person) has been working on that functionality. Why don't you play the role?

**Meredith:** You want to do it the same way we did it last time, right?

**Facilitator:** Right . . . same way.

**Meredith:** Well, obviously the reason for surveillance is to allow the homeowner to check out the house while he or she is away, to record and play back video that is captured . . . that sort of thing.

**Ed:** Will we use compression to store the video?

**Facilitator:** Good question, Ed, but let's postpone implementation issues for now. Meredith?

**Meredith:** Okay, so basically there are two parts to the surveillance function . . . the first configures the system including laying out a floor plan—we have to have tools to help the homeowner do this—and the second part is the actual surveillance function itself. Since the layout is part of the configuration activity, I'll focus on the surveillance function.

**Facilitator (smiling):** Took the words right out of my mouth.

**Meredith:** Um . . . I want to gain access to the surveillance function either via the PC or via the Internet. My feeling is that the Internet access would be more frequently used. Anyway, I want to be able to display camera views on a PC and control pan and zoom for a specific camera. I specify the camera by selecting it from the house floor plan. I want to selectively record camera output and replay camera output. I also want to be able to block access to one or more cameras with a specific password. I also want the option of seeing small windows that show views from all cameras and then be able to pick the one I want enlarged.

**Jamie:** Those are called thumbnail views.

**Meredith:** Okay, then I want thumbnail views of all the cameras. I also want the interface for the surveillance function to have the same look and feel as all other *SafeHome* interfaces. I want it to be intuitive, meaning I don't want to have to read a manual to use it.

**Facilitator:** Good job. Now, let's go into this function in a bit more detail . . .

The *SafeHome* home surveillance function (subsystem) discussed in the sidebar identifies the following functions (an abbreviated list) that are performed by the **homeowner** actor:

- Select camera to view.
- Request thumbnails from all cameras.
- Display camera views in a PC window.
- Control pan and zoom for a specific camera.
- Selectively record camera output.
- Replay camera output.
- Access camera surveillance via the Internet.

As further conversations with the stakeholder (who plays the role of a homeowner) progress, the requirements gathering team develops use cases for each of the functions noted. In general, use cases are written first in an informal narrative fashion. If more formality is required, the same use case is rewritten using a structured format similar to the one proposed in Chapter 5 and reproduced later in this section as a sidebar.

To illustrate, consider the function *access camera surveillance via the Internet—display camera views (ACS-DCV)*. The stakeholder who takes on the role of the **homeowner** actor might write the following narrative:

**Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)**

**Actor: homeowner**

If I'm at a remote location, I can use any PC with appropriate browser software to log on to the *SafeHome Products* website. I enter my user ID and two levels of passwords and once I'm validated, I have access to all functionality for my installed *SafeHome* system. To access a specific camera view, I select "surveillance" from the major function buttons displayed. I then select "pick a camera" and the floor plan of the house is displayed. I then select the camera that I'm interested in. Alternatively, I can look at thumbnail snapshots from all cameras simultaneously by selecting "all cameras" as my viewing choice. Once I choose a camera, I select "view" and a one-frame-per-second view appears in a viewing window that is identified by the camera ID. If I want to switch cameras, I select "pick a camera" and the original viewing window disappears and the floor plan of the house is displayed again. I then select the camera that I'm interested in. A new viewing window appears.

A variation of a narrative use case presents the interaction as an ordered sequence of user actions. Each action is represented as a declarative sentence. Revisiting the **ACS-DCV** function, you would write:

**Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)**

**Actor: homeowner**

 **Quote:**

"Use cases can be used in many [software] processes. Our favorite is a process that is iterative and risk driven."

**Geri Schneider  
and Jason  
Winters**

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

It is important to note that this sequential presentation does not consider any alternative interactions (the narrative is more free-flowing and did represent a few alternatives). Use cases of this type are sometimes referred to as *primary scenarios* [Sch98a].

## 6.2.2 Refining a Preliminary Use Case

A description of alternative interactions is essential for a complete understanding of the function that is being described by a use case. Therefore, each step in the primary scenario is evaluated by asking the following questions [Sch98a]:



- *Can the actor take some other action at this point?*
- *Is it possible that the actor will encounter some error condition at this point? If so, what might it be?*
- *Is it possible that the actor will encounter some other behavior at this point (e.g., behavior that is invoked by some event outside the actor's control)? If so, what might it be?*

Answers to these questions result in the creation of a set of *secondary scenarios* that are part of the original use case but represent alternative behavior. For example, consider steps 6 and 7 in the primary scenario presented earlier:

6. The homeowner selects “pick a camera.”
7. The system displays the floor plan of the house.

*Can the actor take some other action at this point?* The answer is “yes.” Referring to the free-flowing narrative, the actor may choose to view thumbnail snapshots of all cameras simultaneously. Hence, one secondary scenario might be “View thumbnail snapshots for all cameras.”

*Is it possible that the actor will encounter some error condition at this point?* Any number of error conditions can occur as a computer-based system operates. In this context, we consider only error conditions that are likely as a direct result of the action described in step 6 or step 7. Again the answer to the question is “yes.” A floor plan with camera icons may have never been configured. Hence, selecting “pick a camera” results in an error condition: “No floor plan configured for this house.”<sup>9</sup> This error condition becomes a secondary scenario.

*Is it possible that the actor will encounter some other behavior at this point?* Again the answer to the question is “yes.” As steps 6 and 7 occur, the system may encounter an alarm condition. This would result in the system displaying a special alarm notification (type, location, system action) and providing the actor with a number of options relevant to the nature of the alarm. Because this secondary scenario can occur at any time for virtually all interactions, it will not become part of the **ACS-DCV** use case. Rather, a separate use case—**Alarm condition encountered**—would be developed and referenced from other use cases as required.

---

<sup>9</sup> In this case, another actor, the **system administrator**, would have to configure the floor plan, install and initialize (e.g., assign an equipment ID) all cameras, and test each camera to be certain that it is accessible via the system and through the floor plan.

Each of the situations described in the preceding paragraphs is characterized as a use-case exception. An *exception* describes a situation (either a failure condition or an alternative chosen by the actor) that causes the system to exhibit somewhat different behavior.

Cockburn [Coc01b] recommends using a “brainstorming” session to derive a reasonably complete set of exceptions for each use case. In addition to the three generic questions suggested earlier in this section, the following issues should also be explored:

- *Are there cases in which some “validation function” occurs during this use case?* This implies that validation function is invoked and a potential error condition might occur.
- *Are there cases in which a supporting function (or actor) will fail to respond appropriately?* For example, a user action awaits a response but the function that is to respond times out.
- *Can poor system performance result in unexpected or improper user actions?* For example, a Web-based interface responds too slowly, resulting in a user making multiple selects on a processing button. These selects queue inappropriately and ultimately generate an error condition.

The list of extensions developed as a consequence of asking and answering these questions should be “rationalized” [Co01b] using the following criteria: an exception should be noted within the use case if the software can detect the condition described and then handle the condition once it has been detected. In some cases, an exception will precipitate the development of another use case (to handle the condition noted).

### 6.2.3 Writing a Formal Use Case

The informal use cases presented in Section 6.2.1 are sometimes sufficient for requirements modeling. However, when a use case involves a critical activity or describes a complex set of steps with a significant number of exceptions, a more formal approach may be desirable.

The **ACS-DCV** use case shown in the sidebar follows a typical outline for formal use cases. The *goal in context* identifies the overall scope of the use case. The *precondition* describes what is known to be true before the use case is initiated. The *trigger* identifies the event or condition that “gets the use case started” [Coc01b]. The *scenario* lists the specific actions that are required by the actor and the appropriate system responses. *Exceptions* identify the situations uncovered as the preliminary use case is refined (Section 6.2.2). Additional headings may or may not be included and are reasonably self-explanatory.

## SAFEHOME



### Use Case Template for Surveillance



Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

**Iteration:** 2, last modification: January 14 by V. Raman.

**Primary actor:** Homeowner.

**Goal in context:** To view output of camera placed throughout the house from any remote location via the Internet.

**Preconditions:** System must be fully configured; appropriate user ID and passwords must be obtained.

**Trigger:** The homeowner decides to take a look inside the house while away.

**Scenario:**

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the “surveillance” from the major function buttons.
6. The homeowner selects “pick a camera.”
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the “view” button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

**Exceptions:**

1. ID or passwords are incorrect or not recognized—see use case **Validate ID and passwords**.
2. Surveillance function not configured for this system—system displays appropriate error message; see use case **Configure surveillance function**.
3. Homeowner selects “View thumbnail snapshots for all camera”—see use case **View thumbnail snapshots for all cameras**.
4. A floor plan is not available or has not been configured—display appropriate error message and see use case **Configure floor plan**.
5. An alarm condition is encountered—see use case **Alarm condition encountered**.

**Priority:** Moderate priority, to be implemented after basic functions.

**When available:** Third increment.

**Frequency of use:** Moderate frequency.

**Channel to actor:** Via PC-based browser and Internet connection.

**Secondary actors:** System administrator, cameras.

**Channels to secondary actors:**

1. System administrator: PC-based system.
2. Cameras: wireless connectivity.

**Open issues:**

1. What mechanisms protect unauthorized use of this capability by employees of *SafeHome Products*?
2. Is security sufficient? Hacking into this feature would represent a major invasion of privacy.
3. Will system response via the Internet be acceptable given the bandwidth required for camera views?
4. Will we develop a capability to provide video at a higher frames-per-second rate when high-bandwidth connections are available?

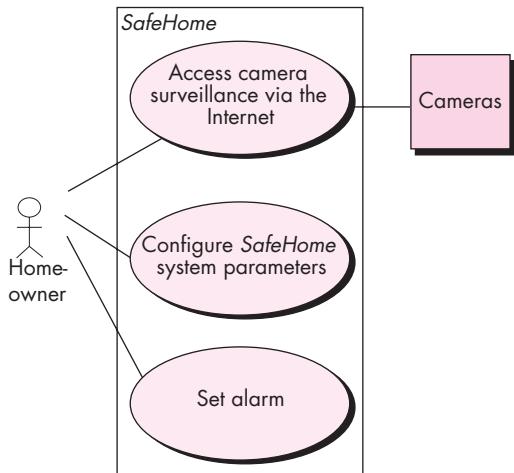
**WebRef**

When are you finished writing use cases? For a worthwhile discussion of this topic, see [ootips.org/use-cases-done.html](http://ootips.org/use-cases-done.html).

In many cases, there is no need to create a graphical representation of a usage scenario. However, diagrammatic representation can facilitate understanding, particularly when the scenario is complex. As we noted earlier in this book, UML does provide use-case diagramming capability. Figure 6.4 depicts a preliminary use-case diagram for the *SafeHome* product. Each use case is represented by an oval. Only the **ACS-DCV** use case has been discussed in this section.

**FIGURE 6.4**

Preliminary  
use-case  
diagram for  
the *SafeHome*  
system



Every modeling notation has limitations, and the use case is no exception. Like any other form of written description, a use case is only as good as its author(s). If the description is unclear, the use case can be misleading or ambiguous. A use case focuses on functional and behavioral requirements and is generally inappropriate for nonfunctional requirements. For situations in which the requirements model must have significant detail and precision (e.g., safety critical systems), a use case may not be sufficient.

However, scenario-based modeling is appropriate for a significant majority of all situations that you will encounter as a software engineer. If developed properly, the use case can provide substantial benefit as a modeling tool.

### 6.3 UML MODELS THAT SUPPLEMENT THE USE CASE

There are many requirements modeling situations in which a text-based model—even one as simple as a use case—may not impart information in a clear and concise manner. In such cases, you can choose from a broad array of UML graphical models.



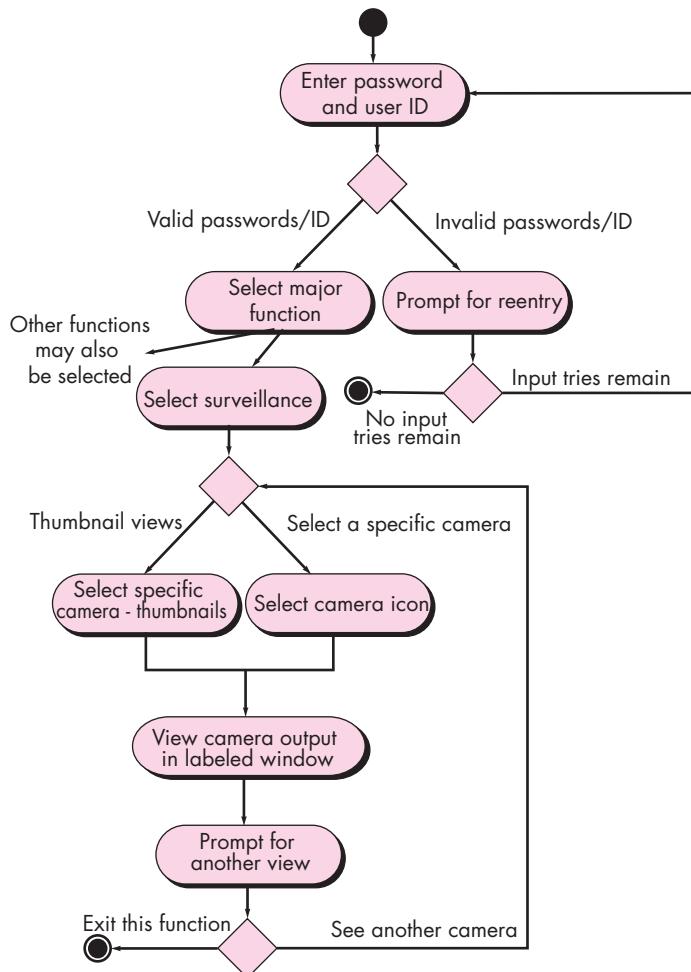
A UML activity diagram represents the actions and decisions that occur as some function is performed.

#### 6.3.1 Developing an Activity Diagram

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart, an activity diagram uses rounded rectangles to imply a specific system function, arrows to represent flow through the system, decision diamonds to depict a branching decision (each arrow emanating from the diamond is labeled), and solid horizontal lines to indicate that parallel activities are occurring. An activity diagram for the **ACS-DCV** use case is shown in Figure 6.5. It should be noted that the activity diagram adds additional detail not directly mentioned (but implied) by the use case.

**FIGURE 6.5**

Activity diagram for Access camera surveillance via the Internet—display camera views function.



For example, a user may only attempt to enter **userID** and **password** a limited number of times. This is represented by a decision diamond below “Prompt for reentry.”

### 6.3.2 Swimlane Diagrams

The UML *swimlane diagram* is a useful variation of the activity diagram and allows you to represent the flow of activities described by the use case and at the same time indicate which actor (if there are multiple actors involved in a specific use case) or analysis class (discussed later in this chapter) has responsibility for the action described by an activity rectangle. Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

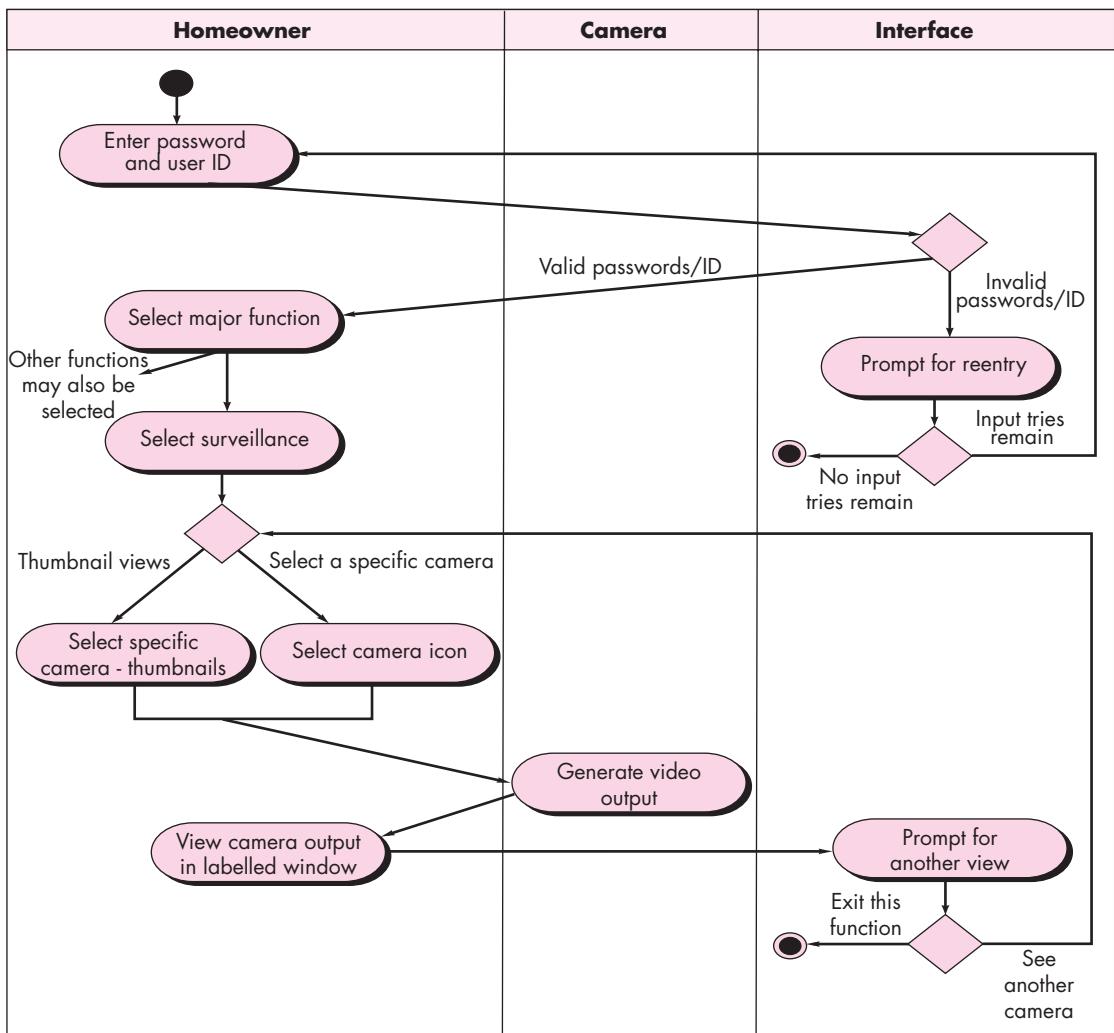
Three analysis classes—**Homeowner**, **Camera**, and **Interface**—have direct or indirect responsibilities in the context of the activity diagram represented in Figure 6.5.



A UML swimlane diagram represents the flow of actions and decisions and indicates which actors perform each.

**FIGURE 6.6**

Swimlane diagram for Access camera surveillance via the Internet—display camera views function



**Quote:**  
“A good model guides your thinking, a bad one warps it.”

Brian Marick

Referring to Figure 6.6, the activity diagram is rearranged so that activities associated with a particular analysis class fall inside the swimlane for that class. For example, the **Interface** class represents the user interface as seen by the homeowner. The activity diagram notes two prompts that are the responsibility of the interface—"prompt for reentry" and "prompt for another view." These prompts and the decisions associated with them fall within the **Interface** swimlane. However, arrows lead from that swimlane back to the **Homeowner** swimlane, where homeowner actions occur.

Use cases, along with the activity and swimlane diagrams, are procedurally oriented. They represent the manner in which various actors invoke specific functions

(or other procedural steps) to meet the requirements of the system. But a procedural view of requirements represents only a single dimension of a system. In Section 6.4, I examine the information space and how data requirements can be represented.

## 6.4 DATA MODELING CONCEPTS

### WebRef

Useful information on data modeling can be found at [www.datamodel.org](http://www.datamodel.org).

If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a *data model* as part of overall requirements modeling. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships. The *entity-relationship diagram* (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application.

### 6.4.1 Data Objects



A *data object* is a representation of composite information that must be understood by software. By *composite information*, I mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but **dimensions** (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a **person** or a **car** can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data.<sup>10</sup> Therefore, the data object can be represented as a table as shown in Figure 6.7. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of **make**, **model**, **ID number**, **body type**, **color**, and **owner**. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object **car**.

### KEY POINT

A data object is a representation of any composite information that is processed by software.

### KEY POINT

Attributes name a data object, describe its characteristics, and in some cases, make reference to another object.

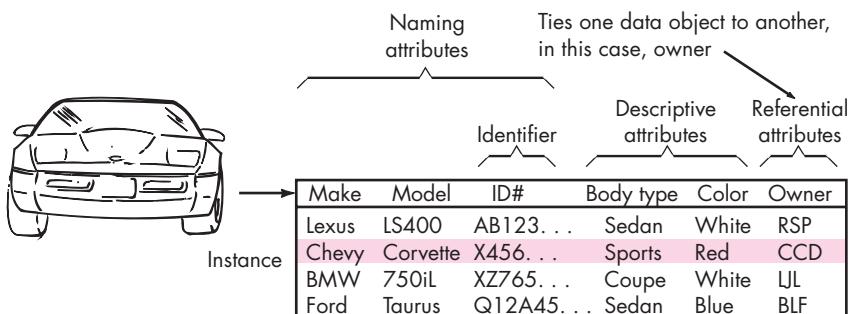
### 6.4.2 Data Attributes

*Data attributes* define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier

<sup>10</sup> This distinction separates the data object from the class or object defined as part of the object-oriented approach (Appendix 2).

**FIGURE 6.7**

Tabular representation of data objects



attribute becomes a “key” when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object **car**, a reasonable identifier might be the **ID number**.

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for **car** might serve well for an application that would be used by a department of motor vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for **car** might also include **ID number**, **body type**, and **color**, but many additional attributes (e.g., **interior code**, **drive train type**, **trim package designator**, **transmission type**) would have to be added to make **car** a meaningful object in the manufacturing control context.

**WebRef**

A concept called “normalization” is important to those who intend to do thorough data modeling. A useful introduction can be found at  
[www.datamodel.org](http://www.datamodel.org).


**Data Objects and Object-Oriented Classes—Are They the Same Thing?**

A common question occurs when data objects are discussed: Is a data object the same thing as an object-oriented<sup>11</sup> class? The answer is “no.”

A data object defines a composite data item; that is, it incorporates a collection of individual data items (attributes) and gives the collection of items a name (the name of the data object).

An object-oriented class encapsulates data attributes but also incorporates the operations (methods) that

manipulate the data implied by those attributes. In addition, the definition of classes implies a comprehensive infrastructure that is part of the object-oriented software engineering approach. Classes communicate with one another via messages, they can be organized into hierarchies, and they provide inheritance characteristics for objects that are an instance of a class.

**INFO**

### 6.4.3 Relationships

Data objects are connected to one another in different ways. Consider the two data objects, **person** and **car**. These objects can be represented using the simple notation

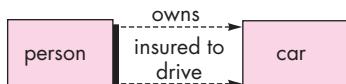
<sup>11</sup> Readers who are unfamiliar with object-oriented concepts and terminology should refer to the brief tutorial presented in Appendix 2.

**FIGURE 6.8**

**Relationships between data objects**



(a) A basic connection between data objects



(b) Relationships between data objects

## KEY POINT

Relationships indicate the manner in which data objects are connected to one another.

illustrated in Figure 6.8a. A connection is established between **person** and **car** because the two objects are related. But what are the relationships? To determine the answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. You can establish a set of object/relationship pairs that define the relevant relationships. For example,

- A person *owns* a car.
- A person is *insured to drive* a car.

The relationships *owns* and *insured to drive* define the relevant connections between **person** and **car**. Figure 6.8b illustrates these object-relationship pairs graphically. The arrows noted in Figure 6.8b provide important information about the directionality of the relationship and often reduce ambiguity or misinterpretations.

## INFO



### Entity-Relationship Diagrams

The object-relationship pair is the cornerstone of the data model. These pairs can be represented graphically using the entity-relationship diagram (ERD).<sup>12</sup> The ERD was originally proposed by Peter Chen [Che77] for the design of relational database systems and has been extended by others. A set of primary components is identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships.

Rudimentary ERD notation has already been introduced. Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality.<sup>13</sup> If you desire further information about data modeling and the entity-relationship diagram, see [Hob06] or [Sim05].

<sup>12</sup> Although the ERD is still used in some database design applications, UML notation (Appendix 1) can now be used for data design.

<sup>13</sup> The *cardinality* of an object-relationship pair specifies "the number of occurrences of one [object] that can be related to the number of occurrences of another [object]" [Til93]. The *modality* of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory.

## SOFTWARE TOOLS



### **Data Modeling**

**Objective:** Data modeling tools provide a software engineer with the ability to represent data objects, their characteristics, and their relationships.

Used primarily for large database applications and other information systems projects, data modeling tools provide an automated means for creating comprehensive entity-relation diagrams, data object dictionaries, and related models.

**Mechanics:** Tools in this category enable the user to describe data objects and their relationships. In some cases, the tools use ERD notation. In others, the tools model relations using some other mechanism. Tools in this category are often used as part of database design and enable the creation of a database model by generating a database schema for common database management systems (DBMS).

#### **Representative Tools:**<sup>14</sup>

*AllFusion ERWin*, developed by Computer Associates

([www3.ca.com](http://www3.ca.com)), assists in the design of data objects, proper structure, and key elements for databases.

*ER/Studio*, developed by Embarcadero Software

([www.embarcadero.com](http://www.embarcadero.com)), supports entity-relationship modeling.

*Oracle Designer*, developed by Oracle Systems

([www.oracle.com](http://www.oracle.com)), “models business processes, data entities and relationships [that] are transformed into designs from which complete applications and databases are generated.”

*Visible Analyst*, developed by Visible Systems

([www.visible.com](http://www.visible.com)), supports a variety of analysis modeling functions including data modeling.

## 6.5 CLASS-BASED MODELING

Class-based modeling represents the objects that the system will manipulate, the operations (also called methods or services) that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. The elements of a class-based model include classes and objects, attributes, operations, class-responsibility-collaborator (CRC) models, collaboration diagrams, and packages. The sections that follow present a series of informal guidelines that will assist in their identification and representation.

### 6.5.1 Identifying Analysis Classes

#### **vote:**

“The really hard problem is discovering what are the right objects [classes] in the first place.”

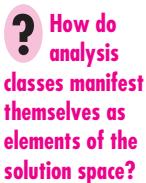
**Carl Argila**

If you look around a room, there is a set of physical objects that can be easily identified, classified, and defined (in terms of attributes and operations). But when you “look around” the problem space of a software application, the classes (and objects) may be more difficult to comprehend.

We can begin to identify classes by examining the usage scenarios developed as part of the requirements model and performing a “grammatical parse” [Abb83] on the use cases developed for the system to be built. Classes are determined by underlining each noun or noun phrase and entering it into a simple table. Synonyms should be noted. If the class (noun) is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a solution, it is part of the problem space.

<sup>14</sup> Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

But what should we look for once all of the nouns have been isolated? *Analysis classes* manifest themselves in one of the following ways:



- *External entities* (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- *Things* (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
- *Occurrences or events* (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- *Roles* (e.g., manager, engineer, salesperson) played by people who interact with the system.
- *Organizational units* (e.g., division, group, team) that are relevant to an application.
- *Places* (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- *Structures* (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

This categorization is but one of many that have been proposed in the literature.<sup>15</sup> For example, Budd [Bud96] suggests a taxonomy of classes that includes *producers* (sources) and *consumers* (sinks) of data, *data managers*, *view* or *observer classes*, and *helper classes*.

It is also important to note what classes or objects are not. In general, a class should never have an “imperative procedural name” [Cas89]. For example, if the developers of software for a medical imaging system defined an object with the name **InvertImage** or even **ImageInversion**, they would be making a subtle mistake. The **Image** obtained from the software could, of course, be a class (it is a thing that is part of the information domain). Inversion of the image is an operation that is applied to the object. It is likely that inversion would be defined as an operation for the object **Image**, but it would not be defined as a separate class to connote “image inversion.” As Cashman [Cas89] states: “the intent of object-orientation is to encapsulate, but still keep separate, data and operations on the data.”

To illustrate how analysis classes might be defined during the early stages of modeling, consider a grammatical parse (nouns are underlined, verbs italicized) for a processing narrative<sup>16</sup> for the *SafeHome* security function.

---

<sup>15</sup> Another important categorization, defining entity, boundary, and controller classes, is discussed in Section 6.5.4.

<sup>16</sup> A processing narrative is similar to the use case in style but somewhat different in purpose. The processing narrative provides an overall description of the function to be developed. It is not a scenario written from one actor’s point of view. It is important to note, however, that a grammatical parse can also be used for every use case developed as part of requirements gathering (elicitation).

The SafeHome security function enables the homeowner to configure the security system when it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through the Internet, a PC, or a control panel.

During installation, the SafeHome PC is used to program and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone number(s) are input for dialing when a sensor event occurs.



*The grammatical parse is not foolproof, but it can provide you with an excellent jump start, if you're struggling to define data objects and the transforms that operate on them.*

When a sensor event is recognized, the software invokes an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, provides information about the location, reporting the nature of the event that has been detected. The telephone number will be redialed every 20 seconds until telephone connection is obtained.

The homeowner receives security information via a control panel, the PC, or a browser, collectively called an interface. The interface displays prompting messages and system status information on the control panel, the PC, or the browser window. Homeowner interaction takes the following form . . .

Extracting the nouns, we can propose a number of potential classes:

Potential Class	General Classification
homeowner	role or external entity
sensor	external entity
control panel	external entity
installation	occurrence
system (alias security system)	thing
number, type	not objects, attributes of sensor
master password	thing
telephone number	thing
sensor event	occurrence
audible alarm	external entity
monitoring service	organizational unit or external entity

The list would be continued until all nouns in the processing narrative have been considered. Note that I call each entry in the list a potential object. You must consider each further before a final decision is made.

Coad and Yourdon [Coad91] suggest six selection characteristics that should be used as you consider each potential class for inclusion in the analysis model:

**How do I determine whether a potential class should, in fact, become an analysis class?**

1. *Retained information.* The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
2. *Needed services.* The potential class must have a set of identifiable operations that can change the value of its attributes in some way.

3. *Multiple attributes.* During requirement analysis, the focus should be on “major” information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
4. *Common attributes.* A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.
5. *Common operations.* A set of operations can be defined for the potential class and these operations apply to all instances of the class.
6. *Essential requirements.* External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

**note:**

“Classes struggle,  
some classes  
triumph, others are  
eliminated.”

**Mao Zedong**

To be considered a legitimate class for inclusion in the requirements model, a potential object should satisfy all (or almost all) of these characteristics. The decision for inclusion of potential classes in the analysis model is somewhat subjective, and later evaluation may cause an object to be discarded or reinstated. However, the first step of class-based modeling is the definition of classes, and decisions (even subjective ones) must be made. With this in mind, you should apply the selection characteristics to the list of potential *SafeHome* classes:

Potential Class	Characteristic Number That Applies
homeowner	rejected: 1, 2 fail even though 6 applies
sensor	accepted: all apply
control panel	accepted: all apply
installation	rejected
system (alias security function)	accepted: all apply
number, type	rejected: 3 fails, attributes of sensor
master password	rejected: 3 fails
telephone number	rejected: 3 fails
sensor event	accepted: all apply
audible alarm	accepted: 2, 3, 4, 5, 6 apply
monitoring service	rejected: 1, 2 fail even though 6 applies

It should be noted that (1) the preceding list is not all-inclusive, additional classes would have to be added to complete the model; (2) some of the rejected potential classes will become attributes for those classes that were accepted (e.g., **number** and **type** are attributes of **Sensor**, and **master password** and **telephone number** may become attributes of **System**); (3) different statements of the problem might cause different “accept or reject” decisions to be made (e.g., if each homeowner had an individual password or was identified by voice print, the **Homeowner** class would satisfy characteristics 1 and 2 and would have been accepted).

### 6.5.2 Specifying Attributes



Attributes are the set of data objects that fully define the class within the context of the problem.

Attributes describe a class that has been selected for inclusion in the requirements model. In essence, it is the attributes that define the class—that clarify what is meant by the class in the context of the problem space. For example, if we were to build a system that tracks baseball statistics for professional baseball players, the attributes of the class **Player** would be quite different than the attributes of the same class when it is used in the context of the professional baseball pension system. In the former, attributes such as **name**, **position**, **batting average**, **fielding percentage**, **years played**, and **games played** might be relevant. For the latter, some of these attributes would be meaningful, but others would be replaced (or augmented) by attributes like **average salary**, **credit toward full vesting**, **pension plan options chosen**, **mailing address**, and the like.

To develop a meaningful set of attributes for an analysis class, you should study each use case and select those “things” that reasonably “belong” to the class. In addition, the following question should be answered for each class: “What data items (composite and/or elementary) fully define this class in the context of the problem at hand?”

To illustrate, we consider the **System** class defined for *SafeHome*. A homeowner can configure the security function to reflect sensor information, alarm response information, activation/deactivation information, identification information, and so forth. We can represent these composite data items in the following manner:

```
identification information = system ID + verification phone number + system status
alarm response information = delay time + telephone number
activation/deactivation information = master password + number of allowable tries +
temporary password
```

Each of the data items to the right of the equal sign could be further defined to an elementary level, but for our purposes, they constitute a reasonable list of attributes for the **System** class (shaded portion of Figure 6.9).

Sensors are part of the overall *SafeHome* system, and yet they are not listed as data items or as attributes in Figure 6.9. **Sensor** has already been defined as a class, and multiple **Sensor** objects will be associated with the **System** class. In general, we avoid defining an item as an attribute if more than one of the items is to be associated with the class.



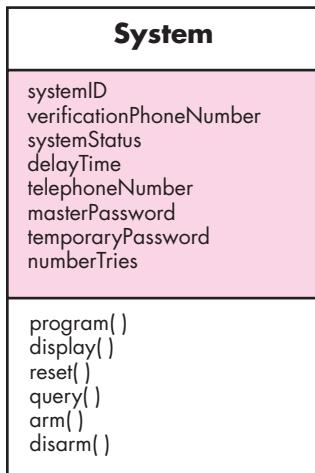
When you define operations for an analysis class, focus on problem-oriented behavior rather than behaviors required for implementation.

### 6.5.3 Defining Operations

Operations define the behavior of an object. Although many different types of operations exist, they can generally be divided into four broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, (3) operations that inquire about the state

**FIGURE 6.9**

Class diagram  
for the system  
class



of an object, and (4) operations that monitor an object for the occurrence of a controlling event. These functions are accomplished by operating on attributes and/or associations (Section 6.5.5). Therefore, an operation must have “knowledge” of the nature of the class’ attributes and associations.

As a first iteration at deriving a set of operations for an analysis class, you can again study a processing narrative (or use case) and select those operations that reasonably belong to the class. To accomplish this, the grammatical parse is again studied and verbs are isolated. Some of these verbs will be legitimate operations and can be easily connected to a specific class. For example, from the *SafeHome* processing narrative presented earlier in this chapter, we see that “sensor is *assigned* a number and type” or “a master password is *programmed* for *arming* and *disarming* the system.” These phrases indicate a number of things:

- That an *assign()* operation is relevant for the **Sensor** class.
- That a *program()* operation will be applied to the **System** class.
- That *arm()* and *disarm()* are operations that apply to **System** class.

Upon further investigation, it is likely that the operation *program()* will be divided into a number of more specific suboperations required to configure the system. For example, *program()* implies specifying phone numbers, configuring system characteristics (e.g., creating the sensor table, entering alarm characteristics), and entering password(s). But for now, we specify *program()* as a single operation.

In addition to the grammatical parse, you can gain additional insight into other operations by considering the communication that occurs between objects. Objects communicate by passing messages to one another. Before continuing with the specification of operations, I explore this matter in a bit more detail.

## SAFEHOME



### Class Models

**The scene:** Ed's cubicle, as requirements modeling begins.

**The players:** Jamie, Vinod, and Ed—all members of the *SafeHome* software engineering team.

#### The conversation:

[Ed has been working to extract classes from the use case template for ACS-DCV (presented in an earlier sidebar in this chapter) and is presenting the classes he has extracted to his colleagues.]

**Ed:** So when the homeowner wants to pick a camera, he or she has to pick it from a floor plan. I've defined a **FloorPlan** class. Here's the diagram.

(They look at Figure 6.10.)

**Jamie:** So **FloorPlan** is an object that is put together with walls, doors, windows, and cameras. That's what those labeled lines mean, right?

**Ed:** Yeah, they're called "associations." One class is associated with another according to the associations I've shown. [Associations are discussed in Section 6.5.5.]

**Vinod:** So the actual floor plan is made up of walls and contains cameras and sensors that are placed within those walls. How does the floor plan know where to put those objects?

**Ed:** It doesn't, but the other classes do. See the attributes under, say, **WallSegment**, which is used to build a wall. The wall segment has start and stop coordinates and the **draw()** operation does the rest.

**Jamie:** And the same goes for windows and doors. Looks like camera has a few extra attributes.

**Ed:** Yeah, I need them to provide pan and zoom info.

**Vinod:** I have a question. Why does the camera have an ID but the others don't? I notice you have an attribute called **nextWall**. How will **WallSegment** know what the next wall will be?

**Ed:** Good question, but as they say, that's a design decision, so I'm going to delay that until . . .

**Jamie:** Give me a break . . . I'll bet you've already figured it out.

**Ed (smiling sheepishly):** True, I'm gonna use a list structure which I'll model when we get to design. If you get religious about separating analysis and design, the level of detail I have right here could be suspect.

**Jamie:** Looks pretty good to me, but I have a few more questions.

(Jamie asks questions which result in minor modifications)

**Vinod:** Do you have CRC cards for each of the objects? If so, we ought to role-play through them, just to make sure nothing has been omitted.

**Ed:** I'm not quite sure how to do them.

**Vinod:** It's not hard and they really pay off. I'll show you.

#### note:

"One purpose of CRC cards is to fail early, to fail often, and to fail inexpensively. It is a lot cheaper to tear up a bunch of cards than it would be to reorganize a large amount of source code."

C. Horstmann

### 6.5.4 Class-Responsibility-Collaborator (CRC) Modeling

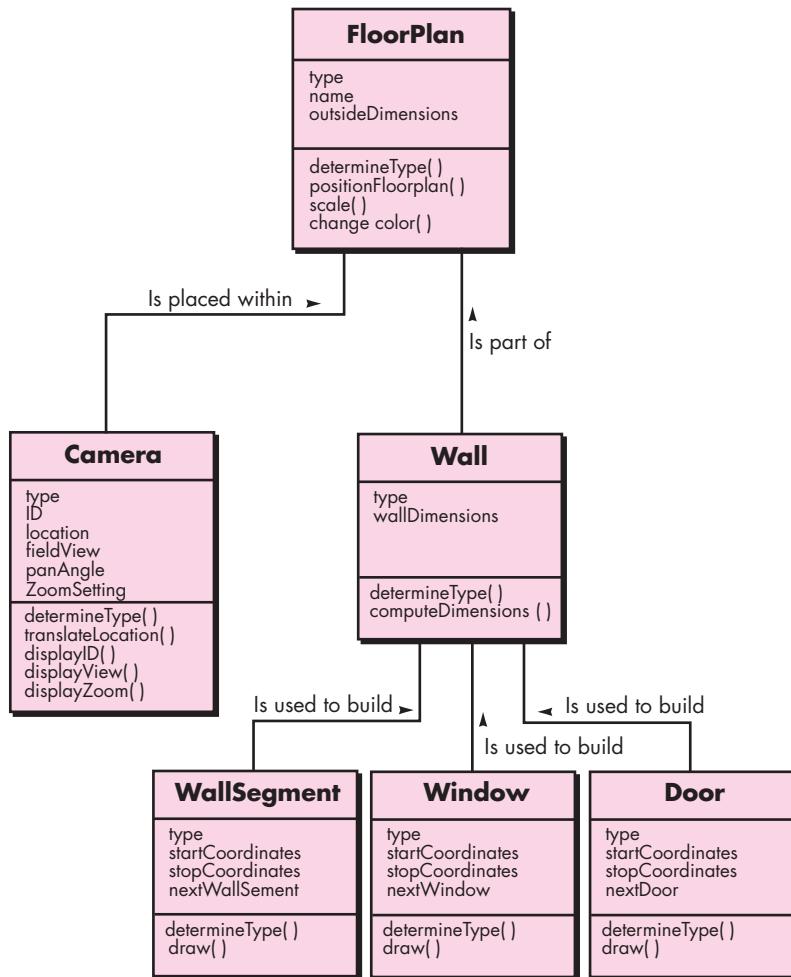
*Class-responsibility-collaborator (CRC) modeling* [Wir90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. Ambler [Amb95] describes CRC modeling in the following way:

A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.

In reality, the CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. *Responsibilities* are the attributes and operations that are relevant for the class. Stated simply, a responsibility is "anything the class knows or does" [Amb95]. *Collaborators* are those classes that are

**FIGURE 6.10**

Class diagram  
for **FloorPlan**  
(see sidebar  
discussion)



required to provide a class with the information needed to complete a responsibility. In general, a *collaboration* implies either a request for information or a request for some action.

A simple CRC index card for the **FloorPlan** class is illustrated in Figure 6.11. The list of responsibilities shown on the CRC card is preliminary and subject to additions or modification. The classes **Wall** and **Camera** are noted next to the responsibility that will require their collaboration.

### WebRef

An excellent discussion of these class types can be found at  
[www.theumlcafe.com/a0079.htm](http://www.theumlcafe.com/a0079.htm).

**Classes.** Basic guidelines for identifying classes and objects were presented earlier in this chapter. The taxonomy of class types presented in Section 6.5.1 can be extended by considering the following categories:

- *Entity classes*, also called *model* or *business* classes, are extracted directly from the statement of the problem (e.g., **FloorPlan** and **Sensor**). These

**FIGURE 6.11**

A CRC model index card

**Class: FloorPlan**

Description

<b>Responsibility:</b>	<b>Collaborator:</b>
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors, and windows	<b>Wall</b>
Shows position of video cameras	<b>Camera</b>

**note:**

"Objects can be classified scientifically into three major categories: those that don't work, those that break down, and those that get lost."

Russell Baker

classes typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifically deleted).

- **Boundary classes** are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used. Entity objects contain information that is important to users, but they do not display themselves. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users. For example, a boundary class called **CameraWindow** would have the responsibility of displaying surveillance camera output for the *SafeHome* system.
- **Controller classes** manage a "unit of work" [UML03] from start to finish. That is, controller classes can be designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, (4) validation of data communicated between objects or between the user and the application. In general, controller classes are not considered until the design activity has begun.

**?** What guidelines can be applied for allocating responsibilities to classes?

**Responsibilities.** Basic guidelines for identifying responsibilities (attributes and operations) have been presented in Sections 6.5.2 and 6.5.3. Wirfs-Brock and her colleagues [Wir90] suggest five guidelines for allocating responsibilities to classes:

1. **System intelligence should be distributed across classes to best address the needs of the problem.** Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do. This intelligence can be distributed across classes in a number of

different ways. “Dumb” classes (those that have few responsibilities) can be modeled to act as servants to a few “smart” classes (those having many responsibilities). Although this approach makes the flow of control in a system straightforward, it has a few disadvantages: it concentrates all intelligence within a few classes, making changes more difficult, and it tends to require more classes, hence more development effort.

If system intelligence is more evenly distributed across the classes in an application, each object knows about and does only a few things (that are generally well focused), the cohesiveness of the system is improved.<sup>17</sup> This enhances the maintainability of the software and reduces the impact of side effects due to change.

To determine whether system intelligence is properly distributed, the responsibilities noted on each CRC model index card should be evaluated to determine if any class has an extraordinarily long list of responsibilities. This indicates a concentration of intelligence.<sup>18</sup> In addition, the responsibilities for each class should exhibit the same level of abstraction. For example, among the operations listed for an aggregate class called **CheckingAccount** a reviewer notes two responsibilities: *balance-the-account* and *check-off-cleared-checks*. The first operation (responsibility) implies a complex mathematical and logical procedure. The second is a simple clerical activity. Since these two operations are not at the same level of abstraction, *check-off-cleared-checks* should be placed within the responsibilities of **CheckEntry**, a class that is encompassed by the aggregate class **CheckingAccount**.

- 2. Each responsibility should be stated as generally as possible.** This guideline implies that general responsibilities (both attributes and operations) should reside high in the class hierarchy (because they are generic, they will apply to all subclasses).
- 3. Information and the behavior related to it should reside within the same class.** This achieves the object-oriented principle called *encapsulation*. Data and the processes that manipulate the data should be packaged as a cohesive unit.
- 4. Information about one thing should be localized with a single class, not distributed across multiple classes.** A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.

---

17 Cohesiveness is a design concept that is discussed in Chapter 8.

18 In such cases, it may be necessary to split the class into multiple classes or complete subsystems in order to distribute intelligence more effectively.

**5. Responsibilities should be shared among related classes, when appropriate.**

There are many cases in which a variety of related objects must all exhibit the same behavior at the same time. As an example, consider a video game that must display the following classes: **Player**, **PlayerBody**,

**PlayerArms**, **PlayerLegs**, **PlayerHead**. Each of these classes has its own attributes (e.g., **position**, **orientation**, **color**, **speed**) and all must be updated and displayed as the user manipulates a joystick. The responsibilities *update()* and *display()* must therefore be shared by each of the objects noted.

**Player** knows when something has changed and *update()* is required. It collaborates with the other objects to achieve a new position or orientation, but each object controls its own display.

**Collaborations.** Classes fulfill their responsibilities in one of two ways: (1) A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or (2) a class can collaborate with other classes. Wirsfs-Brock and her colleagues [Wir90] define collaborations in the following way:

Collaborations represent requests from a client to a server in fulfillment of a client responsibility. A collaboration is the embodiment of the contract between the client and the server. . . . We say that an object collaborates with another object if, to fulfill a responsibility, it needs to send the other object any messages. A single collaboration flows in one direction—representing a request from the client to the server. From the client's point of view, each of its collaborations is associated with a particular responsibility implemented by the server.

Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class. Hence, a collaboration.

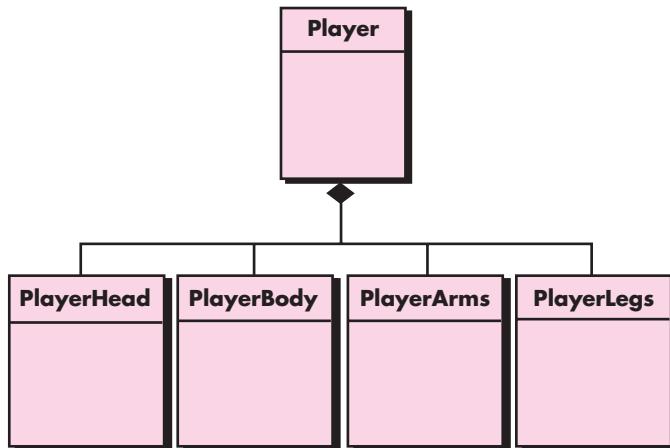
As an example, consider the *SafeHome* security function. As part of the activation procedure, the **ControlPanel** object must determine whether any sensors are open. A responsibility named *determine-sensor-status()* is defined. If sensors are open, **ControlPanel** must set a **status** attribute to “not ready.” Sensor information can be acquired from each **Sensor** object. Therefore, the responsibility *determine-sensor-status()* can be fulfilled only if **ControlPanel** works in collaboration with **Sensor**.

To help in the identification of collaborators, you can examine three different generic relationships between classes [Wir90]: (1) the *is-part-of* relationship, (2) the *has-knowledge-of* relationship, and (3) the *depends-upon* relationship. Each of the three generic relationships is considered briefly in the paragraphs that follow.

All classes that are part of an aggregate class are connected to the aggregate class via an *is-part-of* relationship. Consider the classes defined for the video game noted earlier, the class **PlayerBody** *is-part-of* **Player**, as are **PlayerArms**, **PlayerLegs**, and **PlayerHead**. In UML, these relationships are represented as the aggregation shown in Figure 6.12.

**FIGURE 6.12**

A composite aggregate class



When one class must acquire information from another class, the *has-knowledge-of* relationship is established. The *determine-sensor-status()* responsibility noted earlier is an example of a *has-knowledge-of* relationship.

The *depends-upon* relationship implies that two classes have a dependency that is not achieved by *has-knowledge-of* or *is-part-of*. For example, **PlayerHead** must always be connected to **PlayerBody** (unless the video game is particularly violent), yet each object could exist without direct knowledge of the other. An attribute of the **PlayerHead** object called **center-position** is determined from the center position of **PlayerBody**. This information is obtained via a third object, **Player**, that acquires it from **PlayerBody**. Hence, **PlayerHead** *depends-upon* **PlayerBody**.

In all cases, the collaborator class name is recorded on the CRC model index card next to the responsibility that has spawned the collaboration. Therefore, the index card contains a list of responsibilities and the corresponding collaborations that enable the responsibilities to be fulfilled (Figure 6.11).

When a complete CRC model has been developed, stakeholders can review the model using the following approach [Amb95]:

1. All participants in the review (of the CRC model) are given a subset of the CRC model index cards. Cards that collaborate should be separated (i.e., no reviewer should have two cards that collaborate).
2. All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
3. The review leader reads the use case deliberately. As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card. For example, a use case for *SafeHome* contains the following narrative:

The homeowner observes the *SafeHome* control panel to determine if the system is ready for input. If the system is not ready, the homeowner must physically close

windows/doors so that the ready indicator is present. [A not-ready indicator implies that a sensor is open, i.e., that a door or window is open.]

When the review leader comes to “control panel,” in the use case narrative, the token is passed to the person holding the **ControlPanel** index card. The phrase “implies that a sensor is open” requires that the index card contains a responsibility that will validate this implication (the responsibility *determine-sensor-status()* accomplishes this). Next to the responsibility on the index card is the collaborator **Sensor**. The token is then passed to the **Sensor** object.

4. When the token is passed, the holder of the **Sensor** card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use case, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

This modus operandi continues until the use case is finished. When all use cases have been reviewed, requirements modeling continues.

## SAFEHOME



### CRC Models

**The scene:** Ed’s cubicle, as requirements modeling begins.

**The players:** Vinod and Ed—members of the *SafeHome* software engineering team.

#### The conversation:

[Vinod has decided to show Ed how to develop CRC cards by showing him an example.]

**Vinod:** While you’ve been working on surveillance and Jamie has been tied up with security, I’ve been working on the home management function.

**Ed:** What’s the status of that? Marketing kept changing its mind.

**Vinod:** Here’s the first-cut use case for the whole function . . . we’ve refined it a bit, but it should give you an overall view . . .

**Use case:** *SafeHome* home management function.

**Narrative:** We want to use the home management interface on a PC or an Internet connection to control electronic devices that have wireless interface controllers.

The system should allow me to turn specific lights on and off, to control appliances that are connected to a wireless interface, to set my heating and air conditioning system to temperatures that I define. To do this, I want to select the devices from a floor plan of the house. Each device must be identified on the floor plan. As an optional feature, I want to control all audiovisual devices—audio, television, DVD, digital recorders, and so forth.

With a single selection, I want to be able to set the entire house for various situations. One is *home*, another is *away*, a third is *overnight travel*, and a fourth is *extended travel*. All of these situations will have settings that will be applied to all devices. In the *overnight travel* and *extended travel* states, the system should turn lights on and off at random intervals (to make it look like someone is home) and control the heating and air conditioning system. I should be able to override these setting via the Internet with appropriate password protection . . .

**Ed:** The hardware guys have got all the wireless interfacing figured out?

**Vinod (smiling):** They're working on it; say it's no problem. Anyway, I extracted a bunch of classes for home management and we can use one as an example. Let's use the **HomeManagementInterface** class.

**Ed:** Okay . . . so the responsibilities are what . . . the attributes and operations for the class and the collaborations are the classes that the responsibilities point to.

**Vinod:** I thought you didn't understand CRC.

**Ed:** Maybe a little, but go ahead.

**Vinod:** So here's my class definition for **HomeManagementInterface**.

#### Attributes:

optionsPanel—contains info on buttons that enable user to select functionality.

situationPanel—contains info on buttons that enable user to select situation.

floorplan—same as surveillance object but this one displays devices.

deviceIcons—info on icons representing lights, appliances, HVAC, etc.

devicePanels—simulation of appliance or device control panel; allows control.

#### Operations:

*displayControl()*, *selectControl()*, *displaySituation()*, *selectSituation()*, *accessFloorplan()*, *selectDeviceIcon()*, *displayDevicePanel()*, *accessDevicePanel()*, . . .

**Class:** HomeManagementInterface

Responsibility	Collaborator
<i>displayControl()</i>	<b>OptionsPanel</b> (class)
<i>selectControl()</i>	<b>OptionsPanel</b> (class)
<i>displaySituation()</i>	<b>SituationPanel</b> (class)
<i>selectSituation()</i>	<b>SituationPanel</b> (class)
<i>accessFloorplan()</i>	<b>FloorPlan</b> (class) . . .
	. . .

**Ed:** So when the operation *accessFloorplan()* is invoked, it collaborates with the **FloorPlan** object just like the one we developed for surveillance. Wait, I have a description of it here. (They look at Figure 6.10.)

**Vinod:** Exactly. And if we wanted to review the entire class model, we could start with this index card, then go to the collaborator's index card, and from there to one of the collaborator's collaborators, and so on.

**Ed:** Good way to find omissions or errors.

**Vinod:** Yep.

## KEY POINT

An association defines a relationship between classes. Multiplicity defines how many of one class are related to how many of another class.

### 6.5.5 Associations and Dependencies

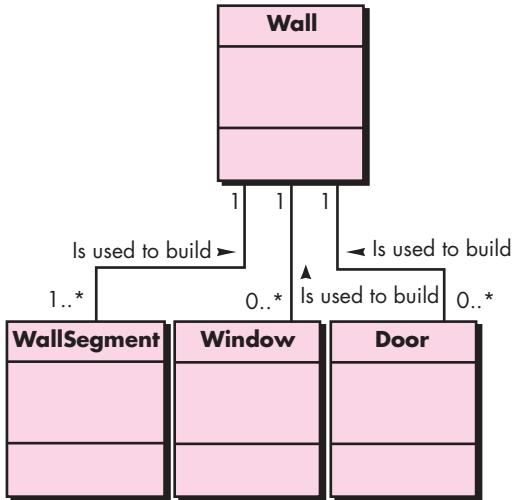
In many instances, two analysis classes are related to one another in some fashion, much like two data objects may be related to one another (Section 6.4.3). In UML these relationships are called *associations*. Referring back to Figure 6.10, the **FloorPlan** class is defined by identifying a set of associations between **FloorPlan** and two other classes, **Camera** and **Wall**. The class **Wall** is associated with three classes that allow a wall to be constructed, **WallSegment**, **Window**, and **Door**.

In some cases, an association may be further defined by indicating *multiplicity*. Referring to Figure 6.10, a **Wall** object is constructed from one or more **WallSegment** objects. In addition, the **Wall** object may contain 0 or more **Window** objects and 0 or more **Door** objects. These multiplicity constraints are illustrated in Figure 6.13, where “one or more” is represented using  $1 \dots *$ , and “0 or more” by  $0 \dots *$ . In UML, the asterisk indicates an unlimited upper bound on the range.<sup>19</sup>

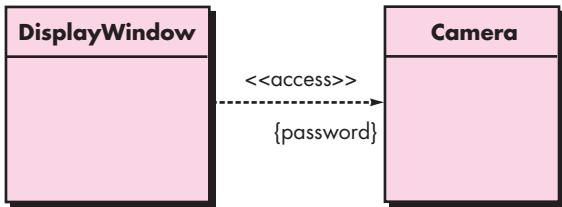
19 Other multiplicity relations—one to one, one to many, many to many, one to a specified range with lower and upper limits, and others—may be indicated as part of an association.

**FIGURE 6.13**

Multiplicity

**FIGURE 6.14**

Dependencies



**What is a stereotype?**

In many instances, a client-server relationship exists between two analysis classes. In such cases, a client class depends on the server class in some way and a *dependency relationship* is established. Dependencies are defined by a stereotype. A *stereotype* is an “extensibility mechanism” [Arl02] within UML that allows you to define a special modeling element whose semantics are custom defined. In UML stereotypes are represented in double angle brackets (e.g., **<<stereotype>>**).

As an illustration of a simple dependency within the *SafeHome* surveillance system, a **Camera** object (in this case, the server class) provides a video image to a **DisplayWindow** object (in this case, the client class). The relationship between these two objects is not a simple association, yet a dependency association does exist. In a use case written for surveillance (not shown), you learn that a special password must be provided in order to view specific camera locations. One way to achieve this is to have **Camera** request a password and then grant permission to the **DisplayWindow** to produce the video display. This can be represented as shown in Figure 6.14 where **<<access>>** implies that the use of the camera output is controlled by a special password.



A package is used to assemble a collection of related classes.

### 6.5.6 Analysis Packages

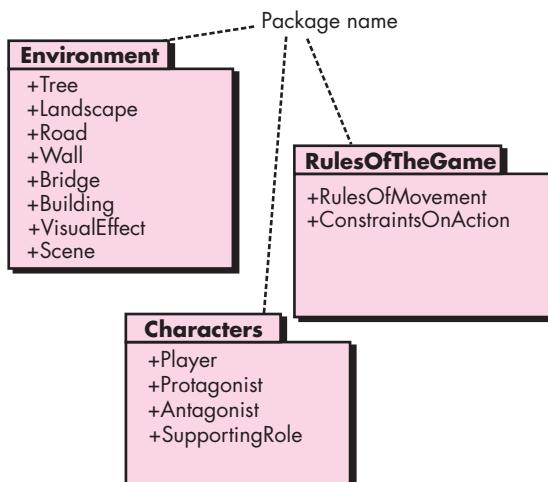
An important part of analysis modeling is categorization. That is, various elements of the analysis model (e.g., use cases, analysis classes) are categorized in a manner that packages them as a grouping—called an *analysis package*—that is given a representative name.

To illustrate the use of analysis packages, consider the video game that I introduced earlier. As the analysis model for the video game is developed, a large number of classes are derived. Some focus on the game environment—the visual scenes that the user sees as the game is played. Classes such as **Tree**, **Landscape**, **Road**, **Wall**, **Bridge**, **Building**, and **VisualEffect** might fall within this category. Others focus on the characters within the game, describing their physical features, actions, and constraints. Classes such as **Player** (described earlier), **Protagonist**, **Antagonist**, and **SupportingRoles** might be defined. Still others describe the rules of the game—how a player navigates through the environment. Classes such as **RulesOfMovement** and **ConstraintsOnAction** are candidates here. Many other categories might exist. These classes can be grouped in analysis packages as shown in Figure 6.15.

The plus sign preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages. Although they are not shown in the figure, other symbols can precede an element within a package. A minus sign indicates that an element is hidden from all other packages and a # symbol indicates that an element is accessible only to packages contained within a given package.

**FIGURE 6.15**

Packages



## 6.6 SUMMARY

The objective of requirements modeling is to create a variety of representations that describe what the customer requires, establish a basis for the creation of a software design, and define a set of requirements that can be validated once the software is built. The requirements model bridges the gap between a system-level representation that describes overall system and business functionality and a software design that describes the software's application architecture, user interface, and component-level structure.

Scenario-based models depict software requirements from the user's point of view. The use case—a narrative or template-driven description of an interaction between an actor and the software—is the primary modeling element. Derived during requirements elicitation, the use case defines the key steps for a specific function or interaction. The degree of use-case formality and detail varies, but the end result provides necessary input to all other analysis modeling activities. Scenarios can also be described using an activity diagram—a flowchart-like graphical representation that depicts the processing flow within a specific scenario. Swim-lane diagrams illustrate how the processing flow is allocated to various actors or classes.

Data modeling is used to describe the information space that will be constructed or manipulated by the software. Data modeling begins by representing data objects—composite information that must be understood by the software. The attributes of each data object are identified and relationships between data objects are described.

Class-based modeling uses information derived from scenario-based and data modeling elements to identify analysis classes. A grammatical parse may be used to extract candidate classes, attributes, and operations from text-based narratives. Criteria for the definition of a class are defined. A set of class-responsibility-collaborator index cards can be used to define relationships between classes. In addition, a variety of UML modeling notation can be applied to define hierarchies, relationships, associations, aggregations, and dependencies among classes. Analysis packages are used to categorize and group classes in a manner that makes them more manageable for large systems.

## PROBLEMS AND POINTS TO PONDER

- 6.1.** Is it possible to begin coding immediately after an analysis model has been created? Explain your answer and then argue the counterpoint.
- 6.2.** An analysis rule of thumb is that the model "should focus on requirements that are visible within the problem or business domain." What types of requirements are *not* visible in these domains? Provide a few examples.
- 6.3.** What is the purpose of domain analysis? How is it related to the concept of requirements patterns?

**6.4.** Is it possible to develop an effective analysis model without developing all four elements shown in Figure 6.3? Explain.

**6.5.** You have been asked to build one of the following systems:

- a. a network-based course registration system for your university.
- b. a Web-based order-processing system for a computer store.
- c. a simple invoicing system for a small business.
- d. an Internet-based cookbook that is built into an electric range or microwave.

Select the system that is of interest to you and develop an entity-relationship diagram that describes data objects, relationships, and attributes.

**6.6.** The department of public works for a large city has decided to develop a Web-based pothole tracking and repair system (PHTRS). A description follows:

Citizens can log onto a website and report the location and severity of potholes. As potholes are reported they are logged within a “public works department repair system” and are assigned an identifying number, stored by street address, size (on a scale of 1 to 10), location (middle, curb, etc.), district (determined from street address), and repair priority (determined from the size of the pothole). Work order data are associated with each pothole and include pothole location and size, repair crew identifying number, number of people on crew, equipment assigned, hours applied to repair, hole status (work in progress, repaired, temporary repair, not repaired), amount of filler material used, and cost of repair (computed from hours applied, number of people, material and equipment used). Finally, a damage file is created to hold information about reported damage due to the pothole and includes citizen’s name, address, phone number, type of damage, and dollar amount of damage. PHTRS is an online system; all queries are to be made interactively.

- a. Draw a UML use case diagram for the PHTRS system. You’ll have to make a number of assumptions about the manner in which a user interacts with this system.
- b. Develop a class model for the PHTRS system.

**6.7.** Write a template-based use case for the *SafeHome* home management system described informally in the sidebar following Section 6.5.4.

**6.8.** Develop a complete set of CRC model index cards on the product or system you chose as part of Problem 6.5.

**6.9.** Conduct a review of the CRC index cards with your colleagues. How many additional classes, responsibilities, and collaborators were added as a consequence of the review?

**6.10.** What is an analysis package and how might it be used?

## FURTHER READINGS AND INFORMATION SOURCES

Use cases can serve as the foundation for all requirements modeling approaches. The subject is discussed at length by Rosenberg and Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality*, Addison-Wesley, 2005), Alexander and Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*, Wiley, 2004), Bittner and Spence (*Use Case Modeling*, Addison-Wesley, 2002), Cockburn [Coc01b], and other references noted in both Chapters 5 and 6.

Data modeling presents a useful method for examining the information space. Books by Hoberman [Hob06] and Simsion and Witt [Sim05] provide reasonably comprehensive treatments. In addition, Allen and Terry (*Beginning Relational Data Modeling*, 2d ed., Apress, 2005), Allen (*Data Modeling for Everyone*, Wrox Press, 2002), Teorey and his colleagues (*Database Modeling and Design: Logical Design*, 4th ed., Morgan Kaufmann, 2005), and Carlis and Maguire (*Mastering Data Modeling*, Addison-Wesley, 2000) present detailed tutorials for creating

industry-quality data models. An interesting book by Hay (*Data Modeling Patterns*, Dorset House, 1995) presents typical data model patterns that are encountered in many different businesses.

UML modeling techniques that can be applied for both analysis and design are discussed by O'Docherty (*Object-Oriented Analysis and Design: Understanding System Development with UML 2.0*, Wiley, 2005), Arlow and Neustadt (*UML 2 and the Unified Process*, 2d ed., Addison-Wesley, 2005), Roques (*UML in Practice*, Wiley, 2004), Dennis and his colleagues (*Systems Analysis and Design with UML Version 2.0*, Wiley, 2004), Larman (*Applying UML and Patterns*, 2d ed., Prentice-Hall, 2001), and Rosenberg and Scott (*Use Case Driven Object Modeling with UML*, Addison-Wesley, 1999).

A wide variety of information sources on requirements modeling are available on the Internet. An up-to-date list of World Wide Web references that are relevant to analysis modeling can be found at the SEPA website: [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm).

# REQUIREMENTS MODELING: FLOW, BEHAVIOR, PATTERNS, AND WEBAPPS

## KEY CONCEPTS

analysis	
patterns .....	200
behavioral model .....	195
configuration model .....	211
content model .....	207
control flow model .....	191
data flow model .....	188
functional model .....	210
interaction model .....	209
navigation modeling .....	212
process specification .....	192
sequence diagrams .....	197
WebApps .....	205

**A**fter my discussion of use cases, data modeling, and class-based models in Chapter 6, it's reasonable to ask, "Aren't those requirements modeling representations enough?"

The only reasonable answer is, "That depends."

For some types of software, the use case may be the only requirements modeling representation that is required. For others, an object-oriented approach is chosen and class-based models may be developed. But in other situations, complex application requirements may demand an examination of how data objects are transformed as they move through a system; how an application behaves as a consequence of external events; whether existing domain knowledge can be adapted to the current problem; or in the case of Web-based systems and applications, how content and functionality meld to provide an end user with the ability to successfully navigate a WebApp to achieve usage goals.

## 7.1 REQUIREMENTS MODELING STRATEGIES

One view of requirements modeling, called *structured analysis*, considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system. A second approach to analysis

## QUICK LOOK

**What is it?** The requirements model has many different dimensions. In this chapter you'll learn about flow-oriented models, behavioral models, and the special requirements analysis considerations that come into play when WebApps are developed.

Each of these modeling representations supplements the use cases, data models, and class-based models discussed in Chapter 6.

**Who does it?** A software engineer (sometimes called an "analyst") builds the model using requirements elicited from various stakeholders.

**Why is it important?** Your insight into software requirements grows in direct proportion to the number of different requirements modeling dimensions. Although you may not have the time, the resources, or the inclination to develop every representation suggested in this chapter and Chapter 6, recognize that each different modeling approach provides you with a different way of looking at the problem. As a consequence, you (and other stakeholders) will be better able to assess whether you've properly specified what must be accomplished.

**What are the steps?** Flow-oriented modeling provides an indication of how data objects are transformed by processing functions. Behavioral modeling depicts the states of the system and its classes and the impact of events on these states. Pattern-based modeling makes use of existing domain knowledge to facilitate requirements analysis. WebApp requirements models are especially adapted for the representation of content, interaction, function, and configuration-related requirements.

**What is the work product?** A wide array of text-based and diagrammatic forms may be chosen for the requirements model. Each of these representations provides a view of one or more of the model elements.

**How do I ensure that I've done it right?** Requirements modeling work products must be reviewed for correctness, completeness, and consistency. They must reflect the needs of all stakeholders and establish a foundation from which design can be conducted.

modeled, called *object-oriented analysis*, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements.

Although the analysis model that we propose in this book combines features of both approaches, software teams often choose one approach and exclude all representations from the other. The question is not which is best, but rather, what combination of representations will provide stakeholders with the best model of software requirements and the most effective bridge to software design.

## 7.2 FLOW-ORIENTED MODELING

Although data flow-oriented modeling is perceived as an outdated technique by some software engineers, it continues to be one of the most widely used requirements analysis notations in use today.<sup>1</sup> Although the *data flow diagram* (DFD) and related diagrams and information are not a formal part of UML, they can be used to complement UML diagrams and provide additional insight into system requirements and flow.



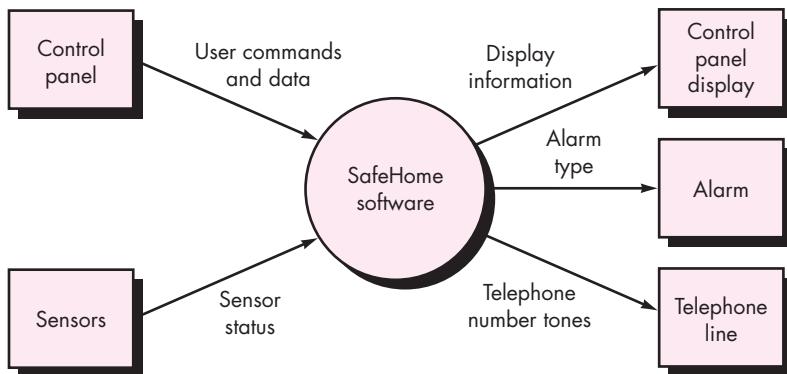
*Some will suggest that the DFD is old-school and it has no place in modern practice. That's a view that excludes a potentially useful mode of representation at the analysis level. If it can help, use the DFD.*

The DFD takes an input-process-output view of a system. That is, data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software. Data objects are represented by labeled arrows, and transformations are represented by circles (also called bubbles). The DFD is presented in a hierarchical fashion. That is, the first data flow model (sometimes called a level 0 DFD or *context diagram*) represents the system as a whole. Subsequent data flow diagrams refine the context diagram, providing increasing detail with each subsequent level.

<sup>1</sup> Data flow modeling is a core modeling activity in *structured analysis*.

**FIGURE 7.1**

Context-level DFD for the *SafeHome* security function



### 7.2.1 Creating a Data Flow Model

**Q** **note:**

"The purpose of data flow diagrams is to provide a semantic bridge between users and systems developers."

**Kenneth Kozar**

**KEY POINT**

Information flow continuity must be maintained as each DFD level is refined. This means that input and output at one level must be the same as input and output at a refined level.

The data flow diagram enables you to develop models of the information domain and functional domain. As the DFD is refined into greater levels of detail, you perform an implicit functional decomposition of the system. At the same time, the DFD refinement results in a corresponding refinement of data as it moves through the processes that embody the application.

A few simple guidelines can aid immeasurably during the derivation of a data flow diagram: (1) the level 0 data flow diagram should depict the software/system as a single bubble; (2) primary input and output should be carefully noted; (3) refinement should begin by isolating candidate processes, data objects, and data stores to be represented at the next level; (4) all arrows and bubbles should be labeled with meaningful names; (5) *information flow continuity* must be maintained from level to level,<sup>2</sup> and (6) one bubble at a time should be refined. There is a natural tendency to overcomplicate the data flow diagram. This occurs when you attempt to show too much detail too early or represent procedural aspects of the software in lieu of information flow.

To illustrate the use of the DFD and related notation, we again consider the *SafeHome* security function. A level 0 DFD for the security function is shown in Figure 7.1. The primary *external entities* (boxes) produce information for use by the system and consume information generated by the system. The labeled arrows represent data objects or data object hierarchies. For example, **user commands and data** encompasses all configuration commands, all activation/deactivation commands, all miscellaneous interactions, and all data that are entered to qualify or expand a command.

The level 0 DFD must now be expanded into a level 1 data flow model. But how do we proceed? Following an approach suggested in Chapter 6, you should apply a

2 That is, the data objects that flow into the system or into any transformation at one level must be the same data objects (or their constituent parts) that flow into the transformation at a more refined level.

“grammatical parse” [Abb83] to the use case narrative that describes the context-level bubble. That is, we isolate all nouns (and noun phrases) and verbs (and verb phrases) in a *SafeHome* processing narrative derived during the first requirements gathering meeting. Recalling the parsed processing narrative text presented in Section 6.5.1:



The grammatical parse is not foolproof, but it can provide you with an excellent jump start, if you're struggling to define data objects and the transforms that operate on them.

The SafeHome security function enables the homeowner to configure the security system when it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through the Internet, a PC, or a control panel.

During installation, the *SafeHome* PC is used to program and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone number(s) are input for dialing when a sensor event occurs.

When a sensor event is recognized, the software invokes an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, provides information about the location, reporting the nature of the event that has been detected. The telephone number will be redialed every 20 seconds until telephone connection is obtained.

The homeowner receives security information via a control panel, the PC, or a browser, collectively called an interface. The interface displays prompting messages and system status information on the control panel, the PC, or the browser window. Homeowner interaction takes the following form . . .



Be certain that the processing narrative you intend to parse is written at the same level of abstraction throughout.

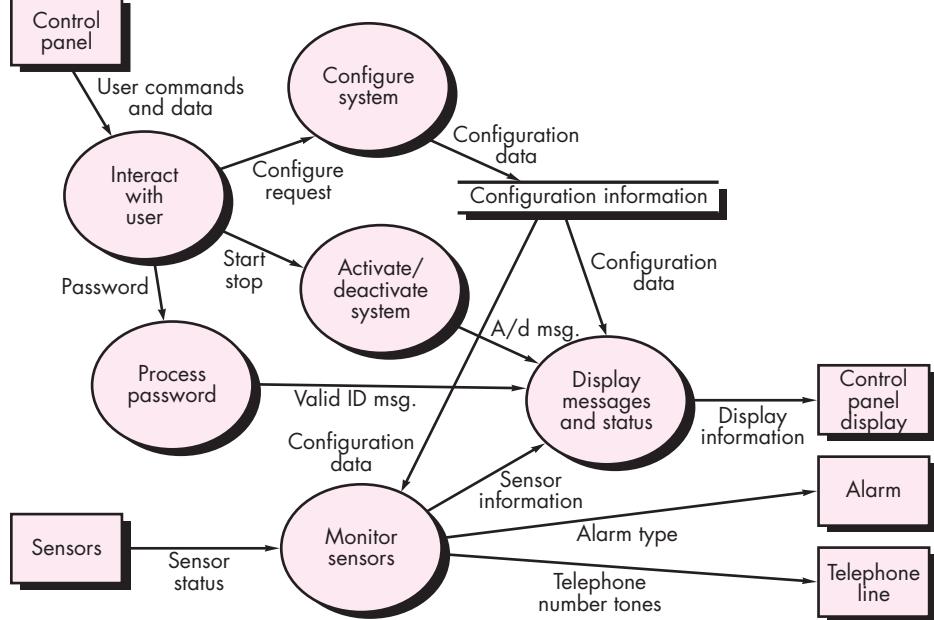
Referring to the grammatical parse, verbs are *SafeHome* processes and can be represented as bubbles in a subsequent DFD. Nouns are either external entities (boxes), data or control objects (arrows), or data stores (double lines). From the discussion in Chapter 6, recall that nouns and verbs can be associated with one another (e.g., each sensor is assigned a number and type; therefore **number** and **type** are attributes of the data object **sensor**). Therefore, by performing a grammatical parse on the processing narrative for a bubble at any DFD level, you can generate much useful information about how to proceed with the refinement to the next level. Using this information, a level 1 DFD is shown in Figure 7.2. The context level process shown in Figure 7.1 has been expanded into six processes derived from an examination of the grammatical parse. Similarly, the information flow between processes at level 1 has been derived from the parse. In addition, information flow continuity is maintained between levels 0 and 1.

The processes represented at DFD level 1 can be further refined into lower levels. For example, the process *monitor sensors* can be refined into a level 2 DFD as shown in Figure 7.3. Note once again that information flow continuity has been maintained between levels.

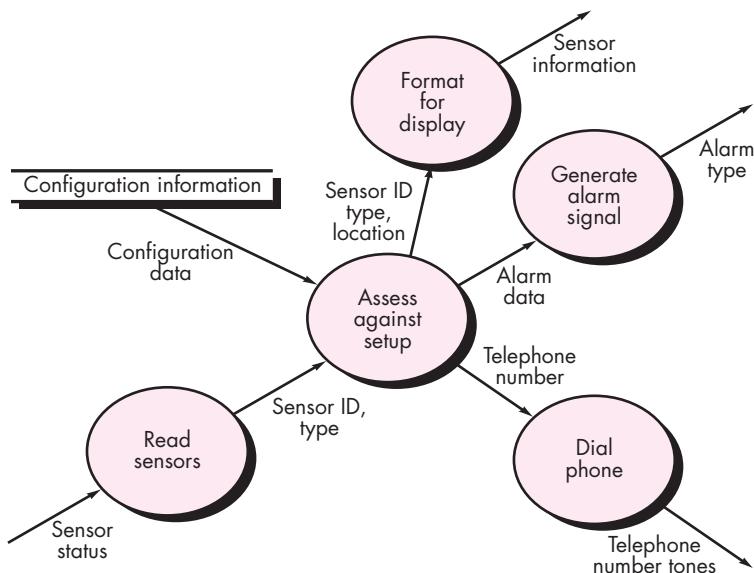
The refinement of DFDs continues until each bubble performs a simple function. That is, until the process represented by the bubble performs a function that would be easily implemented as a program component. In Chapter 8, I discuss a concept, called *cohesion*, that can be used to assess the processing focus of a given function. For now, we strive to refine DFDs until each bubble is “single-minded.”

**FIGURE 7.2**

Level 1 DFD for  
*SafeHome*  
security  
function

**FIGURE 7.3**

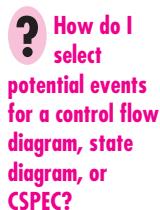
Level 2 DFD  
that refines  
the *monitor  
sensors* process



### 7.2.2 Creating a Control Flow Model

For some types of applications, the data model and the data flow diagram are all that is necessary to obtain meaningful insight into software requirements. As I have already noted, however, a large class of applications are “driven” by events rather than data, produce control information rather than reports or displays, and process information with heavy concern for time and performance. Such applications require the use of *control flow modeling* in addition to data flow modeling.

I have already noted that an event or control item is implemented as a Boolean value (e.g., true or false, on or off, 1 or 0) or a discrete list of conditions (e.g., empty, jammed, full). To select potential candidate events, the following guidelines are suggested:



- List all sensors that are “read” by the software.
- List all interrupt conditions.
- List all “switches” that are actuated by an operator.
- List all data conditions.
- Recalling the noun/verb parse that was applied to the processing narrative, review all “control items” as possible control specification inputs/outputs.
- Describe the behavior of a system by identifying its states, identify how each state is reached, and define the transitions between states.
- Focus on possible omissions—a very common error in specifying control; for example, ask: “Is there any other way I can get to this state or exit from it?”

Among the many events and control items that are part of *SafeHome* software are **sensor event** (i.e., a sensor has been tripped), **blink flag** (a signal to blink the display), and **start/stop switch** (a signal to turn the system on or off).

### 7.2.3 The Control Specification

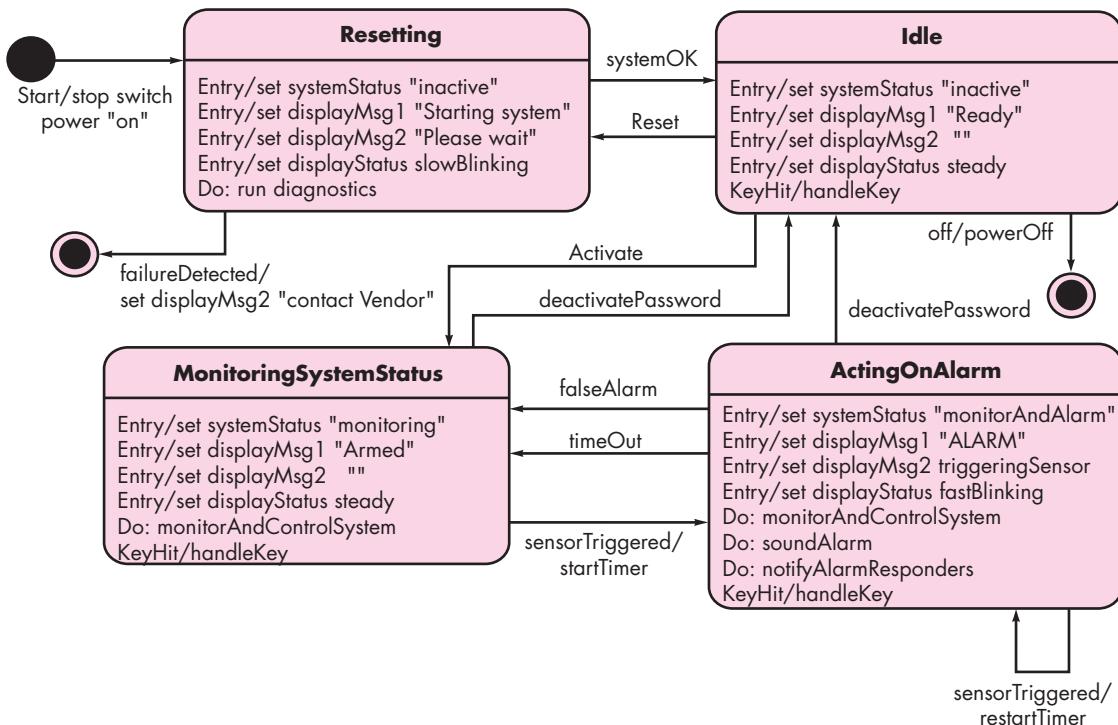
A *control specification* (CSPEC) represents the behavior of the system (at the level from which it has been referenced) in two different ways.<sup>3</sup> The CSPEC contains a state diagram that is a sequential specification of behavior. It can also contain a program activation table—a combinatorial specification of behavior.

Figure 7.4 depicts a preliminary state diagram<sup>4</sup> for the level 1 control flow model for *SafeHome*. The diagram indicates how the system responds to events as it traverses the four states defined at this level. By reviewing the state diagram, you can determine the behavior of the system and, more important, ascertain whether there are “holes” in the specified behavior.

For example, the state diagram (Figure 7.4) indicates that the transitions from the **Idle** state can occur if the system is reset, activated, or powered off. If the system is

<sup>3</sup> Additional behavioral modeling notation is presented in Section 7.3.

<sup>4</sup> The state diagram notation used here conforms to UML notation. A “state transition diagram” is available in structured analysis, but the UML format is superior in information content and representation.

**FIGURE 7.4** State diagram for *SafeHome* security function

activated (i.e., alarm system is turned on), a transition to the **MonitoringSystemStatus** state occurs, display messages are changed as shown, and the process *monitorAndControlSystem* is invoked. Two transitions occur out of the **MonitoringSystemStatus** state—(1) when the system is deactivated, a transition occurs back to the **Idle** state; (2) when a sensor is triggered into the **ActingOnAlarm** state. All transitions and the content of all states are considered during the review.

A somewhat different mode of behavioral representation is the process activation table. The PAT represents information contained in the state diagram in the context of processes, not states. That is, the table indicates which processes (bubbles) in the flow model will be invoked when an event occurs. The PAT can be used as a guide for a designer who must build an executive that controls the processes represented at this level. A PAT for the level 1 flow model of *SafeHome* software is shown in Figure 7.5.

The CSPEC describes the behavior of the system, but it gives us no information about the inner working of the processes that are activated as a result of this behavior. The modeling notation that provides this information is discussed in Section 7.2.4.

#### 7.2.4 The Process Specification

The *process specification* (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. The content of the process specification can

**FIGURE 7.5**

Process activation table for  
*SafeHome*  
security  
function

input events	0	0	0	0	1	0
sensor event	0	0	1	1	0	0
blink flag	0	0	1	1	0	0
start stop switch	0	1	0	0	0	0
display action status complete	0	0	0	1	0	0
in-progress	0	0	1	0	0	0
time out	0	0	0	0	0	1
output	0	0	0	0	1	0
alarm signal	0	0	0	0	1	0
process activation	0	1	0	0	1	1
monitor and control system	0	1	0	0	0	0
activate/deactivate system	1	0	1	1	1	1
display messages and status	1	0	0	1	0	1
interact with user	1	0	0	1	0	1

## SAFEHOME



### Data Flow Modeling

**The scene:** Jamie's cubicle, after the last requirements gathering meeting has concluded.

**The players:** Jamie, Vinod, and Ed—all members of the *SafeHome* software engineering team.

#### The conversation:

(Jamie has sketched out the models shown in Figures 7.1 through 7.5 and is showing them to Ed and Vinod.)

**Jamie:** I took a software engineering course in college, and they taught us this stuff. The Prof said it's a bit old-fashioned, but you know what, it helps me to clarify things.

**Ed:** That's cool. But I don't see any classes or objects here.

**Jamie:** No . . . this is just a flow model with a little behavioral stuff thrown in.

**Vinod:** So these DFDs represent an I-P-O view of the software, right.

**Ed:** I-P-O?

**Vinod:** Input-process-output. The DFDs are actually pretty intuitive . . . if you look at 'em for a moment, they show how data objects flow through the system and get transformed as they go.

**Ed:** Looks like we could convert every bubble into an executable component . . . at least at the lowest level of the DFD.

**Jamie:** That's the cool part, you can. In fact, there's a way to translate the DFDs into an design architecture.

**Ed:** Really?

**Jamie:** Yeah, but first we've got to develop a complete requirements model and this isn't it.

**Vinod:** Well, it's a first step, but we're going to have to address class-based elements and also behavioral aspects, although the state diagram and PAT does some of that.

**Ed:** We've got a lot work to do and not much time to do it.  
(Doug—the software engineering manager—walks into the cubical.)

**Doug:** So the next few days will be spent developing the requirements model, huh?

**Jamie (looking proud):** We've already begun.

**Doug:** Good, we've got a lot of work to do and not much time to do it.

(The three software engineers look at one another and smile.)



The PSPEC is a “mini-specification” for each transform at the lowest refined level of a DFD.

include narrative text, a program design language (PDL) description<sup>5</sup> of the process algorithm, mathematical equations, tables, or UML activity diagrams. By providing a PSPEC to accompany each bubble in the flow model, you can create a “mini-spec” that serves as a guide for design of the software component that will implement the bubble.

To illustrate the use of the PSPEC, consider the *process password* transform represented in the flow model for *SafeHome* (Figure 7.2). The PSPEC for this function might take the form:

**PSPEC: process password (at control panel).** The *process password* transform performs password validation at the control panel for the *SafeHome* security function. *Process password* receives a four-digit password from the *interact with user* function. The password is first compared to the master password stored within the system. If the master password matches, <valid id message = true> is passed to the *message and status display* function. If the master password does not match, the four digits are compared to a table of secondary passwords (these may be assigned to house guests and/or workers who require entry to the home when the owner is not present). If the password matches an entry within the table, <valid id message = true> is passed to the *message and status display function*. If there is no match, <valid id message = false> is passed to the message and status display function.

If additional algorithmic detail is desired at this stage, a program design language representation may also be included as part of the PSPEC. However, many believe that the PDL version should be postponed until component design commences.

## SOFTWARE TOOLS



### Structured Analysis

**Objective:** Structured analysis tools allow a software engineer to create data models, flow models, and behavioral models in a manner that enables consistency and continuity checking and easy editing and extension. Models created using these tools provide the software engineer with insight into the analysis representation and help to eliminate errors before they propagate into design, or worse, into implementation itself.

**Mechanics:** Tools in this category use a “data dictionary” as the central database for the description of all data objects. Once entries in the dictionary are defined, entity-relationship diagrams can be created and object hierarchies can be developed. Data flow diagramming features allow easy creation of this graphical model and also provide features for the creation of PSPECs and CSPECs. Analysis tools also enable the software

engineer to create behavioral models using the state diagram as the operative notation.

#### Representative Tools:<sup>6</sup>

*MacA&D*, *WinA&D*, developed by Excel software ([www.excelsoftware.com](http://www.excelsoftware.com)), provides a set of simple and inexpensive analysis and design tools for Macs and Windows machines.

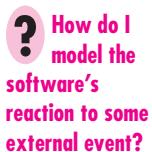
*MetaCASE Workbench*, developed by MetaCase Consulting ([www.metacase.com](http://www.metacase.com)), is a metatool used to define an analysis or design method (including structured analysis) and its concepts, rules, notations, and generators.

*System Architect*, developed by Popkin Software ([www.popkin.com](http://www.popkin.com)) provides a broad range of analysis and design tools including tools for data modeling and structured analysis.

<sup>5</sup> Program design language (PDL) mixes programming language syntax with narrative text to provide procedural design detail. PDL is discussed briefly in Chapter 10.

<sup>6</sup> Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

## 7.3 CREATING A BEHAVIORAL MODEL



The modeling notation that I have discussed to this point represents static elements of the requirements model. It is now time to make a transition to the dynamic behavior of the system or product. To accomplish this, you can represent the behavior of the system as a function of specific events and time.

The *behavioral model* indicates how software will respond to external events or stimuli. To create the model, you should perform the following steps:

1. Evaluate all use cases to fully understand the sequence of interaction within the system.
2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.
3. Create a sequence for each use case.
4. Build a state diagram for the system.
5. Review the behavioral model to verify accuracy and consistency.

Each of these steps is discussed in the sections that follow.

### 7.3.1 Identifying Events with the Use Case

In Chapter 6 you learned that the use case represents a sequence of activities that involves actors and the system. In general, an event occurs whenever the system and an actor exchange information. In Section 7.2.3, I indicated that an event is *not* the information that has been exchanged, but rather the fact that information has been exchanged.

A use case is examined for points of information exchange. To illustrate, we reconsider the use case for a portion of the *SafeHome* security function.

The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

The underlined portions of the use case scenario indicate events. An actor should be identified for each event; the information that is exchanged should be noted, and any conditions or constraints should be listed.

As an example of a typical event, consider the underlined use case phrase “homeowner uses the keypad to key in a four-digit password.” In the context of the requirements model, the object, **Homeowner**,<sup>7</sup> transmits an event to the object **ControlPanel**. The event might be called *password entered*. The information

<sup>7</sup> In this example, we assume that each user (homeowner) that interacts with *SafeHome* has an identifying password and is therefore a legitimate object.

transferred is the four digits that constitute the password, but this is not an essential part of the behavioral model. It is important to note that some events have an explicit impact on the flow of control of the use case, while others have no direct impact on the flow of control. For example, the event *password entered* does not explicitly change the flow of control of the use case, but the results of the event *password compared* (derived from the interaction “password is compared with the valid password stored in the system”) will have an explicit impact on the information and control flow of the *SafeHome* software.

Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events (e.g., **Homeowner** generates the *password entered* event) or recognizing events that have occurred elsewhere (e.g., **ControlPanel** recognizes the binary result of the *password compared* event).

### 7.3.2 State Representations

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its function.<sup>8</sup>

#### KEY POINT

The system has states that represent specific externally observable behavior; a class has states that represent its behavior as the system performs its functions.

The state of a class takes on both passive and active characteristics [Cha93]. A *passive state* is simply the current status of all of an object’s attributes. For example, the passive state of the class **Player** (in the video game application discussed in Chapter 6) would include the current **position** and **orientation** attributes of **Player** as well as other features of **Player** that are relevant to the game (e.g., an attribute that indicates **magic wishes remaining**). The *active state* of an object indicates the current status of the object as it undergoes a continuing transformation or processing. The class **Player** might have the following active states: *moving*, *at rest*, *injured*, *being cured*, *trapped*, *lost*, and so forth. An event (sometimes called a *trigger*) must occur to force an object to make a transition from one active state to another.

Two different behavioral representations are discussed in the paragraphs that follow. The first indicates how an individual class changes state based on external events and the second shows the behavior of the software as a function of time.

**State diagrams for analysis classes.** One component of a behavioral model is a UML state diagram<sup>9</sup> that represents active states for each class and the events (triggers) that cause changes between these active states. Figure 7.6 illustrates a state diagram for the **ControlPanel** object in the *SafeHome* security function.

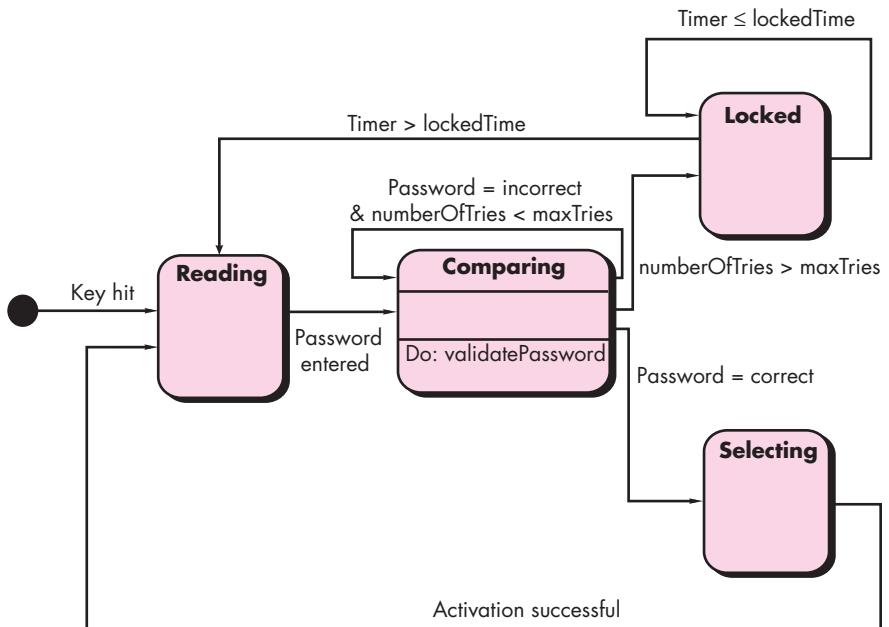
Each arrow shown in Figure 7.6 represents a transition from one active state of an object to another. The labels shown for each arrow represent the event that

<sup>8</sup> The state diagrams presented in Chapter 6 and in Section 7.3.2 depict the state of the system. Our discussion in this section will focus on the state of each class within the analysis model.

<sup>9</sup> If you are unfamiliar with UML, a brief introduction to this important modeling notation is presented in Appendix 1.

**FIGURE 7.6**

State diagram  
for the  
ControlPanel  
class



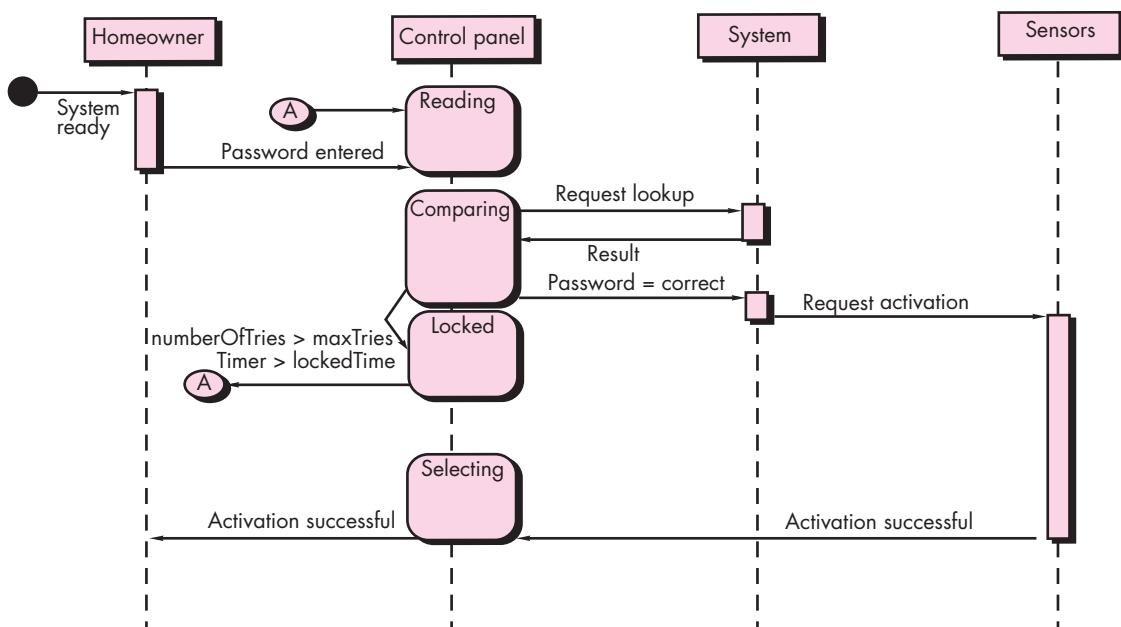
triggers the transition. Although the active state model provides useful insight into the “life history” of an object, it is possible to specify additional information to provide more depth in understanding the behavior of an object. In addition to specifying the event that causes the transition to occur, you can specify a guard and an action [Cha93]. A *guard* is a Boolean condition that must be satisfied in order for the transition to occur. For example, the guard for the transition from the “reading” state to the “comparing” state in Figure 7.6 can be determined by examining the use case:

```
if (password input = 4 digits) then compare to stored password
```

In general, the guard for a transition usually depends upon the value of one or more attributes of an object. In other words, the guard depends on the passive state of the object.

An *action* occurs concurrently with the state transition or as a consequence of it and generally involves one or more operations (responsibilities) of the object. For example, the action connected to the *password entered* event (Figure 7.6) is an operation named *validatePassword()* that accesses a **password** object and performs a digit-by-digit comparison to validate the entered password.

**Sequence diagrams.** The second type of behavioral representation, called a *sequence diagram* in UML, indicates how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler

**FIGURE 7.7** Sequence diagram (partial) for the *SafeHome* security function

creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time. In essence, the sequence diagram is a short-hand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.

Figure 7.7 illustrates a partial sequence diagram for the *SafeHome* security function. Each of the arrows represents an event (derived from a use case) and indicates how the event channels behavior between *SafeHome* objects. Time is measured vertically (downward), and the narrow vertical rectangles represent time spent in processing an activity. States may be shown along a vertical time line.

The first event, *system ready*, is derived from the external environment and channels behavior to the **Homeowner** object. The homeowner enters a password. A *request lookup* event is passed to **System**, which looks up the password in a simple database and returns a *result* (*found* or *not found*) to **ControlPanel** (now in the *comparing* state). A valid password results in a *password=correct* event to **System**, which activates **Sensors** with a *request activation* event. Ultimately, control is passed back to the homeowner with the *activation successful* event.

Once a complete sequence diagram has been developed, all of the events that cause transitions between system objects can be collated into a set of input events and output events (from an object). This information is useful in the creation of an effective design for the system to be built.

### KEY POINT

Unlike a state diagram that represents behavior without noting the classes involved, a sequence diagram represents behavior, by describing how classes move from state to state.

## SOFTWARE TOOLS



### Generalized Analysis Modeling in UML

**Objective:** Analysis modeling tools provide the capability to develop scenario-based models, class-based models, and behavioral models using UML notation.

**Mechanics:** Tools in this category support the full range of UML diagrams required to build an analysis model (these tools also support design modeling). In addition to diagramming, tools in this category (1) perform consistency and correctness checks for all UML diagrams, (2) provide links for design and code generation, (3) build a database that enables the management and assessment of large UML models required for complex systems.

#### Representative Tools:<sup>10</sup>

The following tools support a full range of UML diagrams required for analysis modeling:

ArgoUML is an open source tool available at [argouml.tigris.org](http://argouml.tigris.org).

Enterprise Architect, developed by Sparx Systems ([www.sparxsystems.com.au](http://www.sparxsystems.com.au)).

PowerDesigner, developed by Sybase ([www.sybase.com](http://www.sybase.com)).

Rational Rose, developed by IBM (Rational) ([www01.ibm.com/software/rational/](http://www01.ibm.com/software/rational/)).

System Architect, developed by Popkin Software ([www.popkin.com](http://www.popkin.com)).

UML Studio, developed by Pragsoft Corporation ([www.pragsoft.com](http://www.pragsoft.com)).

Visio, developed by Microsoft ([www.microsoft.com](http://www.microsoft.com)).

Visual UML, developed by Visual Object Modelers ([www.visualuml.com](http://www.visualuml.com)).

## 7.4 PATTERNS FOR REQUIREMENTS MODELING

Software patterns are a mechanism for capturing domain knowledge in a way that allows it to be reapplied when a new problem is encountered. In some cases, the domain knowledge is applied to a new problem within the same application domain. In other cases, the domain knowledge captured by a pattern can be applied by analogy to a completely different application domain.

The original author of an analysis pattern does not “create” the pattern, but, rather, *discovers* it as requirements engineering work is being conducted. Once the pattern has been discovered, it is documented by describing “explicitly the general problem to which the pattern is applicable, the prescribed solution, assumptions and constraints of using the pattern in practice, and often some other information about the pattern, such as the motivation and driving forces for using the pattern, discussion of the pattern’s advantages and disadvantages, and references to some known examples of using that pattern in practical applications” [Dev01].

In Chapter 5, I introduced the concept of analysis patterns and indicated that these patterns represent a solution that often incorporates a class, a function, or a behavior within the application domain. The pattern can be reused when performing requirements modeling for an application within a domain.<sup>11</sup> Analysis patterns are stored in a repository so that members of the software team can use search facilities to find and reuse them. Once an appropriate pattern is selected, it is integrated into the requirements model by reference to the pattern name.

<sup>10</sup> Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

<sup>11</sup> An in-depth discussion of the use of patterns during software design is presented in Chapter 12.

### 7.4.1 Discovering Analysis Patterns

The requirements model is comprised of a wide variety of elements: scenario-based (use cases), data-oriented (the data model), class-based, flow-oriented, and behavioral. Each of these elements examines the problem from a different perspective, and each provides an opportunity to discover patterns that may occur throughout an application domain, or by analogy, across different application domains.

The most basic element in the description of a requirements model is the use case. In the context of this discussion, a coherent set of use cases may serve as the basis for discovering one or more analysis patterns. A *semantic analysis pattern* (SAP) “is a pattern that describes a small set of coherent use cases that together describe a basic generic application” [Fer00].

Consider the following preliminary use case for software required to control and monitor a rear-view camera and proximity sensor for an automobile:

#### **Use case: Monitor reverse motion**

**Description:** When the vehicle is placed in *reverse* gear, the control software enables a video feed from a rear-placed video camera to the dashboard display. The control software superimposes a variety of distance and orientation lines on the dashboard display so that the vehicle operator can maintain orientation as the vehicle moves in reverse. The control software also monitors a proximity sensor to determine whether an object is inside 10 feet of the rear of the vehicle. It will automatically break the vehicle if the proximity sensor indicates an object within  $x$  feet of the rear of the vehicle, where  $x$  is determined based on the speed of the vehicle.

This use case implies a variety of functionality that would be refined and elaborated (into a coherent set of use cases) during requirements gathering and modeling. Regardless of how much elaboration is accomplished, the use cases suggest a simple, yet widely applicable SAP—the software-based monitoring and control of sensors and actuators in a physical system. In this case, the “sensors” provide information about proximity and video information. The “actuator” is the breaking system of the vehicle (invoked if an object is very close to the vehicle). But in a more general case, a widely applicable pattern is discovered.

Software in many different application domains is required to monitor sensors and control physical actuators. It follows that an analysis pattern that describes generic requirements for this capability could be used widely. The pattern, called **Actuator-Sensor**, would be applicable as part of the requirements model for *SafeHome* and is discussed in Section 7.4.2, which follows.

### 7.4.2 A Requirements Pattern Example: Actuator-Sensor<sup>12</sup>

One of the requirements of the *SafeHome* security function is the ability to monitor security sensors (e.g., break-in sensors, fire, smoke or CO sensors, water sensors).

---

12 This section has been adapted from [Kon02] with the permission of the authors.

Internet-based extensions to *SafeHome* will require the ability to control the movement (e.g., pan, zoom) of a security camera within a residence. The implication—*SafeHome* software must manage various sensors and “actuators” (e.g., camera control mechanisms).

Konrad and Cheng [Kon02] have suggested a requirements pattern named **Actuator-Sensor** that provides useful guidance for modeling this requirement within *SafeHome* software. An abbreviated version of the **Actuator-Sensor** pattern, originally developed for automotive applications, follows.

### Pattern Name. **Actuator-Sensor**

**Intent.** Specify various kinds of sensors and actuators in an embedded system.

**Motivation.** Embedded systems usually have various kinds of sensors and actuators. These sensors and actuators are all either directly or indirectly connected to a control unit. Although many of the sensors and actuators look quite different, their behavior is similar enough to structure them into a pattern. The pattern shows how to specify the sensors and actuators for a system, including attributes and operations. The **Actuator-Sensor** pattern uses a *pull* mechanism (explicit request for information) for **PassiveSensors** and a *push* mechanism (broadcast of information) for the **ActiveSensors**.

### Constraints

- Each passive sensor must have some method to read sensor input and attributes that represent the sensor value.
- Each active sensor must have capabilities to broadcast update messages when its value changes.
- Each active sensor should send a *life tick*, a status message issued within a specified time frame, to detect malfunctions.
- Each actuator must have some method to invoke the appropriate response determined by the **ComputingComponent**.
- Each sensor and actuator should have a function implemented to check its own operation state.
- Each sensor and actuator should be able to test the validity of the values received or sent and set its operation state if the values are outside of the specifications.

**Applicability.** Useful in any system in which multiple sensors and actuators are present.

**Structure.** A UML class diagram for the **Actuator-Sensor** pattern is shown in Figure 7.8. **Actuator**, **PassiveSensor**, and **ActiveSensor** are abstract classes and denoted in italics. There are four different types of sensors and actuators in this pattern.

**FIGURE 7.8** UML sequence diagram for the Actuator-Sensor pattern. *Source:* Adapted from [Kon02] with permission.

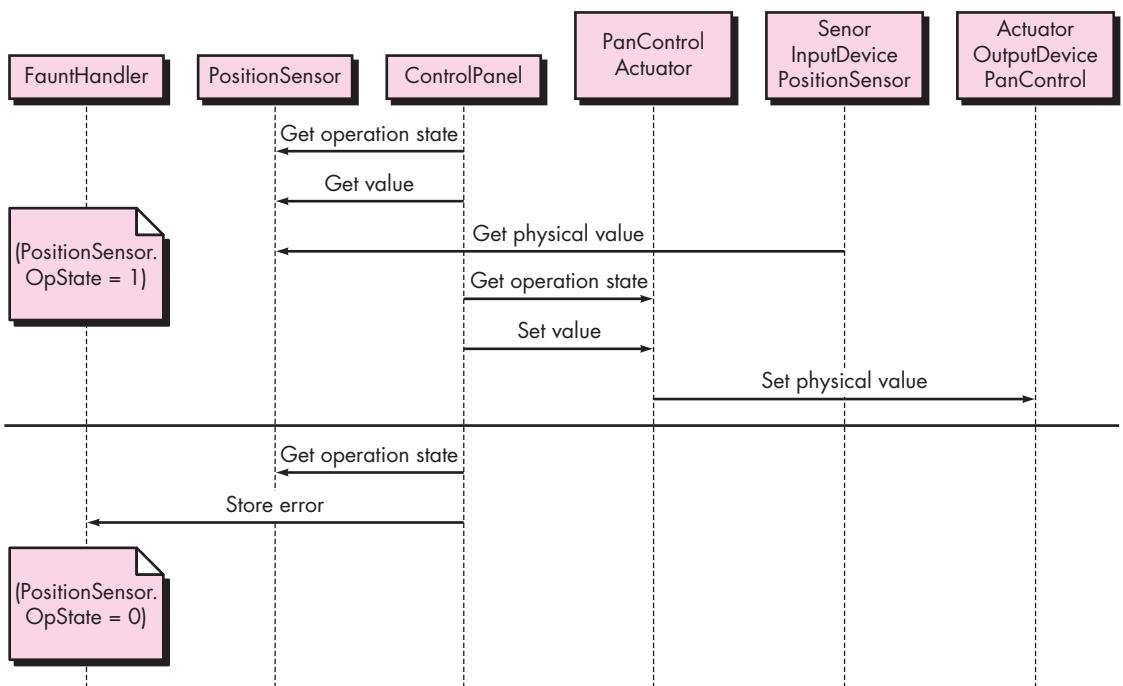


The **Boolean**, **Integer**, and **Real** classes represent the most common types of sensors and actuators. The complex classes are sensors or actuators that use values that cannot be easily represented in terms of primitive data types, such as a radar device. Nonetheless, these devices should still inherit the interface from the abstract classes since they should have basic functionalities such as querying the operation states.

**Behavior.** Figure 7.9 presents a UML sequence diagram for an example of the **Actuator-Sensor** pattern as it might be applied for the *SafeHome* function that controls the positioning (e.g., pan, zoom) of a security camera. Here, the **ControlPanel**<sup>13</sup> queries a sensor (a passive position sensor) and an actuator (pan control) to check the operation state for diagnostic purposes before reading or setting a value. The messages *Set Physical Value* and *Get Physical Value* are not messages between objects. Instead, they describe the interaction between the physical devices of the system and their software counterparts. In the lower part of the diagram, below the horizontal line, the **PositionSensor** reports that the operation state is zero. The **ComputingComponent** (represented as **ControlPanel**) then sends the error code for a position sensor failure to the **FaultHandler** that will decide how this error affects the system and what actions are required. It gets the data from the sensors and computes the required response for the actuators.

<sup>13</sup> The original pattern uses the generic phrase **ComputingComponent**.

**FIGURE 7.9** UML Class diagram for the Actuator-Sensor pattern. *Source:* Reprinted from [Kon02] with permission.



**Participants.** This section of the patterns description “itemizes the classes/objects that are included in the requirements pattern” [Kon02] and describes the responsibilities of each class/object (Figure 7.8). An abbreviated list follows:

- **PassiveSensor abstract:** Defines an interface for passive sensors.
- **PassiveBooleanSensor:** Defines passive Boolean sensors.
- **PassiveIntegerSensor:** Defines passive integer sensors.
- **PassiveRealSensor:** Defines passive real sensors.
- **ActiveSensor abstract:** Defines an interface for active sensors.
- **ActiveBooleanSensor:** Defines active Boolean sensors.
- **ActiveIntegerSensor:** Defines active integer sensors.
- **ActiveRealSensor:** Defines active real sensors.
- **Actuator abstract:** Defines an interface for actuators.
- **BooleanActuator:** Defines Boolean actuators.
- **IntegerActuator:** Defines integer actuators.
- **RealActuator:** Defines real actuators.
- **ComputingComponent:** The central part of the controller; it gets the data from the sensors and computes the required response for the actuators.

- **ActiveComplexSensor:** Complex active sensors have the basic functionality of the abstract **ActiveSensor** class, but additional, more elaborate, methods and attributes need to be specified.
- **PassiveComplexSensor:** Complex passive sensors have the basic functionality of the abstract **PassiveSensor** class, but additional, more elaborate, methods and attributes need to be specified.
- **ComplexActuator:** Complex actuators also have the base functionality of the abstract **Actuator** class, but additional, more elaborate methods and attributes need to be specified.

**Collaborations.** This section describes how objects and classes interact with one another and how each carries out its responsibilities.

- When the **ComputingComponent** needs to update the value of a **PassiveSensor**, it queries the sensors, requesting the value by sending the appropriate message.
- **ActiveSensors** are not queried. They initiate the transmission of sensor values to the computing unit, using the appropriate method to set the value in the **ComputingComponent**. They send a life tick at least once during a specified time frame in order to update their timestamps with the system clock's time.
- When the **ComputingComponent** needs to set the value of an actuator, it sends the value to the actuator.
- The **ComputingComponent** can query and set the operation state of the sensors and actuators using the appropriate methods. If an operation state is found to be zero, then the error is sent to the **FaultHandler**, a class that contains methods for handling error messages, such as starting a more elaborate recovery mechanism or a backup device. If no recovery is possible, then the system can only use the last known value for the sensor or the default value.
- The **ActiveSensors** offer methods to add or remove the addresses or address ranges of the components that want to receive the messages in case of a value change.

### Consequences

1. Sensor and actuator classes have a common interface.
2. Class attributes can only be accessed through messages, and the class decides whether or not to accept the message. For example, if a value of an actuator is set above a maximum value, then the actuator class may not accept the message, or it might use a default maximum value.
3. The complexity of the system is potentially reduced because of the uniformity of interfaces for actuators and sensors.

The requirements pattern description might also provide references to other related requirements and design patterns.

## 7.5 REQUIREMENTS MODELING FOR WEBAPPS<sup>14</sup>

Web developers are often skeptical when the idea of requirements analysis for WebApps is suggested. “After all,” they argue, “the Web development process must be agile, and analysis is time consuming. It’ll slow us down just when we need to be designing and building the WebApp.”

Requirements analysis does take time, but solving the wrong problem takes even more time. The question for every WebApp developer is simple—are you sure you understand the requirements of the problem? If the answer is an unequivocal “yes,” then it may be possible to skip requirements modeling, but if the answer is “no,” then requirements modeling should be performed.

### 7.5.1 How Much Analysis Is Enough?

The degree to which requirements modeling for WebApps is emphasized depends on the following factors:

- Size and complexity of WebApp increment.
- Number of stakeholders (analysis can help to identify conflicting requirements coming from different sources).
- Size of the WebApp team.
- Degree to which members of the WebApp team have worked together before (analysis can help develop a common understanding of the project).
- Degree to which the organization’s success is directly dependent on the success of the WebApp.

The converse of the preceding points is that as the project becomes smaller, the number of stakeholders fewer, the development team more cohesive, and the application less critical, it is reasonable to apply a more lightweight analysis approach.

Although it is a good idea to analyze the problem *before* beginning design, it is not true that *all* analysis must precede *all* design. In fact, the design of a specific part of the WebApp only demands an analysis of those requirements that affect only that part of the WebApp. As an example from *SafeHome*, you could validly design the overall website aesthetics (layouts, color schemes, etc.) without having analyzed the functional requirements for e-commerce capabilities. You only need to analyze that part of the problem that is relevant to the design work for the increment to be delivered.

---

14 This section has been adapted from Pressman and Lowe [Pre08] with permission.

### 7.5.2 Requirements Modeling Input

An agile version of the generic software process discussed in Chapter 2 can be applied when WebApps are engineered. The process incorporates a communication activity that identifies stakeholders and user categories, the business context, defined informational and applicative goals, general WebApp requirements, and usage scenarios—information that becomes input to requirements modeling. This information is represented in the form of natural language descriptions, rough outlines, sketches, and other informal representations.

Analysis takes this information, structures it using a formally defined representation scheme (where appropriate), and then produces more rigorous models as an output. The requirements model provides a detailed indication of the true structure of the problem and provides insight into the shape of the solution.

The *SafeHome ACS-DCV* (camera surveillance) function was introduced in Chapter 6. When it was introduced, this function seemed relatively clear and was described in some detail as part of a use case (Section 6.2.1). However, a reexamination of the use case might uncover information that is missing, ambiguous, or unclear.

Some aspects of this missing information would naturally emerge during the design. Examples might include the specific layout of the function buttons, their aesthetic look and feel, the size of snapshot views, the placement of camera views and the house floor plan, or even minutiae such as the maximum and minimum length of passwords. Some of these aspects are design decisions (such as the layout of the buttons) and others are requirements (such as the length of the passwords) that don't fundamentally influence early design work.

But some missing information might actually influence the overall design itself and relate more to an actual understanding of the requirements. For example:

- Q<sub>1</sub>: What output video resolution is provided by *SafeHome* cameras?
- Q<sub>2</sub>: What occurs if an alarm condition is encountered while the camera is being monitored?
- Q<sub>3</sub>: How does the system handle cameras that can be panned and zoomed?
- Q<sub>4</sub>: What information should be provided along with the camera view? (For example, location? time/date? last previous access?)

None of these questions were identified or considered in the initial development of the use case, and yet, the answers could have a substantial effect on different aspects of the design.

Therefore, it is reasonable to conclude that although the communication activity provides a good foundation for understanding, requirements analysis refines this understanding by providing additional interpretation. As the problem structure is delineated as part of the requirements model, questions invariably arise. It is these questions that fill in the gaps—or in some cases, actually help us to find the gaps in the first place.

To summarize, the inputs to the requirements model will be the information collected during the communication activity—anything from an informal e-mail to a detailed project brief complete with comprehensive usage scenarios and product specifications.

### 7.5.3 Requirements Modeling Output

Requirements analysis provides a disciplined mechanism for representing and evaluating WebApp content and function, the modes of interaction that users will encounter, and the environment and infrastructure in which the WebApp resides.

Each of these characteristics can be represented as a set of models that allow the WebApp requirements to be analyzed in a structured manner. While the specific models depend largely upon the nature of the WebApp, there are five main classes of models:

- **Content model**—identifies the full spectrum of content to be provided by the WebApp. Content includes text, graphics and images, video, and audio data.
- **Interaction model**—describes the manner in which users interact with the WebApp.
- **Functional model**—defines the operations that will be applied to WebApp content and describes other processing functions that are independent of content but necessary to the end user.
- **Navigation model**—defines the overall navigation strategy for the WebApp.
- **Configuration model**—describes the environment and infrastructure in which the WebApp resides.

You can develop each of these models using a representation scheme (often called a “language”) that allows its intent and structure to be communicated and evaluated easily among members of the Web engineering team and other stakeholders. As a consequence, a list of key issues (e.g., errors, omissions, inconsistencies, suggestions for enhancement or modification, points of clarification) are identified and acted upon.

### 7.5.4 Content Model for WebApps

The content model contains structural elements that provide an important view of content requirements for a WebApp. These structural elements encompass content objects and all analysis classes—user-visible entities that are created or manipulated as a user interacts with the WebApp.<sup>15</sup>

Content can be developed prior to the implementation of the WebApp, while the WebApp is being built, or long after the WebApp is operational. In every case, it is

---

15 Analysis classes were discussed in Chapter 6.

incorporated via navigational reference into the overall WebApp structure. A *content object* might be a textual description of a product, an article describing a news event, an action photograph taken at a sporting event, a user's response on a discussion forum, an animated representation of a corporate logo, a short video of a speech, or an audio overlay for a collection of presentation slides. The content objects might be stored as separate files, embedded directly into Web pages, or obtained dynamically from a database. In other words, a content object is any item of cohesive information that is to be presented to an end user.

Content objects can be determined directly from use cases by examining the scenario description for direct and indirect references to content. For example, a WebApp that supports *SafeHome* is established at **SafeHomeAssured.com**. A use case, *Purchasing Select SafeHome Components*, describes the scenario required to purchase a *SafeHome* component and contains the sentence:

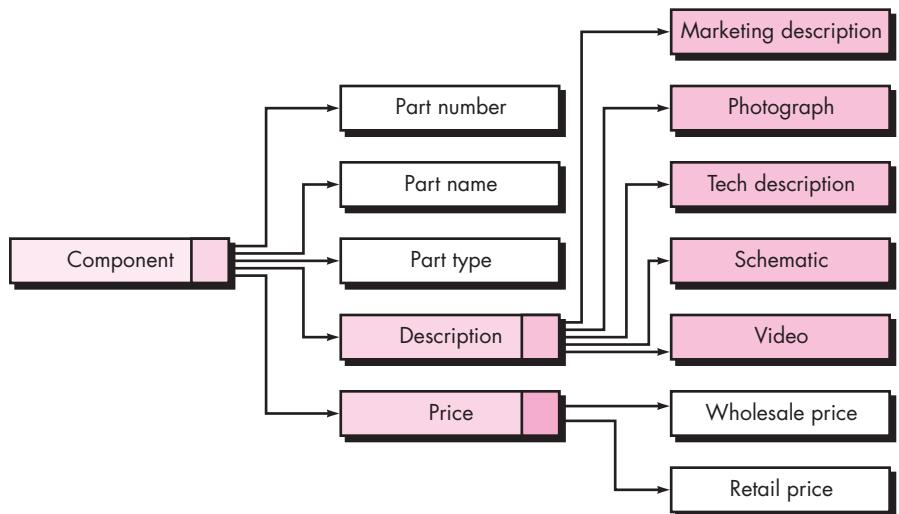
I will be able to get descriptive and pricing information for each product component.

The content model must be capable of describing the content object **Component**. In many instances, a simple list of content objects, coupled with a brief description of each object, is sufficient to define the requirements for content that must be designed and implemented. However, in some cases, the content model may benefit from a richer analysis that graphically illustrates the relationships among content objects and/or the hierarchy of content maintained by a WebApp.

For example, consider the *data tree* [Sri01] created for a **SafeHomeAssured.com** component shown in Figure 7.10. The tree represents a hierarchy of information that is used to describe a component. Simple or composite data items (one or more data

**FIGURE 7.10**

Data tree for  
a **SafeHome-  
Assured.com**  
component



values) are represented as unshaded rectangles. Content objects are represented as shaded rectangles. In the figure, **description** is defined by five content objects (the shaded rectangles). In some cases, one or more of these objects would be further refined as the data tree expands.

A data tree can be created for any content that is composed of multiple content objects and data items. The data tree is developed in an effort to define hierarchical relationships among content objects and to provide a means for reviewing content so that omissions and inconsistencies are uncovered before design commences. In addition, the data tree serves as the basis for content design.

### 7.5.5 Interaction Model for WebApps

The vast majority of WebApps enable a “conversation” between an end user and application functionality, content, and behavior. This conversation can be described using an *interaction* model that can be composed of one or more of the following elements: (1) use cases, (2) sequence diagrams, (3) state diagrams,<sup>16</sup> and/or (4) user interface prototypes.

In many instances, a set of use cases is sufficient to describe the interaction at an analysis level (further refinement and detail will be introduced during design). However, when the sequence of interaction is complex and involves multiple analysis classes or many tasks, it is sometimes worthwhile to depict it using a more rigorous diagrammatic form.

The layout of the user interface, the content it presents, the interaction mechanisms it implements, and the overall aesthetic of the user-WebApp connections have much to do with user satisfaction and the overall success of the WebApp. Although it can be argued that the creation of a user interface prototype is a design activity, it is a good idea to perform it during the creation of the analysis model. The sooner that a physical representation of a user interface can be reviewed, the higher the likelihood that end users will get what they want. The design of user interfaces is discussed in detail in Chapter 11.

Because WebApp construction tools are plentiful, relatively inexpensive, and functionally powerful, it is best to create the interface prototype using such tools. The prototype should implement the major navigational links and represent the overall screen layout in much the same way that it will be constructed. For example, if five major system functions are to be provided to the end user, the prototype should represent them as the user will see them upon first entering the WebApp. Will graphical links be provided? Where will the navigation menu be displayed? What other information will the user see? Questions like these should be answered by the prototype.

---

<sup>16</sup> Sequence diagrams and state diagrams are modeled using UML notation. State diagrams are described in Section 7.3. See Appendix 1 for additional detail.

### 7.5.6 Functional Model for WebApps

Many WebApps deliver a broad array of computational and manipulative functions that can be associated directly with content (either using it or producing it) and that are often a major goal of user-WebApp interaction. For this reason, functional requirements must be analyzed, and when necessary, modeled.

The *functional model* addresses two processing elements of the WebApp, each representing a different level of procedural abstraction: (1) user-observable functionality that is delivered by the WebApp to end users, and (2) the operations contained within analysis classes that implement behaviors associated with the class.

User-observable functionality encompasses any processing functions that are initiated directly by the user. For example, a financial WebApp might implement a variety of financial functions (e.g., a college tuition savings calculator or a retirement savings calculator). These functions may actually be implemented using operations within analysis classes, but from the point of view of the end user, the function (more correctly, the data provided by the function) is the visible outcome.

At a lower level of procedural abstraction, the requirements model describes the processing to be performed by analysis class operations. These operations manipulate class attributes and are involved as classes collaborate with one another to accomplish some required behavior.

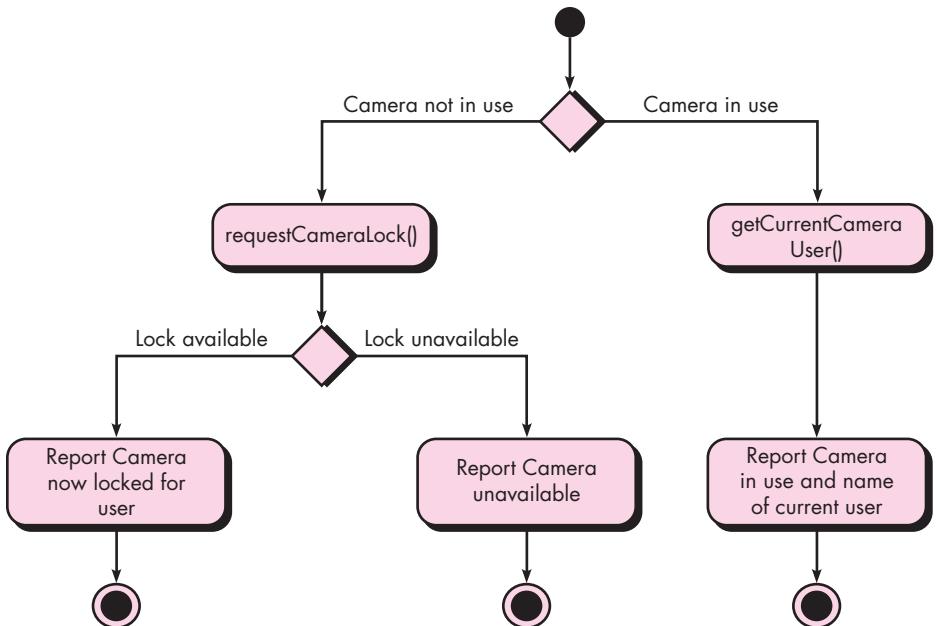
Regardless of the level of procedural abstraction, the UML activity diagram can be used to represent processing details. At the analysis level, activity diagrams should be used only where the functionality is relatively complex. Much of the complexity of many WebApps occurs not in the functionality provided, but rather with the nature of the information that can be accessed and the ways in which this can be manipulated.

An example of relatively complex functionality for **SafeHomeAssured.com** is addressed by a use case entitled *Get recommendations for sensor layout for my space*. The user has already developed a layout for the space to be monitored, and in this use case, selects that layout and requests recommended locations for sensors within the layout. **SafeHomeAssured.com** responds with a graphical representation of the layout with additional information on the recommended locations for sensors. The interaction is quite simple, the content is somewhat more complex, but the underlying functionality is very sophisticated. The system must undertake a relatively complex analysis of the floor layout in order to determine the optimal set of sensors. It must examine room dimensions, the location of doors and windows, and coordinate these with sensor capabilities and specifications. No small task! A set of activity diagrams can be used to describe processing for this use case.

The second example is the use case *Control cameras*. In this use case, the interaction is relatively simple, but there is the potential for complex functionality, given that this “simple” operation requires complex communication with devices located remotely and accessible across the Internet. A further possible complication relates

**FIGURE 7.11**

Activity diagram for the `takeControlOfCamera()` operation



to negotiation of control when multiple authorized people attempt to monitor and/or control a single sensor at the same time.

Figure 7.11 depicts an activity diagram for the `takeControlOfCamera()` operation that is part of the **Camera** analysis class used within the *Control cameras* use case. It should be noted that two additional operations are invoked with the procedural flow: `requestCameraLock()`, which tries to lock the camera for this user, and `getCurrentCameraUser()`, which retrieves the name of the user who is currently controlling the camera. The construction details indicating how these operations are invoked and the interface details for each operation are not considered until WebApp design commences.

### 7.5.7 Configuration Models for WebApps

In some cases, the configuration model is nothing more than a list of server-side and client-side attributes. However, for more complex WebApps, a variety of configuration complexities (e.g., distributing load among multiple servers, caching architectures, remote databases, multiple servers serving various objects on the same Web page) may have an impact on analysis and design. The UML *deployment diagram* can be used in situations in which complex configuration architectures must be considered.

For **SafeHomeAssured.com** the public content and functionality should be specified to be accessible across all major Web clients (i.e., those with more than

1 percent market share or greater<sup>17</sup>). Conversely, it may be acceptable to restrict the more complex control and monitoring functionality (which is only accessible to **Homeowner** users) to a smaller set of clients. The configuration model for **SafeHomeAssured.com** will also specify interoperability with existing product databases and monitoring applications.

### 7.5.8 Navigation Modeling

Navigation modeling considers how each user category will navigate from one WebApp element (e.g., content object) to another. The mechanics of navigation are defined as part of design. At this stage, you should focus on overall navigation requirements. The following questions should be considered:

- Should certain elements be easier to reach (require fewer navigation steps) than others? What is the priority for presentation?
- Should certain elements be emphasized to force users to navigate in their direction?
- How should navigation errors be handled?
- Should navigation to related groups of elements be given priority over navigation to a specific element?
- Should navigation be accomplished via links, via search-based access, or by some other means?
- Should certain elements be presented to users based on the context of previous navigation actions?
- Should a navigation log be maintained for users?
- Should a full navigation map or menu (as opposed to a single “back” link or directed pointer) be available at every point in a user’s interaction?
- Should navigation design be driven by the most commonly expected user behaviors or by the perceived importance of the defined WebApp elements?
- Can a user “store” his previous navigation through the WebApp to expedite future usage?
- For which user category should optimal navigation be designed?
- How should links external to the WebApp be handled? Overlaying the existing browser window? As a new browser window? As a separate frame?

These and many other questions should be asked and answered as part of navigation analysis.

---

<sup>17</sup> Determining market share for browsers is notoriously problematic and varies depending on which survey is used. Nevertheless, at the time of writing, Internet Explorer and Firefox are the only browsers that were reported in excess of 30 percent, and Mozilla, Opera, and Safari the only other ones consistently above 1 percent.

You and other stakeholders must also determine overall requirements for navigation. For example, will a “site map” be provided to give users an overview of the entire WebApp structure? Can a user take a “guided tour” that will highlight the most important elements (content objects and functions) that are available? Will a user be able to access content objects or functions based on defined attributes of those elements (e.g., a user might want to access all photographs of a specific building or all functions that allow computation of weight)?

## 7.6 SUMMARY

Flow-oriented models focus on the flow of data objects as they are transformed by processing functions. Derived from structured analysis, flow-oriented models use the data flow diagram, a modeling notation that depicts how input is transformed into output as data objects move through a system. Each software function that transforms data is described by a process specification or narrative. In addition to data flow, this modeling element also depicts control flow—a representation that illustrates how events affect the behavior of a system.

Behavioral modeling depicts dynamic behavior. The behavioral model uses input from scenario-based, flow-oriented, and class-based elements to represent the states of analysis classes and the system as a whole. To accomplish this, states are identified, the events that cause a class (or the system) to make a transition from one state to another are defined, and the actions that occur as transition is accomplished are also identified. State diagrams and sequence diagrams are the notation used for behavioral modeling.

Analysis patterns enable a software engineer to use existing domain knowledge to facilitate the creation of a requirements model. An analysis pattern describes a specific software feature or function that can be described by a coherent set of use cases. It specifies the intent of the pattern, the motivation for its use, constraints that limit its use, its applicability in various problem domains, the overall structure of the pattern, its behavior and collaborations, and other supplementary information.

Requirements modeling for WebApps can use most, if not all, of the modeling elements discussed in this book. However, these elements are applied within a set of specialized models that address content, interaction, function, navigation, and the client-server configuration in which the WebApp resides.

## PROBLEMS AND POINTS TO PONDER

- 7.1.** What is the fundamental difference between the structured analysis and object-oriented strategies for requirements analysis?
- 7.2.** In a data flow diagram, does an arrow represent a flow of control or something else?
- 7.3.** What is “information flow continuity” and how is it applied as a data flow diagram is refined?

- 7.4.** How is a grammatical parse used in the creation of a DFD?
- 7.5.** What is a control specification?
- 7.6.** Are a PSPEC and a use case the same thing? If not, explain the differences.
- 7.7.** There are two different types of “states” that behavioral models can represent. What are they?
- 7.8.** How does a sequence diagram differ from a state diagram. How are they similar?
- 7.9.** Suggest three requirements patterns for a modern mobile phone and write a brief description of each. Could these patterns be used for other devices. Provide an example.
- 7.10.** Select one of the patterns you developed in Problem 7.9 and develop a reasonably complete pattern description similar in content and style to the one presented in Section 7.4.2.
- 7.11.** How much analysis modeling do you think would be required for **SafeHomeAssured.com**? Would each of the model types described in Section 7.5.3 be required?
- 7.12.** What is the purpose of the interaction model for a WebApp?
- 7.13.** It could be argued that a WebApp functional model should be delayed until design. Present pros and cons for this argument.
- 7.14.** What is the purpose of a configuration model?
- 7.15.** How does the navigation model differ from the interaction model?

## FURTHER READINGS AND INFORMATION SOURCES

Dozens of books have been published on structured analysis. All cover the subject adequately, but only a few do a truly excellent job. DeMarco and Plauger (*Structured Analysis and System Specification*, Pearson, 1985) is a classic that remains a good introduction to the basic notation. Books by Kendall and Kendall (*Systems Analysis and Design*, 5th ed., Prentice-Hall, 2002), Hoffer et al. (*Modern Systems Analysis and Design*, Addison-Wesley, 3d ed., 2001), Davis and Yen (*The Information System Consultant's Handbook: Systems Analysis and Design*, CRC Press, 1998), and Modell (*A Professional's Guide to Systems Analysis*, 2d ed., McGraw-Hill, 1996) are worthwhile references. Yourdon's book (*Modern Structured Analysis*, Yourdon-Press, 1989) on the subject remains among the most comprehensive coverage published to date.

Behavioral modeling presents an important dynamic view of system behavior. Books by Wagner and his colleagues (*Modeling Software with Finite State Machines: A Practical Approach*, Auerbach, 2006) and Boerger and Staerk (*Abstract State Machines*, Springer, 2003) present thorough discussion of state diagrams and other behavioral representations.

The majority of books written about software patterns focus on software design. However, books by Evans (*Domain-Driven Design*, Addison-Wesley, 2003) and Fowler ([Fow03] and [Fow97]) address analysis patterns specifically.

An in-depth treatment of analysis modeling for WebApps is presented by Pressman and Lowe [Pre08]. Papers contained within an anthology edited by Murugesan and Desphande (*Web Engineering: Managing Diversity and Complexity of Web Application Development*, Springer, 2001) treat various aspects of WebApp requirements. In addition, the annual *Proceedings of the International Conference on Web Engineering* regularly addresses requirements modeling issues.

A wide variety of information sources on requirements modeling are available on the Internet. An up-to-date list of World Wide Web references that are relevant to analysis modeling can be found at the SEPA website: [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm).

## DESIGN CONCEPTS

### KEY CONCEPTS

abstraction	... 223
architecture	... 223
aspects	..... 228
cohesion	..... 227
data design	... 234
design process	. 219
functional independence	.. 227
good design	... 219
information hiding	..... 226

### QUICK LOOK

**What is it?** Design is what almost every engineer wants to do. It is the place where creativity rules—where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system. Design creates a representation or model of the software, but unlike the requirements model (that focuses on describing required data, function, and behavior), the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.

**Who does it?** Software engineers conduct each of the design tasks.

**Why is it important?** Design allows you to model the system or product that is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted, and end users become involved in large numbers. Design is the place where software quality is established.

**What are the steps?** Design depicts the software in a number of different ways. First, the

**S**oftware design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. Design principles establish an overriding philosophy that guides you in the design work you must perform. Design concepts must be understood before the mechanics of design practice are applied, and design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

Design is pivotal to successful software engineering. In the early 1990s Mitch Kapor, the creator of Lotus 1-2-3, presented a “software design manifesto” in *Dr. Dobbs Journal*. He said:

What is design? It's where you stand with a foot in two worlds—the world of technology and the world of people and human purposes—and you try to bring the two together. . . .

architecture of the system or product must be represented. Then, the interfaces that connect the software to end users, to other systems and devices, and to its own constituent components are modeled. Finally, the software components that are used to construct the system are designed. Each of these views represents a different design action, but all must conform to a set of basic design concepts that guide software design work.

**What is the work product?** A design model that encompasses architectural, interface, component-level, and deployment representations is the primary work product that is produced during software design.

**How do I ensure that I've done it right?** The design model is assessed by the software team in an effort to determine whether it contains errors, inconsistencies, or omissions; whether better alternatives exist; and whether the model can be implemented within the constraints, schedule, and cost that have been established.

<b>modularity . . . . .</b>	<b>225</b>
<b>object-oriented design . . . . .</b>	<b>230</b>
<b>patterns . . . . .</b>	<b>224</b>
<b>quality attributes . . . . .</b>	<b>220</b>
<b>quality guidelines . . . . .</b>	<b>219</b>
<b>refactoring . . . . .</b>	<b>229</b>
<b>separation of concerns . . . . .</b>	<b>225</b>
<b>software design . . . . .</b>	<b>221</b>
<b>stepwise refinement . . . . .</b>	<b>228</b>

The Roman architecture critic Vitruvius advanced the notion that well-designed buildings were those which exhibited firmness, commodity, and delight. The same might be said of good software. *Firmness*: A program should not have any bugs that inhibit its function. *Commodity*: A program should be suitable for the purposes for which it was intended. *Delight*: The experience of using the program should be a pleasurable one. Here we have the beginnings of a theory of design for software.

The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, you must practice diversification and then convergence. Belady [Bel81] states that “diversification is the acquisition of a repertoire of alternatives, the raw material of design: components, component solutions, and knowledge, all contained in catalogs, textbooks, and the mind.” Once this diverse set of information is assembled, you must pick and choose elements from the repertoire that meet the requirements defined by requirements engineering and the analysis model (Chapters 5 through 7). As this occurs, alternatives are considered and rejected and you converge on “one particular configuration of components, and thus the creation of the final product” [Bel81].

Diversification and convergence combine intuition and judgment based on experience in building similar entities, a set of principles and/or heuristics that guide the way in which the model evolves, a set of criteria that enables quality to be judged, and a process of iteration that ultimately leads to a final design representation.

Software design changes continually as new methods, better analysis, and broader understanding evolve.<sup>1</sup> Even today, most software design methodologies lack the depth, flexibility, and quantitative nature that are normally associated with more classical engineering design disciplines. However, methods for software design do exist, criteria for design quality are available, and design notation can be applied. In this chapter, I explore the fundamental concepts and principles that are applicable to all software design, the elements of the design model, and the impact of patterns on the design process. In Chapters 9 through 13 I’ll present a variety of software design methods as they are applied to architectural, interface, and component-level design as well as pattern-based and Web-oriented design approaches.

## 8.1 DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

### Quote:

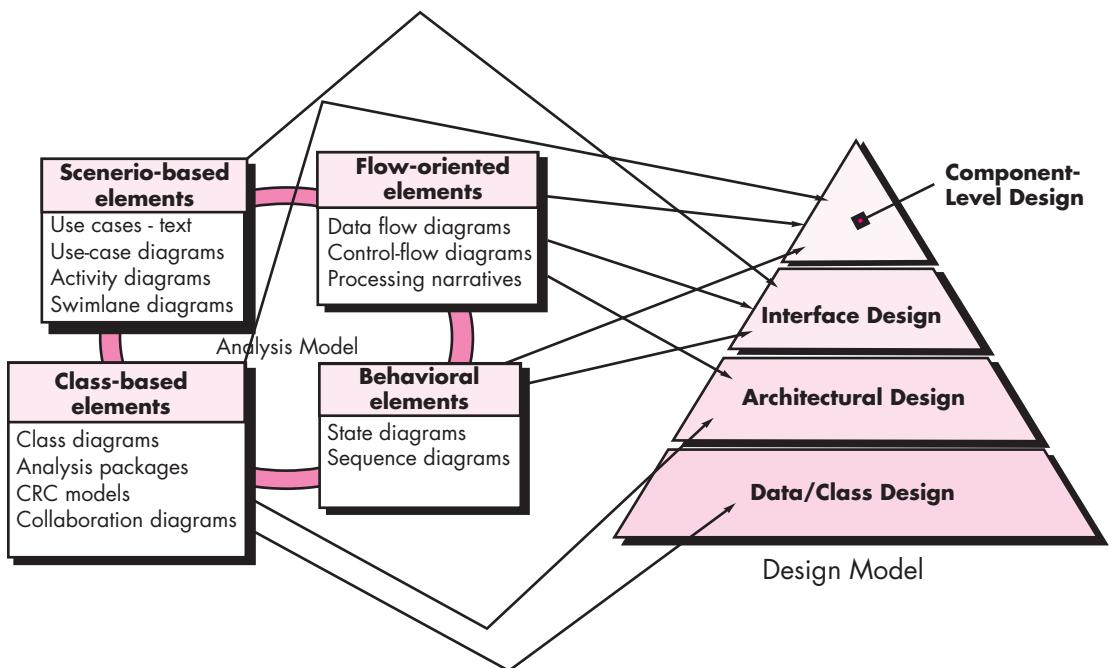
“The most common miracle of software engineering is the transition from analysis to design and design to code.”

Richard Due’

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for **construction** (code generation and testing).

<sup>1</sup> Those readers with further interest in the philosophy of software design might have interest in Philippe Kruchten’s intriguing discussion of “post-modern” design [Kru05a].

**FIGURE 8.1** Translating the requirements model into the design model



Each of the elements of the requirements model (Chapters 6 and 7) provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in Figure 8.1. The requirements model, manifested by scenario-based, class-based, flow-oriented, and behavioral elements, feed the design task. Using design notation and design methods discussed in later chapters, design produces a data/class design, an architectural design, an interface design, and a component design.



*Software design should always begin with a consideration of data—the foundation for all other elements of the design. After the foundation is laid, the architecture must be derived. Only then should you perform other design tasks.*

The data/class design transforms class models (Chapter 6) into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action. Part of class design may occur in conjunction with the design of software architecture. More detailed class design occurs as each software component is designed.

The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented [Sha96]. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

**Quote:**

"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

C. A. R. Hoare

The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

During design you make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained. But why is design so important?

The importance of software design can be stated with a single word—*quality*. Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow. Without design, you risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

## SAFEHOME



### Design versus Coding

**The scene:** Jamie's cubicle, as the team prepares to translate requirements into design.

**The players:** Jamie, Vinod, and Ed—all members of the SafeHome software engineering team.

#### The conversation:

**Jamie:** You know, Doug [the team manager] is obsessed with design. I gotta be honest, what I really love doing is coding. Give me C++ or Java, and I'm happy.

**Ed:** Nah . . . you like to design.

**Jamie:** You're not listening; coding is where it's at.

**Vinod:** I think what Ed means is you don't really like coding; you like to design and express it in code. Code is the language you use to represent the design.

**Jamie:** And what's wrong with that?

**Vinod:** Level of abstraction.

**Jamie:** Huh?

**Ed:** A programming language is good for representing details like data structures and algorithms, but it's not so good for representing architecture or component-to-component collaboration . . . stuff like that.

**Vinod:** And a screwed-up architecture can ruin even the best code.

**Jamie (thinking for a minute):** So, you're saying that I can't represent architecture in code . . . that's not true.

**Vinod:** You can certainly imply architecture in code, but in most programming languages, it's pretty difficult to get a quick, big-picture read on architecture by examining the code.

**Ed:** And that's what we want before we begin coding.

**Jamie:** Okay, maybe design and coding are different, but I still like coding better.

## 8.2 THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

### 8.2.1 Software Quality Guidelines and Attributes



#### quote:

“... writing a clever piece of code that works is one thing; designing something that can support a long-lasting business is quite another.”

C. Ferguson

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews discussed in Chapter 15. McGlaughlin [McG91] suggests three characteristics that serve as a guide for the evaluation of a good design:

- The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Each of these characteristics is actually a goal of the design process. But how is each of these goals achieved?

**Quality Guidelines.** In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design. In Section 8.3, I discuss design concepts that also serve as software quality criteria. For the time being, consider the following guidelines:



#### What are the characteristics of a good design?

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion,<sup>2</sup> thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.

---

2 For smaller systems, design can sometimes be developed linearly.

4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

These design guidelines are not achieved by chance. They are achieved through the application of fundamental design principles, systematic methodology, and thorough review.

### INFO



#### **Assessing Design Quality—The Technical Review**

Design is important because it allows a software team to assess the quality<sup>3</sup> of the software before it is implemented—at a time when errors, omissions, or inconsistencies are easy and inexpensive to correct. But how do we assess quality during design? The software can't be tested, because there is no executable software to test. What to do?

During design, quality is assessed by conducting a series of technical reviews (TRs). TRs are discussed in detail in Chapter 15,<sup>4</sup> but it's worth providing a summary of the technique at this point. A technical review is a meeting conducted by members of the software team. Usually two, three, or four people participate depending on the scope of the design information to be reviewed. Each person plays

a role: the *review leader* plans the meeting, sets an agenda, and runs the meeting; the *recorder* takes notes so that nothing is missed; the *producer* is the person whose work product (e.g., the design of a software component) is being reviewed. Prior to the meeting, each person on the review team is given a copy of the design work product and is asked to read it, looking for errors, omissions, or ambiguity. When the meeting commences, the intent is to note all problems with the work product so that they can be corrected before implementation begins. The TR typically lasts between 90 minutes and 2 hours. At the conclusion of the TR, the review team determines whether further actions are required on the part of the producer before the design work product can be approved as part of the final design model.

### NOTE:

"Quality isn't something you lay on top of subjects and objects like tinsel on a Christmas tree."

**Robert Pirsig**

**Quality Attributes.** Hewlett-Packard [Gra87] developed a set of software quality attributes that has been given the acronym FURPS—functionality, usability, reliability, performance, and supportability. The FURPS quality attributes represent a target for all software design:

- *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.

3 The quality factors discussed in Chapter 23 can assist the review team as it assesses quality.

4 You might consider reviewing Chapter 15 at this time. Technical reviews are a critical part of the design process and are an important mechanism for achieving design quality.



*Software designers tend to focus on the problem to be solved. Just don't forget that the FURPS attributes are always part of the problem. They must be considered.*

- *Usability* is assessed by considering human factors (Chapter 11), overall aesthetics, consistency, and documentation.
- *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- *Performance* is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- *Supportability* combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, *Maintainability*—and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration, Chapter 22), the ease with which a system can be installed, and the ease with which problems can be localized.

Not every software quality attribute is weighted equally as the software design is developed. One application may stress functionality with a special emphasis on security. Another may demand performance with particular emphasis on processing speed. A third might focus on reliability. Regardless of the weighting, it is important to note that these quality attributes must be considered as design commences, *not* after the design is complete and construction has begun.

### 8.2.2 The Evolution of Software Design

#### Q uote:

"A designer knows that he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away."

**Antoine de St-Exupéry**

What characteristics are common to all design methods?

The evolution of software design is a continuing process that has now spanned almost six decades. Early design work concentrated on criteria for the development of modular programs [Den73] and methods for refining software structures in a top-down manner [Wir71]. Procedural aspects of design definition evolved into a philosophy called *structured programming* [Dah72], [Mil72]. Later work proposed methods for the translation of data flow [Ste74] or data structure (e.g., [Jac75], [War74]) into a design definition. Newer design approaches (e.g., [Jac92], [Gam95]) proposed an object-oriented approach to design derivation. More recent emphasis in software design has been on software architecture [Kru06] and the design patterns that can be used to implement software architectures and lower levels of design abstractions (e.g., [Hol06] [Sha05]). Growing emphasis on aspect-oriented methods (e.g., [Cla05], [Jac04]), model-driven development [Sch06], and test-driven development [Ast04] emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

A number of design methods, growing out of the work just noted, are being applied throughout the industry. Like the analysis methods presented in Chapters 6 and 7, each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes design quality. Yet, all of these methods have a number of common characteristics: (1) a mechanism for the translation of the requirements model into a design representation, (2) a notation for

representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

Regardless of the design method that is used, you should apply a set of basic concepts to data, architectural, interface, and component-level design. These concepts are considered in the sections that follow.

## TASK SET



### Generic Task Set for Design

1. Examine the information domain model, and design appropriate data structures for data objects and their attributes.
2. Using the analysis model, select an architectural style that is appropriate for the software.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture:
  - Be certain that each subsystem is functionally cohesive.
  - Design subsystem interfaces.
  - Allocate analysis classes or functions to each subsystem.
4. Create a set of design classes or components:
  - Translate analysis class description into a design class.
  - Check each design class against design criteria; consider inheritance issues.
  - Define methods and messages associated with each design class.
5. Evaluate and select design patterns for a design class or a subsystem.
6. Review design classes and revise as required.
7. Design any interface required with external systems or devices.
8. Design the user interface:
  - Review results of task analysis.
  - Specify action sequence based on user scenarios.
  - Create behavioral model of the interface.
  - Define interface objects, control mechanisms.
  - Review the interface design and revise as required.
9. Conduct component-level design.
  - Specify all algorithms at a relatively low level of abstraction.
  - Refine the interface of each component.
  - Define component-level data structures.
  - Review each component and correct all errors uncovered.
10. Develop a deployment model.

## 8.3 DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the history of software engineering. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps you answer the following questions:

- What criteria can be used to partition software into individual components?
- How is function or data structure detail separated from a conceptual representation of the software?
- What uniform criteria define the technical quality of a software design?

M. A. Jackson [Jac75] once said: “The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right.” Fundamental software design concepts provide the necessary framework for “getting it right.”

In the sections that follow, I present a brief overview of important software design concepts that span both traditional and object-oriented software development.

### 8.3.1 Abstraction



#### note:

"Abstraction is one of the fundamental ways that we as humans cope with complexity."

**Grady Booch**



#### ADVICE

As a designer, work hard to derive both procedural and data abstractions that serve the problem at hand. If they can serve an entire domain of problems, that's even better.

When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

As different levels of abstraction are developed, you work to create both procedural and data abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).<sup>5</sup>

A *data abstraction* is a named collection of data that describes a data object. In the context of the procedural abstraction *open*, we can define a data abstraction called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door**.

### 8.3.2 Architecture



#### WebRef

An in-depth discussion of software architecture can be found at [www.sei.cmu.edu/ata/ata\\_init.html](http://www.sei.cmu.edu/ata/ata_init.html).

*Software architecture* alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system" [Sha95a]. In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions.

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to solve common design problems.

---

<sup>5</sup> It should be noted, however, that one set of operations can be replaced with another, as long as the function implied by the procedural abstraction remains the same. Therefore, the steps required to implement *open* would change dramatically if the door were automatic and attached to a sensor.

**Quote:**

"A software architecture is the development work product that gives the highest return on investment with respect to quality, schedule, and cost."

**Len Bass et al.**

**Advice ..**

*Don't just let architecture happen. If you do, you'll spend the rest of the project trying to force fit the design. Design architecture explicitly.*

**Quote:**

*"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."*

**Christopher Alexander**

Shaw and Garlan [Sha95a] describe a set of properties that should be specified as part of an architectural design:

**Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

**Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

**Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models [Gar95]. *Structural models* represent architecture as an organized collection of program components. *Framework models* increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications. *Dynamic models* address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. *Process models* focus on the design of the business or technical process that the system must accommodate. Finally, *functional models* can be used to represent the functional hierarchy of a system.

A number of different *architectural description languages* (ADLs) have been developed to represent these models [Sha95b]. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

You should note that there is some debate about the role of architecture in design. Some researchers argue that the derivation of software architecture should be separated from design and occurs between requirements engineering actions and more conventional design actions. Others believe that the derivation of architecture is an integral part of the design process. The manner in which software architecture is characterized and its role in design are discussed in Chapter 9.

### 8.3.3 Patterns

Brad Appleton defines a *design pattern* in the following manner: "A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns" [App00]. Stated in another way, a design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can be reused (hence, saving design time), and (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern. Design patterns are discussed in detail in Chapter 12.

### 8.3.4 Separation of Concerns

*Separation of concerns* is a design concept [Dij82] that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A *concern* is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.



*The argument for separation of concerns can be taken too far. If you divide a problem into an inordinate number of very small problems, solving each will be easy, but putting the solution together—integration—may be very difficult.*

For two problems,  $p_1$  and  $p_2$ , if the perceived complexity of  $p_1$  is greater than the perceived complexity of  $p_2$ , it follows that the effort required to solve  $p_1$  is greater than the effort required to solve  $p_2$ . As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.

It also follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. This leads to a divide-and-conquer strategy—it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to software modularity.

Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement. Each will be discussed in the subsections that follow.

### 8.3.5 Modularity

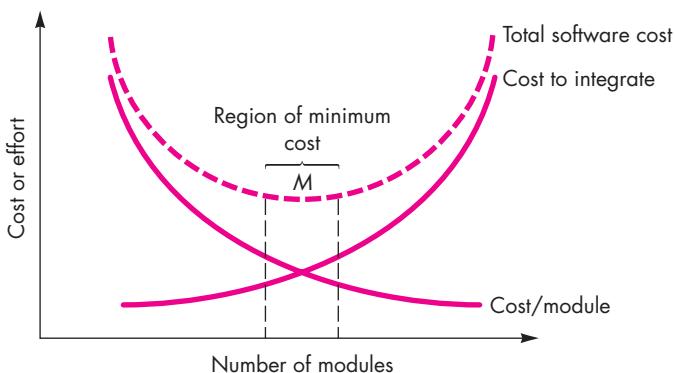
Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called *modules*, that are integrated to satisfy problem requirements.

It has been stated that “modularity is the single attribute of software that allows a program to be intellectually manageable” [Mye78]. Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

Recalling my discussion of separation of concerns, it is possible to conclude that if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure 8.2, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the

**FIGURE 8.2**

**Modularity and software cost**



same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number,  $M$ , of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict  $M$  with assurance.

**What is the right number of modules for a given system?**

The curves shown in Figure 8.2 do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of  $M$ . Undermodularity or overmodularity should be avoided. But how do you know the vicinity of  $M$ ? How modular should you make software? The answers to these questions require an understanding of other design concepts considered later in this chapter.

You modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

### 8.3.6 Information Hiding

The concept of modularity leads you to a fundamental question: "How do I decompose a software solution to obtain the best set of modules?" The principle of information hiding [Par72] suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

**KEY POINT**

The intent of information hiding is to hide the details of data structures and procedural processing behind a module interface. Knowledge of the details need not be known by users of the module.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module [Ros75].

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

### 8.3.7 Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. In landmark papers on software design, Wirth [Wir71] and Parnas [Par72] allude to refinement techniques that enhance module independence. Later work by Stevens, Myers, and Constantine [Ste74] solidified the concept.

Functional independence is achieved by developing modules with “single-minded” function and an “aversion” to excessive interaction with other modules. Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure. It is fair to ask why independence is important.

 **Why should you strive to create independent modules?**

 **KEY POINT**

Cohesion is a qualitative indication of the degree to which a module focuses on just one thing.

 **KEY POINT**

Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world.

Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: cohesion and coupling. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules.

Cohesion is a natural extension of the information-hiding concept described in Section 8.3.6. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions. However, “schizophrenic” components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a “ripple effect” [Ste74], caused when errors occur at one location and propagate throughout a system.



*There is a tendency to move immediately to full detail, skipping refinement steps. This leads to errors and omissions and makes the design much more difficult to review. Perform stepwise refinement.*

**quote:**  
"It's hard to read through a book on the principles of magic without glancing at the cover periodically to make sure it isn't a book on software design."

Bruce Tognazzini



A crosscutting concern is some characteristic of the system that applies across many different requirements.

### 8.3.8 Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth [Wir71]. A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Refinement is actually a process of *elaboration*. You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses. Both concepts allow you to create a complete design model as the design evolves.

### 8.3.9 Aspects

As requirements analysis occurs, a set of "concerns" is uncovered. These concerns "include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts" [AOS07]. Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently. In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized.

As design begins, requirements are refined into a modular design representation. Consider two requirements, *A* and *B*. Requirement *A* *crosscuts* requirement *B* "if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account" [Ros04].

For example, consider two requirements for the **SafeHomeAssured.com** WebApp. Requirement *A* is described via the **ACS-DCV** use case discussed in Chapter 6. A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using SafeHomeAssured.com*. This requirement is applicable for all functions that are available to registered *SafeHome* users. As design refinement occurs, *A\** is a design representation for requirement *A* and *B\** is a design representation for requirement *B*. Therefore, *A\** and *B\** are representations of concerns, and *B\** *crosscuts* *A\**.

An aspect is a representation of a crosscutting concern. Therefore, the design representation, *B\**, of the requirement *a registered user must be validated prior to using SafeHomeAssured.com*, is an aspect of the *SafeHome* WebApp. It is important to

identify aspects so that the design can properly accommodate them as refinement and modularization occur. In an ideal context, an aspect is implemented as a separate module (component) rather than as software fragments that are “scattered” or “tangled” throughout many components [Ban06]. To accomplish this, the design architecture should support a mechanism for defining an aspect—a module that enables the concern to be implemented across all other concerns that it crosscuts.

### 8.3.10 Refactoring

#### WebRef

Excellent resources for refactoring can be found at [www.refactoring.com](http://www.refactoring.com).

#### WebRef

A variety of refactoring patterns can be found at <http://c2.com/cgi/wiki?RefactoringPatterns>.

An important design activity suggested for many agile methods (Chapter 3), *refactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler [Fow00] defines refactoring in the following manner: “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. For example, a first design iteration might yield a component that exhibits low cohesion (i.e., it performs three functions that have only limited relationship to one another). After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion.

## SAFEHOME



### Design Concepts

**The scene:** Vinod’s cubicle, as design modeling begins.

**The players:** Vinod, Jamie, and Ed—members of the *SafeHome* software engineering team. Also, Shakira, a new member of the team.

#### The conversation:

[All four team members have just returned from a morning seminar entitled “Applying Basic Design Concepts,” offered by a local computer science professor.]

**Vinod:** Did you get anything out of the seminar?

**Ed:** Knew most of the stuff, but it’s not a bad idea to hear it again, I suppose.

**Jamie:** When I was an undergrad CS major, I never really understood why information hiding was as important as they say it is.

**Vinod:** Because . . . bottom line . . . it’s a technique for reducing error propagation in a program. Actually, functional independence also accomplishes the same thing.

**Shakira:** I wasn’t a CS grad, so a lot of the stuff the instructor mentioned is new to me. I can generate good code and fast. I don’t see why this stuff is so important.

**Jamie:** I’ve seen your work, Shak, and you know what, you do a lot of this stuff naturally . . . that’s why your designs and code work.

**Shakira (smiling):** Well, I always do try to partition the code, keep it focused on one thing, keep interfaces simple and constrained, reuse code whenever I can . . . that sort of thing.

**Ed:** Modularity, functional independence, hiding, patterns . . . see.

**Jamie:** I still remember the very first programming course I took . . . they taught us to refine the code iteratively.

**Vinod:** Same thing can be applied to design, you know.

**Vinod:** The only concepts I hadn’t heard of before were “aspects” and “refactoring.”

**Shakira:** That's used in Extreme Programming, I think she said.

**Ed:** Yep. It's not a whole lot different than refinement, only you do it after the design or code is completed. Kind of an optimization pass through the software, if you ask me.

**Jamie:** Let's get back to *SafeHome* design. I think we should put these concepts on our review checklist as we develop the design model for *SafeHome*.

**Vinod:** I agree. But as important, let's all commit to think about them as we develop the design.

The result will be software that is easier to integrate, easier to test, and easier to maintain.

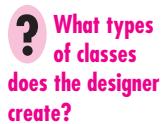
### 8.3.11 Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering. Appendix 2 has been provided for those readers who may be unfamiliar with OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others.

### 8.3.12 Design Classes

The requirements model defines a set of analysis classes (Chapter 6). Each describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high.

As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution. Five different types of design classes, each representing a different layer of the design architecture, can be developed [Amb01]:



- *User interface classes* define all abstractions that are necessary for human-computer interaction (HCI). In many cases, HCI occurs within the context of a *metaphor* (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.
- *Business domain classes* are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- *Process classes* implement lower-level business abstractions required to fully manage the business domain classes.
- *Persistent classes* represent data stores (e.g., a database) that will persist beyond the execution of the software.
- *System classes* implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the architecture forms, the level of abstraction is reduced as each analysis class is transformed into a design representation. That is, analysis classes represent data objects (and associated services that are applied to them) using the jargon of the business domain. Design classes present significantly more technical detail as a guide for implementation.

Arlow and Neustadt [Arl02] suggest that each design class be reviewed to ensure that it is “well-formed.” They define four characteristics of a well-formed design class:



**Complete and sufficient.** A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class.

For example, the class **Scene** defined for video-editing software is complete only if it contains all attributes and methods that can reasonably be associated with the creation of a video scene. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

**Primitiveness.** Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing. For example, the class **VideoClip** for video-editing software might have attributes **start-point** and **end-point** to indicate the start and end points of the clip (note that the raw video loaded into the system may be longer than the clip that is used). The methods, *setStartPoint()* and *setEndPoint()*, provide the only means for establishing start and end points for the clip.

**High cohesion.** A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities. For example, the class **VideoClip** might contain a set of methods for editing the video clip. As long as each method focuses solely on attributes associated with the video clip, cohesion is maintained.

**Low coupling.** Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the *Law of Demeter* [Lie03], suggests that a method should only send messages to methods in neighboring classes.<sup>6</sup>

---

<sup>6</sup> A less formal way of stating the Law of Demeter is “Each unit should only talk to its friends; Don’t talk to strangers.”

## SAFEHOME



### Refining an Analysis Class into a Design Class

**The scene:** Ed's cubicle, as design modeling begins.

**The players:** Vinod and Ed—members of the SafeHome software engineering team.

#### The conversation:

[Ed is working on the **FloorPlan** class (see sidebar discussion in Section 6.5.3 and Figure 6.10) and has refined it for the design model.]

**Ed:** So you remember the **FloorPlan** class, right? It's used as part of the surveillance and home management functions.

**Vinod (nodding):** Yeah, I seem to recall that we used it as part of our CRC discussions for home management.

**Ed:** We did. Anyway, I'm refining it for design. Want to show how we'll actually implement the **FloorPlan** class. My idea is to implement it as a set of linked lists [a specific data structure] So . . . I had to refine the analysis class **FloorPlan** (Figure 6.10) and actually, sort of simplify it.

**Vinod:** The analysis class showed only things in the problem domain, well, actually on the computer screen, that were visible to the end user, right?

**Ed:** Yep, but for the **FloorPlan** design class, I've got to add some things that are implementation specific. I needed to show that **FloorPlan** is an aggregation of segments—hence the **Segment** class—and that the **Segment** class is composed of lists for wall segments, windows, doors, and so on. The class **Camera** collaborates with **FloorPlan**, and obviously, there can be many cameras in the floor plan.

**Vinod:** Phew, let's see a picture of this new **FloorPlan** design class.

[Ed shows Vinod the drawing shown in Figure 8.3.]

**Vinod:** Okay, I see what you're trying to do. This allows you to modify the floor plan easily because new items can be added to or deleted from the list—the aggregation—without any problems.

**Ed (nodding):** Yeah, I think it'll work.

**Vinod:** So do I.

FIGURE 8.3

Design class for **FloorPlan** and composite aggregation for the class (see sidebar discussion)



## 8.4 THE DESIGN MODEL

The design model can be viewed in two different dimensions as illustrated in Figure 8.4. The *process dimension* indicates the evolution of the design model as design tasks are executed as part of the software process. The *abstraction dimension* represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. Referring to Figure 8.4, the dashed line indicates the boundary between the analysis and design models. In some cases, a clear distinction between the analysis and design models is possible. In other cases, the analysis model slowly blends into the design and a clear distinction is less obvious.

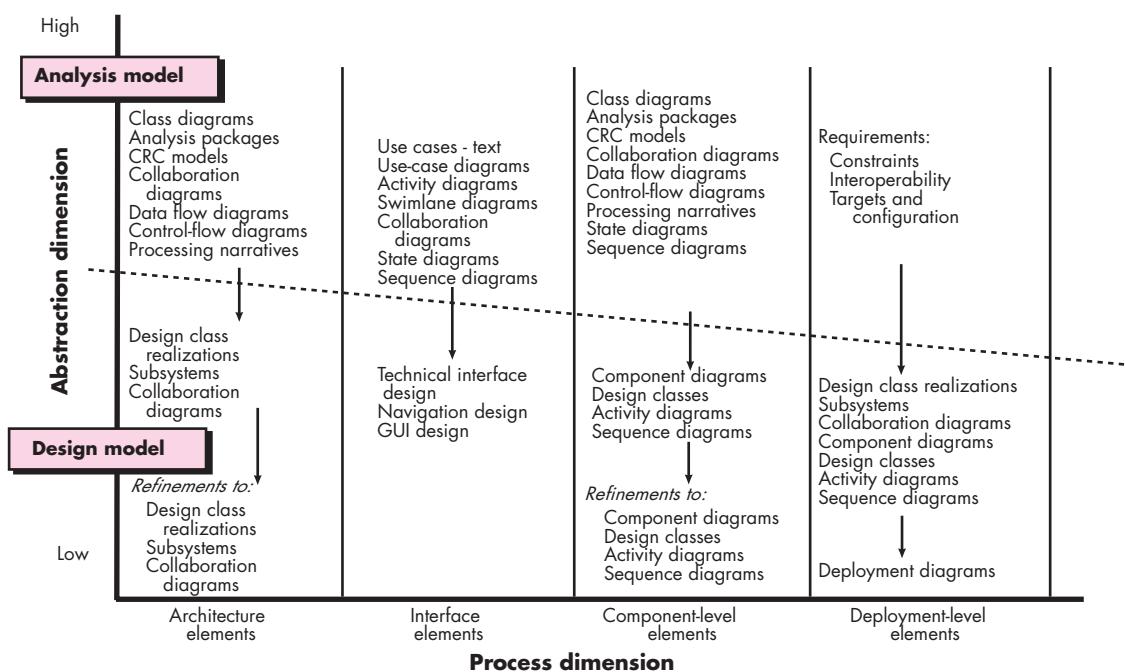


The design model has four major elements: data, architecture, components, and interface.

The elements of the design model use many of the same UML diagrams<sup>7</sup> that were used in the analysis model. The difference is that these diagrams are refined and elaborated as part of design; more implementation-specific detail is provided, and architectural structure and style, components that reside within the architecture, and interfaces between the components and with the outside world are all emphasized.

**FIGURE 8.4**

Dimensions of the design model



7 Appendix 1 provides a tutorial on basic UML concepts and notation.

**note:**

"Questions about whether design is necessary or affordable are quite beside the point: design is inevitable. The alternative to good design is bad design, not no design at all."

Douglas Martin

**KEY POINT**

At the architectural (application) level, data design focuses on files or databases; at the component level, data design considers the data structures that are required to implement local data objects.

**note:**

"You can use an eraser on the drafting table or a sledge hammer on the construction site."

Frank Lloyd Wright

You should note, however, that model elements indicated along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model is usually delayed until the design has been fully developed.

You can apply design patterns (Chapter 12) at any point during design. These patterns enable you to apply design knowledge to domain-specific problems that have been encountered and solved by others.

### 8.4.1 Data Design Elements

Like other software engineering activities, data design (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role. Data design is discussed in more detail in Chapter 9.

### 8.4.2 Architectural Design Elements

The *architectural design* for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.

The architectural model [Sha96] is derived from three sources: (1) information about the application domain for the software to be built; (2) specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and (3) the availability of architectural styles (Chapter 9) and patterns (Chapter 12).

The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model. Each subsystem may have its own architecture (e.g., a graphical user interface might

**quote:**

"The public is more familiar with bad design than good design. It is, in effect, conditioned to prefer bad design, because that is what it lives with. The new becomes threatening, the old reassuring."

Paul Rand

**KEY POINT**

There are three parts to the interface design element: the user interface, interfaces to system external to the application, and interfaces to components within the application.

**quote:**

"Every now and then go away, have a little relaxation, for when you come back to your work your judgment will be surer. Go some distance away because then the work appears smaller and more of it can be taken in at a glance and a lack of harmony and proportion is more readily seen."

Leonardo DaVinci

be structured according to a preexisting architectural style for user interfaces). Techniques for deriving specific elements of the architectural model are presented in Chapter 9.

### 8.4.3 Interface Design Elements

The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. These drawings depict the size and shape of doors and windows, the manner in which they operate, the way in which utility connections (e.g., water, electrical, gas, telephone) come into the house and are distributed among the rooms depicted in the floor plan. They tell us where the doorbell is located, whether an intercom is to be used to announce a visitor's presence, and how a security system is to be installed. In essence, the detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan. The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.

There are three important elements of interface design: (1) the user interface (UI); (2) external interfaces to other systems, devices, networks, or other producers or consumers of information; and (3) internal interfaces between various design components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

UI design (increasingly called *usability design*) is a major software engineering action and is considered in detail in Chapter 11. Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture.

The design of external interfaces requires definitive information about the entity to which information is sent or received. In every case, this information should be collected during requirements engineering (Chapter 5) and verified once the interface design commences.<sup>8</sup> The design of external interfaces should incorporate error checking and (when necessary) appropriate security features.

The design of internal interfaces is closely aligned with component-level design (Chapter 10). Design realizations of analysis classes represent all operations and the messaging schemes required to enable communication and collaboration between operations in various classes. Each message must be designed to accommodate the requisite information transfer and the specific functional requirements of the

<sup>8</sup> Interface characteristics can change with time. Therefore, a designer should ensure that the specification for the interface is accurate and complete.

operation that has been requested. If the classic input-process-output approach to design is chosen, the interface of each software component is designed based on data flow representations and the functionality described in a processing narrative.

### WebRef

Extremely valuable information on UI design can be found at [www.useit.com](http://www.useit.com).

### note:

"A common mistake that people make when trying to design something completely foolproof was to underestimate the ingenuity of complete fools."

Douglas Adams

In some cases, an interface is modeled in much the same way as a class. In UML, an interface is defined in the following manner [OMG03a]: "An interface is a specifier for the externally-visible [public] operations of a class, component, or other classifier (including subsystems) without specification of internal structure." Stated more simply, an interface is a set of operations that describes some part of the behavior of a class and provides access to these operations.

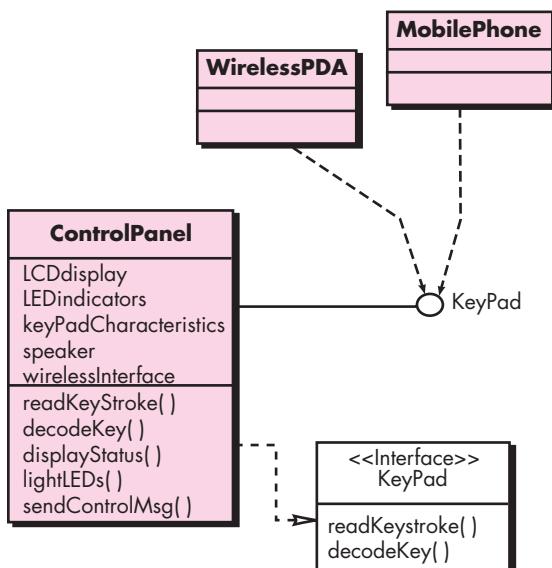
For example, the *SafeHome* security function makes use of a control panel that allows a homeowner to control certain aspects of the security function. In an advanced version of the system, control panel functions may be implemented via a wireless PDA or mobile phone.

The **ControlPanel** class (Figure 8.5) provides the behavior associated with a keypad, and therefore, it must implement the operations *readKeyStroke()* and *decodeKey()*. If these operations are to be provided to other classes (in this case, **WirelessPDA** and **MobilePhone**), it is useful to define an interface as shown in the figure. The interface, named **KeyPad**, is shown as an <<interface>> stereotype or as a small, labeled circle connected to the class with a line. The interface is defined with no attributes and the set of operations that are necessary to achieve the behavior of a keypad.

The dashed line with an open triangle at its end (Figure 8.5) indicates that the **ControlPanel** class provides **KeyPad** operations as part of its behavior. In UML, this

**FIGURE 8.5**

Interface representation for Control Panel



is characterized as a *realization*. That is, part of the behavior of **ControlPanel** will be implemented by realizing **Keypad** operations. These operations will be provided to other classes that access the interface.

#### 8.4.4 Component-Level Design Elements

The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets. They also describe the flooring to be used, the moldings to be applied, and every other detail associated with a room. The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

**Quote:**

"The details are not the details.  
They make the design."

Charles Eames

Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in Figure 8.6. In this figure, a component named **SensorManagement** (part of the *SafeHome* security function) is represented. A dashed arrow connects the component to a class named **Sensor** that is assigned to it. The **SensorManagement** component performs all functions associated with *SafeHome* sensors including monitoring and configuring them. Further discussion of component diagrams is presented in Chapter 10.

The design details of a component can be modeled at many different levels of abstraction. A UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode (a programming language-like representation described in Chapter 10) or some other diagrammatic form (e.g., flowchart or box diagram). Algorithmic structure follows the rules established for structured programming (i.e., a set of constrained procedural constructs). Data structures, selected based on the nature of the data objects to be processed, are usually modeled using pseudocode or the programming language to be used for implementation.

#### 8.4.5 Deployment-Level Design Elements

Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support

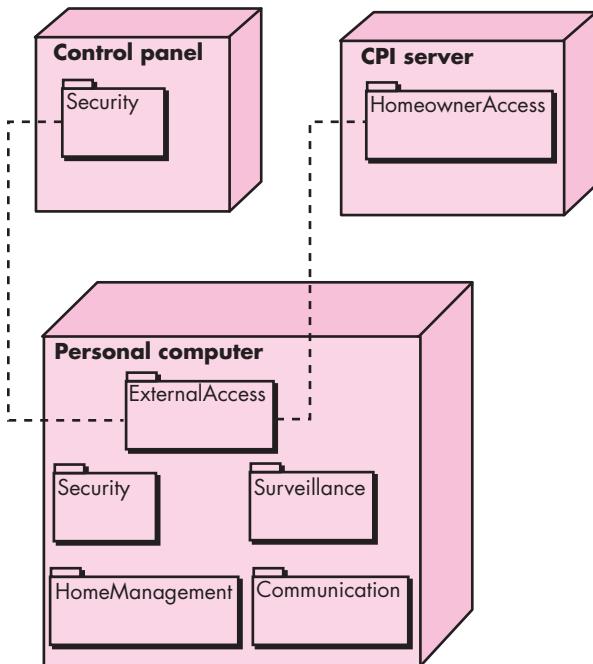
FIGURE 8.6

A UML  
component  
diagram



**FIGURE 8.7**

A UML deployment diagram



the software. For example, the elements of the *SafeHome* product are configured to operate within three primary computing environments—a home-based PC, the *SafeHome* control panel, and a server housed at CPI Corp. (providing Internet-based access to the system).

### KEY POINT

Deployment diagrams begin in descriptor form, where the deployment environment is described in general terms. Later, instance form is used and elements of the configuration are explicitly described.

During design, a UML deployment diagram is developed and then refined as shown in Figure 8.7. In the figure, three computing environments are shown (in actuality, there would be more including sensors, cameras, and others). The subsystems (functionality) housed within each computing element are indicated. For example, the personal computer houses subsystems that implement security, surveillance, home management, and communications features. In addition, an external access subsystem has been designed to manage all attempts to access the *SafeHome* system from an external source. Each subsystem would be elaborated to indicate the components that it implements.

The diagram shown in Figure 8.7 is in *descriptor form*. This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details. For example, the “personal computer” is not further identified. It could be a Mac or a Windows-based PC, a Sun workstation, or a Linux-box. These details are provided when the deployment diagram is revisited in *instance form* during the latter stages of design or as construction begins. Each instance of the deployment (a specific, named hardware configuration) is identified.

## 8.5 SUMMARY

Software design commences as the first iteration of requirements engineering comes to a conclusion. The intent of software design is to apply a set of principles, concepts, and practices that lead to the development of a high-quality system or product. The goal of design is to create a model of software that will implement all customer requirements correctly and bring delight to those who use it. Software designers must sift through many design alternatives and converge on a solution that best suits the needs of project stakeholders.

The design process moves from a “big picture” view of software to a more narrow view that defines the detail required to implement a system. The process begins by focusing on architecture. Subsystems are defined; communication mechanisms among subsystems are established; components are identified, and a detailed description of each component is developed. In addition, external, internal, and user interfaces are designed.

Design concepts have evolved over the first 60 years of software engineering work. They describe attributes of computer software that should be present regardless of the software engineering process that is chosen, the design methods that are applied, or the programming languages that are used. In essence, design concepts emphasize the need for abstraction as a mechanism for creating reusable software components; the importance of architecture as a way to better understand the overall structure of a system; the benefits of pattern-based engineering as a technique for designing software with proven capabilities; the value of separation of concerns and effective modularity as a way to make software more understandable, more testable, and more maintainable; the consequences of information hiding as a mechanism for reducing the propagation of side effects when errors do occur; the impact of functional independence as a criterion for building effective modules; the use of refinement as a design mechanism; a consideration of aspects that crosscut system requirements; the application of refactoring for optimizing the design that is derived; and the importance of object-oriented classes and the characteristics that are related to them.

The design model encompasses four different elements. As each of these elements is developed, a more complete view of the design evolves. The architectural element uses information derived from the application domain, the requirements model, and available catalogs for patterns and styles to derive a complete structural representation of the software, its subsystems, and components. Interface design elements model external and internal interfaces and the user interface. Component-level elements define each of the modules (components) that populate the architecture. Finally, deployment-level design elements allocate the architecture, its components, and the interfaces to the physical configuration that will house the software.

## PROBLEMS AND POINTS TO PONDER

- 8.1.** Do you design software when you “write” a program? What makes software design different from coding?
- 8.2.** If a software design is not a program (and it isn’t), then what is it?
- 8.3.** How do we assess the quality of a software design?
- 8.4.** Examine the task set presented for design. Where is quality assessed within the task set? How is this accomplished? How are the quality attributes discussed in Section 8.2.1 achieved?
- 8.5.** Provide examples of three data abstractions and the procedural abstractions that can be used to manipulate them.
- 8.6.** Describe software architecture in your own words.
- 8.7.** Suggest a design pattern that you encounter in a category of everyday things (e.g., consumer electronics, automobiles, appliances). Briefly describe the pattern.
- 8.8.** Describe separation of concerns in your own words. Is there a case when a divide-and-conquer strategy may not be appropriate? How might such a case affect the argument for modularity?
- 8.9.** When should a modular design be implemented as monolithic software? How can this be accomplished? Is performance the only justification for implementation of monolithic software?
- 8.10.** Discuss the relationship between the concept of information hiding as an attribute of effective modularity and the concept of module independence.
- 8.11.** How are the concepts of coupling and software portability related? Provide examples to support your discussion.
- 8.12.** Apply a “stepwise refinement approach” to develop three different levels of procedural abstractions for one or more of the following programs: (a) Develop a check writer that, given a numeric dollar amount, will print the amount in words normally required on a check. (b) Iteratively solve for the roots of a transcendental equation. (c) Develop a simple task scheduling algorithm for an operating system.
- 8.13.** Consider the software required to implement a full navigation capability (using GPS) in a mobile, handheld communication device. Describe two or three crosscutting concerns that would be present. Discuss how you would represent one of these concerns as an aspect.
- 8.14.** Does “refactoring” mean that you modify the entire design iteratively? If not, what does it mean?
- 8.15.** Briefly describe each of the four elements of the design model.

## FURTHER READINGS AND INFORMATION SOURCES

Donald Norman has written two books (*The Design of Everyday Things*, Doubleday, 1990, and *The Psychology of Everyday Things*, HarperCollins, 1988) that have become classics in the design literature and “must” reading for anyone who designs anything that humans use. Adams (*Conceptual Blockbusting*, 3d ed., Addison-Wesley, 1986) has written a book that is essential reading for designers who want to broaden their way of thinking. Finally, a classic text by Polya (*How to Solve It*, 2d ed., Princeton University Press, 1988) provides a generic problem-solving process that can help software designers when they are faced with complex problems.

Following in the same tradition, Winograd et al. (*Bringing Design to Software*, Addison-Wesley, 1996) discusses software designs that work, those that don’t, and why. A fascinating book edited by Wixon and Ramsey (*Field Methods Casebook for Software Design*, Wiley, 1996)

suggests field research methods (much like those used by anthropologists) to understand how end users do the work they do and then design software that meets their needs. Beyer and Holtzblatt (*Contextual Design: A Customer-Centered Approach to Systems Designs*, Academic Press, 1997) offer another view of software design that integrates the customer/user into every aspect of the software design process. Bain (*Emergent Design*, Addison-Wesley, 2008) couples patterns, refactoring, and test-driven development into an effective design approach.

Comprehensive treatment of design in the context of software engineering is presented by Fox (*Introduction to Software Engineering Design*, Addison-Wesley, 2006) and Zhu (*Software Design Methodology*, Butterworth-Heinemann, 2005). McConnell (*Code Complete*, 2d ed., Microsoft Press, 2004) presents an excellent discussion of the practical aspects of designing high-quality computer software. Robertson (*Simple Program Design*, 3d ed., Boyd and Fraser Publishing, 1999) presents an introductory discussion of software design that is useful for those beginning their study of the subject. Budgen (*Software Design*, 2d ed., Addison-Wesley, 2004) introduces a variety of popular design methods, comparing and contrasting each. Fowler and his colleagues (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) discusses techniques for the incremental optimization of software designs. Rosenberg and Stevens (*Use Case Driven Object Modeling with UML*, Apress, 2007) discuss the development of object-oriented designs using use cases as a foundation.

An excellent historical survey of software design is contained in an anthology edited by Freeman and Wasserman (*Software Design Techniques*, 4th ed., IEEE, 1983). This tutorial reprints many of the classic papers that have formed the basis for current trends in software design. Measures of design quality, presented from both the technical and management perspectives, are considered by Card and Glass (*Measuring Software Design Quality*, Prentice-Hall, 1990).

A wide variety of information sources on software design are available on the Internet. An up-to-date list of World Wide Web references that are relevant to software design and design engineering can be found at the SEPA website: [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm).

**KEY  
CONCEPTS**

archetypes	.... 257
architectural description	
language	.... 264
architecture	.... 243
alternatives	.... 261
components	.... 258
complexity	.... 263
data centered	.... 250
data flow	.... 251
design	.... 255
genres	.... 247
layered	.... 253
object oriented	.... 252
patterns	.... 253
refinement	.... 258
styles	.... 249
template	.... 247
ATAM	..... 262
factoring	.... 268
instantiation	.... 260
mapping	.... 265

**QUICK  
LOOK**

**What is it?** Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

**Who does it?** Although a software engineer can design both data and architecture, the job is often allocated to specialists when large, complex systems are to be built. A database or data

**D**esign has been described as a multistep process in which representations of data and program structure, interface characteristics, and procedural detail are synthesized from information requirements. This description is extended by Freeman [Fre80]:

[D]esign is an activity concerned with making major decisions, often of a structural nature. It shares with programming a concern for abstracting information representation and processing sequences, but the level of detail is quite different at the extremes. Design builds coherent, well-planned representations of programs that concentrate on the interrelationships of parts at the higher level and the logical operations involved at the lower levels.

As I noted in Chapter 8, design is information driven. Software design methods are derived from consideration of each of the three domains of the analysis model. The data, functional, and behavioral domains serve as a guide for the creation of the software design.

Methods required to create “coherent, well-planned representations” of the data and architectural layers of the design model are presented in this chapter. The objective is to provide a systematic approach for the derivation of the architectural design—the preliminary blueprint from which software is constructed.

warehouse designer creates the data architecture for a system. The “system architect” selects an appropriate architectural style from the requirements derived during software requirements analysis.

**Why is it important?** You wouldn’t attempt to build a house without a blueprint, would you? You also wouldn’t begin drawing blueprints by sketching the plumbing layout for the house. You’d need to look at the big picture—the house itself—before you worry about details. That’s what architectural design does—it provides you with the big picture and ensures that you’ve got it right.

**What are the steps?** Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes. Once an alternative has been selected, the architecture is elaborated using an architectural design method.

**What is the work product?** An architecture model encompassing data architecture and program structure is created during architectural design. In addition, component properties and relationships (interactions) are described.

**How do I ensure that I've done it right?** At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

## 9.1 SOFTWARE ARCHITECTURE

In their landmark book on the subject, Shaw and Garlan [Sha96] discuss software architecture in the following manner:

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 legacy systems of the 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.

Today, effective software architecture and its explicit representation and design have become dominant themes in software engineering.

### 9.1.1 What Is Architecture?

#### Q uote:

"The architecture of a system is a comprehensive framework that describes its form and structure—its components and how they fit together."

**Jerrold Grochow**

When you consider the architecture of a building, many different attributes come to mind. At the most simplistic level, you think about the overall shape of the physical structure. But in reality, architecture is much more. It is the manner in which the various components of the building are integrated to form a cohesive whole. It is the way in which the building fits into its environment and meshes with other buildings in its vicinity. It is the degree to which the building meets its stated purpose and satisfies the needs of its owner. It is the aesthetic feel of the structure—the visual impact of the building—and the way textures, colors, and materials are combined to create the external facade and the internal "living environment." It is small details—the design of lighting fixtures, the type of flooring, the placement of wall hangings, the list is almost endless. And finally, it is art.

But architecture is also something else. It is "thousands of decisions, both big and small" [Tyr05]. Some of these decisions are made early in design and can have a profound impact on all other design actions. Others are delayed until later, thereby

**KEY POINT**

Software architecture must model the structure of a system and the manner in which data and procedural components collaborate with one another.

**Quote:**

"Marry your architecture in haste, repent at your leisure."

Barry Boehm

eliminating overly restrictive constraints that would lead to a poor implementation of the architectural style.

But what about software architecture? Bass, Clements, and Kazman [Bas03] define this elusive term in the following way:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The architecture is not the operational software. Rather, it is a representation that enables you to (1) analyze the effectiveness of the design in meeting its stated requirements, (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and (3) reduce the risks associated with the construction of the software.

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers. The properties of components are those characteristics that are necessary for an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are not specified. The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

Some members of the software engineering community (e.g., [Kaz03]) make a distinction between the actions associated with the derivation of a software architecture (what I call "architectural design") and the actions that are applied to derive the software design. As one reviewer of this edition noted:

There is a distinct difference between the terms architecture and design. A *design* is an instance of an *architecture* similar to an object being an instance of a class. For example, consider the client-server architecture. I can design a network-centric software system in many different ways from this architecture using either the Java platform (Java EE) or Microsoft platform (.NET framework). So, there is one architecture, but many designs can be created based on that architecture. Therefore, you cannot mix "architecture" and "design" with each other.

Although I agree that a software design is an instance of a specific software architecture, the elements and structures that are defined as part of an architecture are the root of every design that evolves from them. Design begins with a consideration of architecture.

In this book the design of software architecture considers two levels of the design pyramid (Figure 8.1)—data design and architectural design. In the context of the preceding discussion, data design enables you to represent the data component of the architecture in conventional systems and class definitions (encompassing attributes

**WebRef**

Useful pointers to many software architecture sites can be obtained at  
[www2.umassd.edu/SECenter/SAResources.html](http://www2.umassd.edu/SECenter/SAResources.html).

and operations) in object-oriented systems. Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

### 9.1.2 Why Is Architecture Important?

In a book dedicated to software architecture, Bass and his colleagues [Bas03] identify three key reasons that software architecture is important:

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together” [Bas03].

The architectural design model and the architectural patterns contained within it are transferable. That is, architecture genres, styles, and patterns (Sections 9.2 through 9.4) can be applied to the design of other systems and represent a set of abstractions that enable software engineers to describe architecture in predictable ways.

### 9.1.3 Architectural Descriptions

Each of us has a mental image of what the word *architecture* means. In reality, however, it means different things to different people. The implication is that different stakeholders will see an architecture from different viewpoints that are driven by different sets of concerns. This implies that an architectural description is actually a set of work products that reflect different views of the system.

For example, the architect of a major office building must work with a variety of different stakeholders. The primary concern of the owner of the building (one stakeholder) is to ensure that it is aesthetically pleasing and that it provides sufficient office space and infrastructure to ensure its profitability. Therefore, the architect must develop a description using views of the building that address the owner's concerns. The viewpoints used are a three-dimensional drawings of the building (to illustrate the aesthetic view) and a set of two-dimensional floor plans to address this stakeholder's concern for office space and infrastructure.

But the office building has many other stakeholders, including the structural steel fabricator who will provide steel for the building skeleton. The structural steel fabricator needs detailed architectural information about the structural steel that will support the building, including types of I-beams, their dimensions, connectivity, materials, and many other details. These concerns are addressed by different work products that represent different views of the architecture. Specialized drawings



#### note:

“Architecture is far too important to leave in the hands of a single person, no matter how bright they are.”

Scott Ambler



#### KEY POINT

The architectural model provides a Gestalt view of the system, allowing the software engineer to examine it as a whole.



#### ADVICE

Your effort should focus on architectural representations that will guide all other aspects of design. Spend the time to carefully review the architecture. A mistake here will have a long-term negative impact.

(another viewpoint) of the structural steel skeleton of the building focus on only one of many of the fabricator's concerns.

An architectural description of a software-based system must exhibit characteristics that are analogous to those noted for the office building. Tyree and Akerman [Tyr05] note this when they write: "Developers want clear, decisive guidance on how to proceed with design. Customers want a clear understanding on the environmental changes that must occur and assurances that the architecture will meet their business needs. Other architects want a clear, salient understanding of the architecture's key aspects." Each of these "wants" is reflected in a different view represented using a different viewpoint.

The IEEE Computer Society has proposed IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*, [IEE00], with the following objectives: (1) to establish a conceptual framework and vocabulary for use during the design of software architecture, (2) to provide detailed guidelines for representing an architectural description, and (3) to encourage sound architectural design practices.

The IEEE standard defines an *architectural description* (AD) as "a collection of products to document an architecture." The description itself is represented using multiple views, where each *view* is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns." A *view* is created according to rules and conventions defined in a *viewpoint*—"a specification of the conventions for constructing and using a view" [IEE00]. A number of different work products that are used to develop different views of the software architecture are discussed later in this chapter.

#### 9.1.4 Architectural Decisions

Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view (and the architectural description as a whole) the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern. Therefore, architectural decisions themselves can be considered to be one view of the architecture. The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns.

As a system architect, you can use the template suggested in the sidebar to document each major decision. By doing this, you provide a rationale for your work and establish an historical record that can be useful when design modifications must be made.

## 9.2 ARCHITECTURAL GENRES

Although the underlying principles of architectural design apply to all types of architecture, the architectural *genre* will often dictate the specific architectural approach to the structure that must be built. In the context of architectural design, *genre* implies a

**INFO****Architecture Decision Description Template**

Each major architectural decision can be documented for later review by stakeholders who want to understand the architecture description that has been proposed. The template presented in this sidebar is an adapted and abbreviated version of a template proposed by Tyree and Ackerman [Tyr05].

**Design issue:**

Describe the architectural design issues that are to be addressed.

**Resolution:**

State the approach you've chosen to address the design issue.

**Category:**

Specify the design category that the issue and resolution address (e.g., data design, content structure, component structure, integration, presentation).

**Assumptions:**

Indicate any assumptions that helped shape the decision.

**Constraints:**

Specify any environmental constraints that helped shape the decision (e.g., technology standards, available patterns, project-related issues).

**Alternatives:**

Briefly describe the architectural design alternatives that were considered and why they were rejected.

**Argument:**

State why you chose the resolution over other alternatives.

**Implications:**

Indicate the design consequences of making the decision. How will the resolution affect other architectural design issues? Will the resolution constrain the design in any way?

**Related decisions:**

What other documented decisions are related to this decision?

**Related concerns:**

What other requirements are related to this decision?

**Work products:**

Indicate where this decision will be reflected in the architecture description.

**Notes:**

Reference any team notes or other documentation that was used to make the decision.

**KEY POINT**

A number of different architectural styles may be applicable to a specific genre (also called an application domain).

specific category within the overall software domain. Within each category, you encounter a number of subcategories. For example, within the genre of *buildings*, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on. Within each general style, more specific styles might apply (Section 9.3). Each style would have a structure that can be described using a set of predictable patterns.

In his evolving *Handbook of Software Architecture* [Boo08], Grady Booch suggests the following architectural genres for software-based systems:

- **Artificial intelligence**—Systems that simulate or augment human cognition, locomotion, or other organic processes.
- **Commercial and nonprofit**—Systems that are fundamental to the operation of a business enterprise.
- **Communications**—Systems that provide the infrastructure for transferring and managing data, for connecting users of that data, or for presenting data at the edge of an infrastructure.
- **Content authoring**—Systems that are used to create or manipulate textual or multimedia artifacts.

- **Devices**—Systems that interact with the physical world to provide some point service for an individual.
- **Entertainment and sports**—Systems that manage public events or that provide a large group entertainment experience.
- **Financial**—Systems that provide the infrastructure for transferring and managing money and other securities.
- **Games**—Systems that provide an entertainment experience for individuals or groups.
- **Government**—Systems that support the conduct and operations of a local, state, federal, global, or other political entity.
- **Industrial**—Systems that simulate or control physical processes.
- **Legal**—Systems that support the legal industry.
- **Medical**—Systems that diagnose or heal or that contribute to medical research.
- **Military**—Systems for consultation, communications, command, control, and intelligence (C4I) as well as offensive and defensive weapons.
- **Operating systems**—Systems that sit just above hardware to provide basic software services.
- **Platforms**—Systems that sit just above operating systems to provide advanced services.
- **Scientific**—Systems that are used for scientific research and applications.
- **Tools**—Systems that are used to develop other systems.
- **Transportation**—Systems that control water, ground, air, or space vehicles.
- **Utilities**—Systems that interact with other software to provide some point service.

From the standpoint of architectural design, each genre represents a unique challenge. As an example, consider the software architecture for a game system. Game systems, sometimes called *immersive interactive applications*, require the computation of intensive algorithms, sophisticated computer graphics, streaming multimedia data sources, real-time interactivity via conventional and unconventional inputs, and a variety of other specialized concerns.

Alexandre Francois [Fra03] suggests a software architecture for *Immersipresence*<sup>1</sup> that can be applied for a gaming environment. He describes the architecture in the following manner:

SAI (Software Architecture for Immersipresence) is a new software architecture model for designing, analyzing and implementing applications performing distributed,

<sup>1</sup> Francois uses the term *immersipresence* for immersive, interactive applications.

asynchronous parallel processing of generic data streams. The goal of SAI is to provide a universal framework for the distributed implementation of algorithms and their easy integration into complex systems. . . . The underlying extensible data model and hybrid (shared repository and message-passing) distributed asynchronous parallel processing model allow natural and efficient manipulation of generic data streams, using existing libraries or native code alike. The modularity of the style facilitates distributed code development, testing, and reuse, as well as fast system design and integration, maintenance and evolution.

A detailed discussion of SAI is beyond the scope of this book. However, it is important to recognize that the gaming system genre can be addressed with an architectural style (Section 9.3) that has been specifically designed to address gaming system concerns. If you have further interest, see [Fra03].

### 9.3 ARCHITECTURAL STYLES



#### quote:

"There is at the back of every artist's mind, a pattern or type of architecture."

G. K. Chesterton



What is an architectural style?



#### WebRef

Attribute-based architectural styles (ABAS) can be used as building blocks for software architectures. Information can be obtained at [www.sei.cmu.edu/architecture/abas.html](http://www.sei.cmu.edu/architecture/abas.html).

When a builder uses the phrase "center hall colonial" to describe a house, most people familiar with houses in the United States will be able to conjure a general image of what the house will look like and what the floor plan is likely to be. The builder has used an *architectural style* as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod). But more important, the architectural style is also a template for construction. Further details of the house must be defined, its final dimensions must be specified, customized features may be added, building materials are to be determined, but the style—a "center hall colonial"—guides the builder in his work.

The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses (1) a set of components (e.g., a database, computational modules) that perform a function required by a system; (2) a set of connectors that enable "communication, coordination and cooperation" among components; (3) constraints that define how components can be integrated to form the system; and (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts [Bas03].

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. In the case where an existing architecture is to be reengineered (Chapter 29), the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components [Bos00].

An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture. However, a pattern differs from a style in a number of fundamental ways: (1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety; (2) a pattern imposes a

rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency) [Bos00]; (3) architectural patterns (Section 9.4) tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts). Patterns can be used in conjunction with an architectural style to shape the overall structure of a system. In Section 9.3.1, I consider commonly used architectural styles and patterns for software.

## INFO



### Canonical Architectural Structures

In essence, software architecture represents a structure in which some collection of entities (often called *components*) is connected by a set of defined relationships (often called *connectors*). Both components and connectors are associated with a set of *properties* that allow the designer to differentiate the types of components and connectors that can be used. But what kinds of structures (components, connectors, and properties) can be used to describe an architecture? Bass and Kazman [Bas03] suggest five canonical or foundation architectural structures:

**Functional structure.** Components represent function or processing entities. Connectors represent interfaces that provide the ability to “use” or “pass data to” a component. Properties describe the nature of the components and the organization of the interfaces.

**Implementation structure.** “Components can be packages, classes, objects, procedures, functions, methods, etc., all of which are vehicles for packaging functionality at various levels of abstraction” [Bas03]. Connectors include the ability to pass data and control, share data, “use”, and “is-an-instance-of.” Properties

focus on quality characteristics (e.g., maintainability, reusability) that result when the structure is implemented.

**Concurrency structure.** Components represent “units of concurrency” that are organized as parallel tasks or threads. “Relations [connectors] include synchronizes-with, is-higher-priority-than, sends-data-to, can’t-run-without, and can’t-run-with. Properties relevant to this structure include priority, preemptability, and execution time” [Bas03].

**Physical structure.** This structure is similar to the deployment model developed as part of design. The components are the physical hardware on which software resides. Connectors are the interfaces between hardware components, and properties address capacity, bandwidth, performance, and other attributes.

**Developmental structure.** This structure defines the components, work products, and other information sources that are required as software engineering proceeds. Connectors represent the relationships among work products, and properties identify the characteristics of each item.

Each of these structures presents a different view of software architecture, exposing information that is useful to the software team as modeling and construction proceed.

### 9.3.1 A Brief Taxonomy of Architectural Styles

#### Quote:

“The use of patterns and styles of design is pervasive in engineering disciplines.”

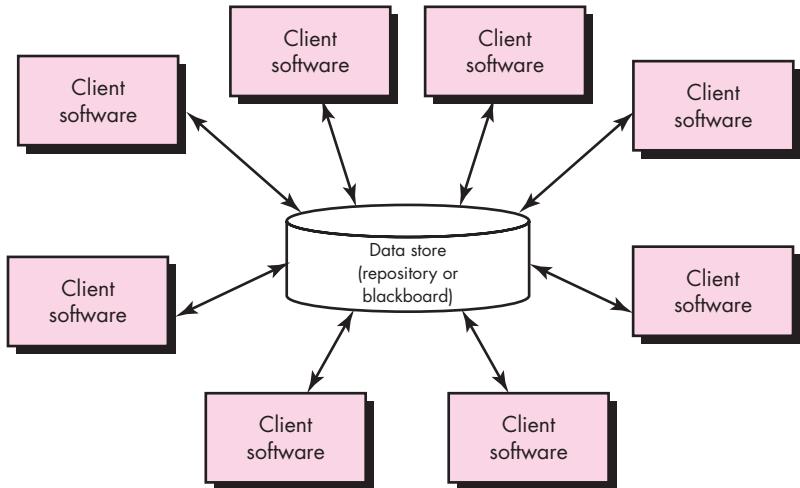
Mary Shaw and David Garlan

Although millions of computer-based systems have been created over the past 60 years, the vast majority can be categorized into one of a relatively small number of architectural styles:

**Data-centered architectures.** A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 9.1 illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a “blackboard”

**FIGURE 9.1**

**Data-centered architecture**



that sends notifications to client software when data of interest to the client changes.

Data-centered architectures promote *integrability* [Bas03]. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

**Data-flow architectures.** This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern (Figure 9.2) has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.

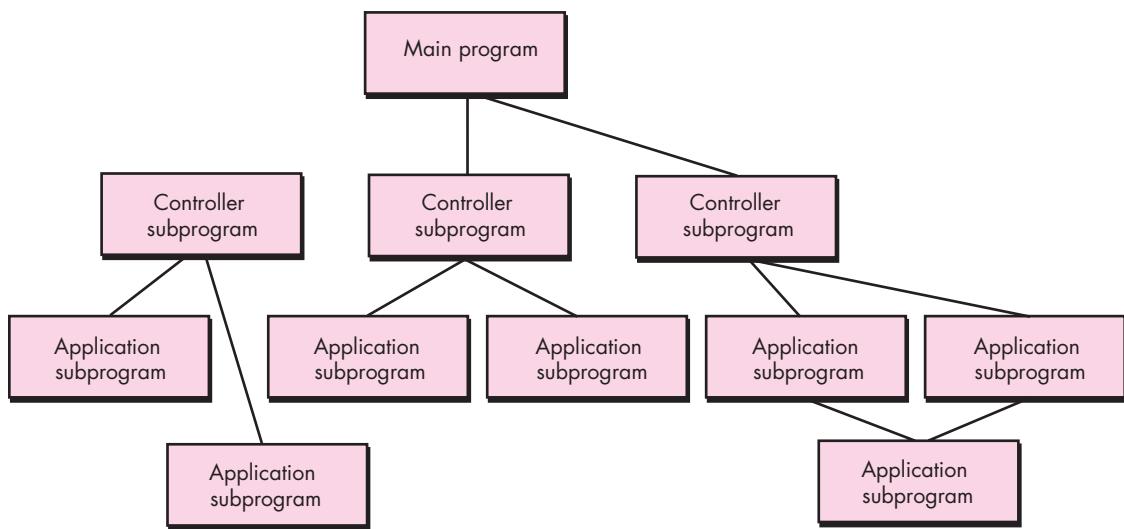
If the data flow degenerates into a single line of transforms, it is termed batch sequential. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

**Call and return architectures.** This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles [Bas03] exist within this category:

- *Main program/subprogram architectures.* This classic program structure decomposes function into a control hierarchy where a “main” program

**FIGURE 9.2**

Data-flow architecture

**FIGURE 9.3** Main program/subprogram architecture

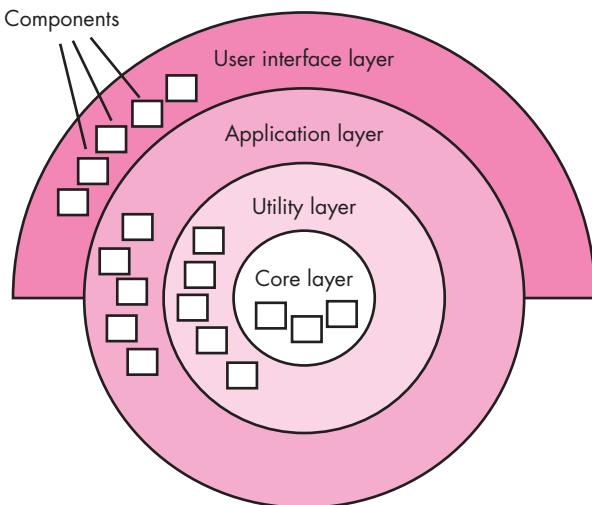
invokes a number of program components that in turn may invoke still other components. Figure 9.3 illustrates an architecture of this type.

- *Remote procedure call architectures.* The components of a main program/subprogram architecture are distributed across multiple computers on a network.

**Object-oriented architectures.** The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

**FIGURE 9.4**

**Layered architecture**



**Layered architectures.** The basic structure of a layered architecture is illustrated in Figure 9.4. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

These architectural styles are only a small subset of those available.<sup>2</sup> Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles can be designed and evaluated. For example, a layered style (appropriate for most systems) can be combined with a data-centered architecture in many database applications.

**Q** **note:**

"Maybe it's in the basement. Let me go upstairs and check."

**M. C. Escher**

### 9.3.2 Architectural Patterns

As the requirements model is developed, you'll notice that the software must address a number of broad problems that span the entire application. For example, the requirements model for virtually every e-commerce application is faced with the following problem: *How do we offer a broad array of goods to a broad array of customers and allow those customers to purchase our goods online?*

<sup>2</sup> See [Bus07], [Gor06], [Roz05], [Bas03], [Bos00], or [Hof00] for a detailed discussion of architectural styles and patterns.

## SAFEHOME



### Choosing an Architectural Style

**The scene:** Jamie's cubicle, as design modeling begins.

**The players:** Jamie and Ed—members of the *SafeHome* software engineering team.

#### The conversation:

**Ed (frowning):** We've been modeling the security function using UML . . . you know classes, relationships, that sort of stuff. So I guess the object-oriented architecture<sup>3</sup> is the right way to go.

**Jamie:** But . . . ?

**Ed:** But . . . I have trouble visualizing what an object-oriented architecture is. I get the call and return architecture, sort of a conventional process hierarchy, but OO . . . I don't know, it seems sort of amorphous.

**Jamie (smiling):** Amorphous, huh?

**Ed:** Yeah . . . what I mean is I can't visualize a real structure, just design classes floating in space.

**Jamie:** Well, that's not true. There are class hierarchies . . . think of the hierarchy (aggregation) we did for the **FloorPlan** object [Figure 8.3]. An OO architecture is a combination of that structure and the interconnections—you know, collaborations—between the classes. We can show it by fully describing the attributes and operations, the messaging that goes on, and the structure of the classes.

**Ed:** I'm going to spend an hour mapping out a call and return architecture; then I'll go back and consider an OO architecture.

**Jamie:** Doug'll have no problem with that. He said that we should consider architectural alternatives. By the way, there's absolutely no reason why both of these architectures couldn't be used in combination with one another.

**Ed:** Good. I'm on it.

The requirements model also defines a context in which this question must be answered. For example, an e-commerce business that sells golf equipment to consumers will operate in a different context than an e-commerce business that sells high-priced industrial equipment to medium and large corporations. In addition, a set of limitations and constraints may affect the way in which you address the problem to be solved.

Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

Earlier in this chapter, I noted that most applications fit within a specific domain or genre and that one or more architectural styles may be appropriate for that genre. For example, the overall architectural style for an application might be call-and-return or object-oriented. But within that style, you will encounter a set of common problems that might best be addressed with specific architectural patterns. Some of these problems and a more complete discussion of architectural patterns are presented in Chapter 12.

<sup>3</sup> It can be argued that the *SafeHome* architecture should be considered at a higher level than the architecture noted. *SafeHome* has a variety of subsystems—home monitoring functionality, the company's monitoring site, and the subsystem running on the owner's PC. Within subsystems, concurrent processes (e.g., those monitoring sensors) and event handling are prevalent. Some architectural decisions at this level are made during product engineering, but architectural design within software engineering may very well have to consider these issues.

### 9.3.3 Organization and Refinement

Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions [Bas03] provide insight into an architectural style:



**Control.** How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system? How is control shared among components? What is the control topology (i.e., the geometric form that the control takes)? Is control synchronized or do components operate asynchronously?

**Data.** How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)? Do data components (e.g., a blackboard or repository) exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system?

These questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

## 9.4 ARCHITECTURAL DESIGN

### quote:

"A doctor can bury his mistakes, but an architect can only advise his client to plant vines."

Frank Lloyd Wright

As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes. An *archetype* is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail. Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived. In the sections that follow we examine each of these architectural design tasks in a bit more detail.

### 9.4.1 Representing the System in Context

#### KEY POINT

Architectural context represents how the software interacts with entities external to its boundaries.

**How do systems interoperate with one another?**

At the architectural design level, a software architect uses an *architectural context diagram* (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in Figure 9.5.

Referring to the figure, systems that interoperate with the *target system* (the system for which an architectural design is to be developed) are represented as

- *Superordinate systems*—those systems that use the target system as part of some higher-level processing scheme.
- *Subordinate systems*—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- *Peer-level systems*—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system).
- *Actors*—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

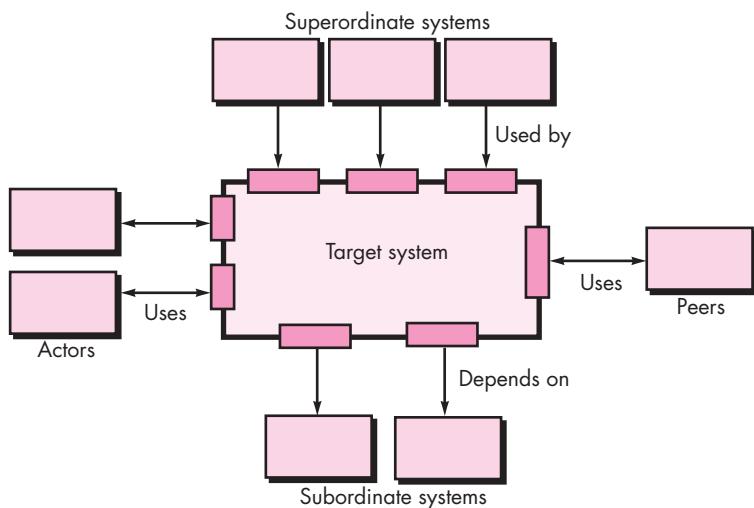
Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

To illustrate the use of the ACD, consider the home security function of the *SafeHome* product. The overall *SafeHome* product controller and the Internet-based system are both superordinate to the security function and are shown above the

**FIGURE 9.5**

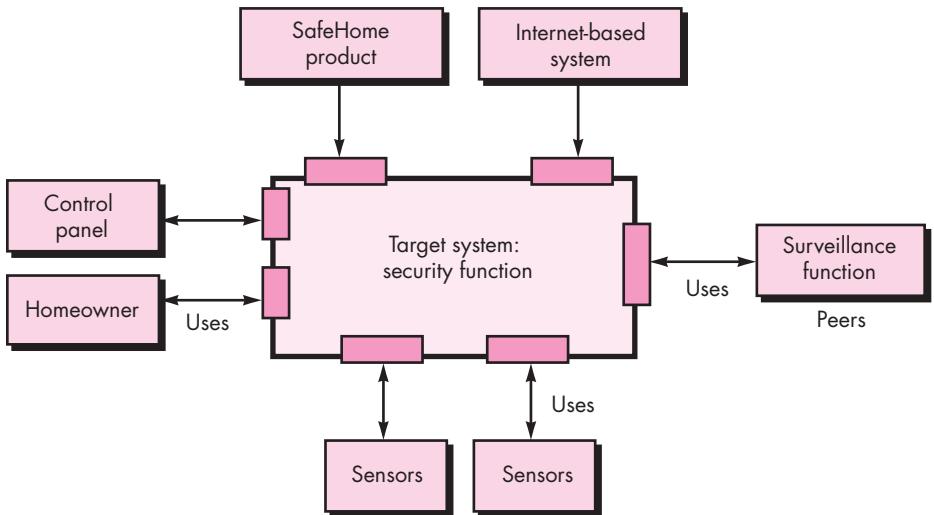
Architectural context diagram

Source: Adapted from [Bos00].



**FIGURE 9.6**

Architectural context diagram for the *SafeHome* security function



function in Figure 9.6. The surveillance function is a *peer system* and uses (is used by) the home security function in later versions of the product. The homeowner and control panels are actors that are both producers and consumers of information used-produced by the security software. Finally, sensors are used by the security software and are shown as subordinate to it.

As part of the architectural design, the details of each interface shown in Figure 9.6 would have to be specified. All data that flow into and out of the target system must be identified at this stage.

#### 9.4.2 Defining Archetypes

**KEY POINT**  
Archetypes are the abstract building blocks of an architectural design.

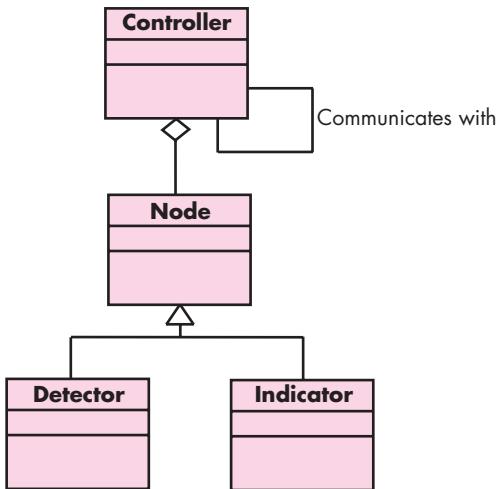
An *archetype* is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model. Continuing the discussion of the *SafeHome* home security function, you might define the following archetypes:

- **Node.** Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors and (2) a variety of alarm (output) indicators.
- **Detector.** An abstraction that encompasses all sensing equipment that feeds information into the target system.

**FIGURE 9.7**

UML relationships for *SafeHome* security function archetypes  
Source: Adapted from [Bos00].



- **Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
- **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

Each of these archetypes is depicted using UML notation as shown in Figure 9.7. Recall that the archetypes form the basis for the architecture but are abstractions that must be further refined as architectural design proceeds. For example, **Detector** might be refined into a class hierarchy of sensors.

### **note:**

"The structure of a software system provides the ecology in which code is born, matures, and dies. A well-designed habitat allows for the successful evolution of all the components needed in a software system."

R. Pattis

### 9.4.3 Refining the Architecture into Components

As the software architecture is refined into components, the structure of the system begins to emerge. But how are these components chosen? In order to answer this question, you begin with the classes that were described as part of the requirements model.<sup>4</sup> These analysis classes represent entities within the application (business) domain that must be addressed within the software architecture. Hence, the application domain is one source for the derivation and refinement of components. Another source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. For example, memory management components, communication components, database components, and task management components are often integrated into the software architecture.

<sup>4</sup> If a conventional (non-object-oriented) approach is chosen, components are derived from the data flow model. I discuss this approach briefly in Section 9.6.

The interfaces depicted in the architecture context diagram (Section 9.4.1) imply one or more specialized components that process the data that flows across the interface. In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.

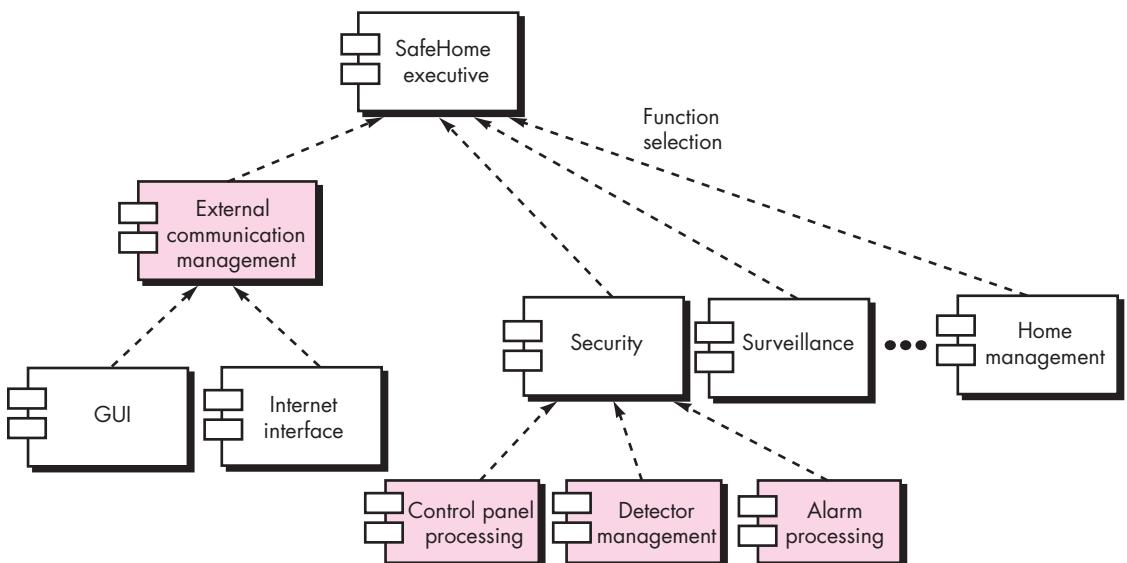
Continuing the *SafeHome* home security function example, you might define the set of top-level components that address the following functionality:

- *External communication management*—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
- *Control panel processing*—manages all control panel functionality.
- *Detector management*—coordinates access to all detectors attached to the system.
- *Alarm processing*—verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall *SafeHome* architecture. Design classes (with appropriate attributes and operations) would be defined for each. It is important to note, however, that the design details of all attributes and operations would not be specified until component-level design (Chapter 10).

The overall architectural structure (represented as a UML component diagram) is illustrated in Figure 9.8. Transactions are acquired by *external communication management* as they move in from components that process the *SafeHome* GUI and the

**FIGURE 9.8** Overall architectural structure for *SafeHome* with top-level components



Internet interface. This information is managed by a *SafeHome* executive component that selects the appropriate product function (in this case security). The *control panel processing* component interacts with the homeowner to arm/disarm the security function. The *detector management* component polls sensors to detect an alarm condition, and the *alarm processing* component produces output when an alarm is detected.

#### 9.4.4 Describing Instantiations of the System

The architectural design that has been modeled to this point is still relatively high level. The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified. However, further refinement (recall that all design is iterative) is still necessary.

To accomplish this, an actual instantiation of the architecture is developed. By this I mean that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

Figure 9.9 illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in Figure 9.8 are elaborated to show additional detail. For example, the *detector management* component interacts with a *scheduler* infrastructure component that implements polling of each *sensor* object used by the security system. Similar elaboration is performed for each of the components represented in Figure 9.8.

### SOFTWARE TOOLS



#### Architectural Design

**Objective:** Architectural design tools model the overall software structure by representing component interface, dependencies and relationships, and interactions.

**Mechanics:** Tool mechanics vary. In most cases, architectural design capability is part of the functionality provided by automated tools for analysis and design modeling.

##### Representative Tools:<sup>5</sup>

*Adalon*, developed by Synthesis Corp. ([www.synthesis.com](http://www.synthesis.com)), is a specialized design tool for the design and

construction of specific Web-based component architectures.

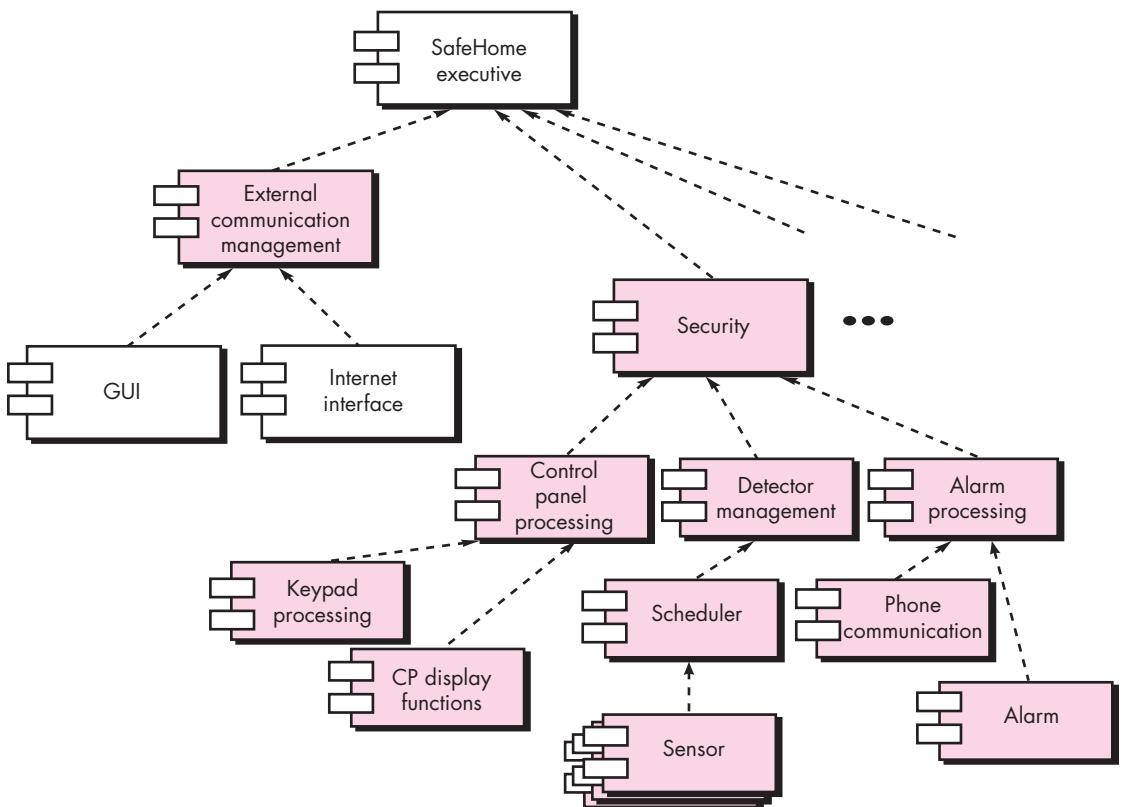
*ObjectifF*, developed by microTOOL GmbH ([www.microtool.de/objectif/en/](http://www.microtool.de/objectif/en/)), is a UML-based design tool that leads to architectures (e.g., Coldfusion, J2EE, Fusebox) amenable to component-based software engineering (Chapter 29).

*Rational Rose*, developed by Rational ([www-306.ibm.com/software/rational/](http://www-306.ibm.com/software/rational/)), is a UML-based design tool that supports all aspects of architectural design.

5 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

**FIGURE 9.9**

An instantiation of the security function with component elaboration



## 9.5 ASSESSING ALTERNATIVE ARCHITECTURAL DESIGNS

In their book on the evaluation of software architectures, Clements and his colleagues [Cle03] state:

To put it bluntly, an architecture is a bet, a wager on the success of a system. Wouldn't it be nice to know in advance if you've placed your bet on a winner, as opposed to waiting until the system is mostly completed before knowing whether it will meet its requirements or not? If you're buying a system or paying for its development, wouldn't you like to have some assurance that it's started off down the right path? If you're the architect yourself, wouldn't you like to have a good way to validate your intuitions and experience, so that you can sleep at night knowing that the trust placed in your design is well founded?

Indeed, answers to these questions would have value. Design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved. In the sections that follow, I present two different approaches for the assessment of alternative architectural designs. The first method uses an iterative method to assess design trade-offs. The second approach applies a pseudo-quantitative technique for assessing design quality.

### 9.5.1 An Architecture Trade-Off Analysis Method

**WebRef**

In-depth information on ATAM can be obtained at:

[www.sei.cmu.edu/  
activities/  
architecture/  
ata\\_method.html](http://www.sei.cmu.edu/activities/architecture/ata_method.html).

The Software Engineering Institute (SEI) has developed an *architecture trade-off analysis method* (ATAM) [Kaz98] that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

1. *Collect scenarios.* A set of use cases (Chapters 5 and 6) is developed to represent the system from the user's point of view.
2. *Elicit requirements, constraints, and environment description.* This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.
3. *Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements.* The architectural style(s) should be described using one of the following architectural views:
  - *Module view* for analysis of work assignments with components and the degree to which information hiding has been achieved.
  - *Process view* for analysis of system performance.
  - *Data flow view* for analysis of the degree to which the architecture meets functional requirements.
4. *Evaluate quality attributes by considering each attribute in isolation.* The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.
5. *Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.* This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed *sensitivity points*.
6. *Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.* The SEI describes this approach in the following manner [Kaz98]:

Once the architectural sensitivity points have been determined, finding trade-off points is simply the identification of architectural elements to which multiple attributes are sensitive. For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). . . . The number of servers, then, is a trade-off point with respect to this architecture.

These six steps represent the first ATAM iteration. Based on the results of steps 5 and 6, some architecture alternatives may be eliminated, one or more of the remaining

architectures may be modified and represented in more detail, and then the ATAM steps are reapplied.<sup>6</sup>

## SAFEHOME



### Architecture Assessment

**The scene:** Doug Miller's office as architectural design modeling proceeds.

**The players:** Vinod, Jamie, and Ed—members of the *SafeHome* software engineering team and Doug Miller, manager of the software engineering group.

#### The conversation:

**Doug:** I know you guys are deriving a couple of different architectures for the *SafeHome* product, and that's a good thing. I guess my question is, how are we going to choose the one that's best?

**Ed:** I'm working on a call and return style and then either Jamie or I are going to derive an OO architecture.

**Doug:** Okay, and how do we choose?

**Jamie:** I took a CS course in design in my senior year, and I remember that there are a number of ways to do it.

**Vinod:** There are, but they're a bit academic. Look, I think we can do our assessment and choose the right one using use cases and scenarios.

**Doug:** Isn't that the same thing?

**Vinod:** Not when you're talking about architectural assessment. We already have a complete set of use cases. So we apply each to both architectures and see how the

system reacts, how components and connectors work in the use case context.

**Ed:** That's a good idea. Makes sure we didn't leave anything out.

**Vinod:** True, but it also tells us whether the architectural design is convoluted, whether the system has to twist itself into a pretzel to get the job done.

**Jamie:** Scenarios aren't just another name for use cases.

**Vinod:** No, in this case a scenario implies something different.

**Doug:** You're talking about a quality scenario or a change scenario, right?

**Vinod:** Yes. What we do is go back to the stakeholders and ask them how *SafeHome* is likely to change over the next, say, three years. You know, new versions, features, that sort of thing. We build a set of change scenarios. We also develop a set of quality scenarios that define the attributes we'd like to see in the software architecture.

**Jamie:** And we apply them to the alternatives.

**Vinod:** Exactly. The style that handles the use cases and scenarios best is the one we choose.

### 9.5.2 Architectural Complexity

A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the system. Zhao [Zha98] suggests three types of dependencies:

*Sharing dependencies* represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers. For example, for two components **u** and **v**, if **u** and **v** refer to the same global data, then there exists a shared dependence relationship between **u** and **v**.

*Flow dependencies* represent dependence relationships between producers and consumers of resources. For example, for two components **u** and **v**, if **u** must complete before

<sup>6</sup> The *Software Architecture Analysis Method* (SAAM) is an alternative to ATAM and is well-worth examining by those readers interested in architectural analysis. A paper on SAAM can be downloaded from [www.sei.cmu.edu/publications/articles/saam-metho-propert-sas.html](http://www.sei.cmu.edu/publications/articles/saam-metho-propert-sas.html).

control flows into **v** (prerequisite), or if **u** communicates with **v** by parameters, then there exists a flow dependence relationship between **u** and **v**.

*Constrained dependencies* represent constraints on the relative flow of control among a set of activities. For example, for two components **u** and **v**, **u** and **v** cannot execute at the same time (mutual exclusion), then there exists a constrained dependence relationship between **u** and **v**.

The sharing and flow dependencies noted by Zhao are similar to the concept of coupling discussed in Chapter 8. Coupling is an important design concept that is applicable at the architectural level and at the component level. Simple metrics for evaluating coupling are discussed in Chapter 23.

### 9.5.3 Architectural Description Languages

The architect of a house has a set of standardized tools and notation that allow the design to be represented in an unambiguous, understandable fashion. Although the software architect can draw on UML notation, other diagrammatic forms, and a few related tools, there is a need for a more formal approach to the specification of an architectural design.

*Architectural description language* (ADL) provides a semantics and syntax for describing a software architecture. Hofmann and his colleagues [Hof01] suggest that an ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components. Once descriptive, language-based techniques for architectural design have been established, it is more likely that effective assessment methods for architectures will be established as the design evolves.

## SOFTWARE TOOLS



### Architectural Description Languages

The following summary of a number of important ADLs was prepared by Rickard Land [Lan02] and is reprinted with the author's permission. It should be noted that the first five ADLs listed have been developed for research purposes and are not commercial products.

**Rapide** (<http://poset.stanford.edu/rapide/>) builds on the notion of partial ordered sets, and thus introduces quite new (but seemingly powerful) programming constructs.

**UniCon** ([www.cs.cmu.edu/~UniCon](http://www.cs.cmu.edu/~UniCon)) is "an architectural description language intended to aid designers in defining software architectures in terms of abstractions that they find useful."

**Aesop** ([www.cs.cmu.edu/~able/aesop/](http://www.cs.cmu.edu/~able/aesop/)) addresses the problem of style reuse. With Aesop, it is possible to define styles and use them when constructing an actual system.

**Wright** ([www.cs.cmu.edu/~able/wright/](http://www.cs.cmu.edu/~able/wright/)) is a formal language including the following elements: *components* with *ports*, *connectors* with *roles*, and *glue* to attach roles to ports. Architectural styles can be formalized in the language with predicates, thus allowing for static checks to determine the consistency and completeness of an architecture.

**Acme** ([www.cs.cmu.edu/~acme/](http://www.cs.cmu.edu/~acme/)) can be seen as a second-generation ADL, in that its intention is to identify a kind of least common denominator for ADLs.

**UML** ([www.uml.org/](http://www.uml.org/)) includes many of the artifacts needed for architectural descriptions—processes, nodes, views, etc. For informal descriptions, UML is well suited just because it is a widely understood standard. It, however, lacks the full strength needed for an adequate architectural description.

## 9.6 ARCHITECTURAL MAPPING USING DATA FLOW

The architectural styles discussed in Section 9.3.1 represent radically different architectures. So it should come as no surprise that a comprehensive mapping that accomplishes the transition from the requirements model to a variety of architectural styles does not exist. In fact, there is no practical mapping for some architectural styles, and the designer must approach the translation of requirements to design for these styles in using the techniques discussed in Section 9.4.

To illustrate one approach to architectural mapping, consider the call and return architecture—an extremely common structure for many types of systems. The call and return architecture can reside within other more sophisticated architectures discussed earlier in this chapter. For example, the architecture of one or more components of a client-server architecture might be call and return.

A mapping technique, called *structured design* [You79], is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram (Chapter 7) to software architecture.<sup>7</sup> The transition from information flow (represented as a DFD) to program structure is accomplished as part of a six-step process: (1) the type of information flow is established, (2) flow boundaries are indicated, (3) the DFD is mapped into the program structure, (4) control hierarchy is defined, (5) the resultant structure is refined using design measures and heuristics, and (6) the architectural description is refined and elaborated.

As a brief example of data flow mapping, I present a step-by-step “transform” mapping for a small part of the *SafeHome* security function.<sup>8</sup> In order to perform the mapping, the type of information flow must be determined. One type of information flow is called *transform flow* and exhibits a linear quality. Data flows into the system along an *incoming flow path* where it is transformed from an external world representation into internalized form. Once it has been internalized, it is processed at a *transform center*. Finally, it flows out of the system along an *outgoing flow path* that transforms the data into external world form.<sup>9</sup>

### 9.6.1 Transform Mapping

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. To illustrate this approach, we again consider the *SafeHome* security function.<sup>10</sup> One element of the analysis model is a set of data flow diagrams that describe information flow within

<sup>7</sup> It should be noted that other elements of the requirements model are also used during the mapping method.

<sup>8</sup> A more detailed discussion of structured design is presented within the website that accompanies this book.

<sup>9</sup> Another important type of information flow, *transaction flow*, is not considered in this example, but is addressed in the structured design example presented within the website that accompanies this book.

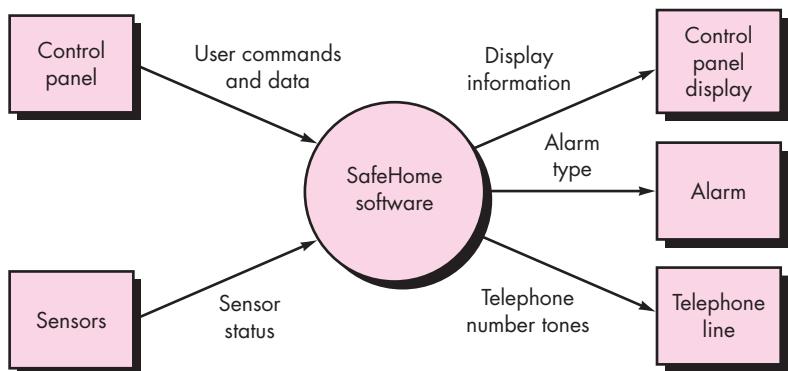
<sup>10</sup> We consider only the portion of the *SafeHome* security function that uses the control panel. Other features discussed throughout this book are not considered here.

the security function. To map these data flow diagrams into a software architecture, you would initiate the following design steps:

**Step 1. Review the fundamental system model.** The fundamental system model or context diagram depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function. Figure 9.10 depicts a level 0 context model, and Figure 9.11 shows refined data flow for the security function.

**FIGURE 9.10**

Context-level DFD for the *SafeHome* security function



**FIGURE 9.11**

Level 1 DFD for the *SafeHome* security function



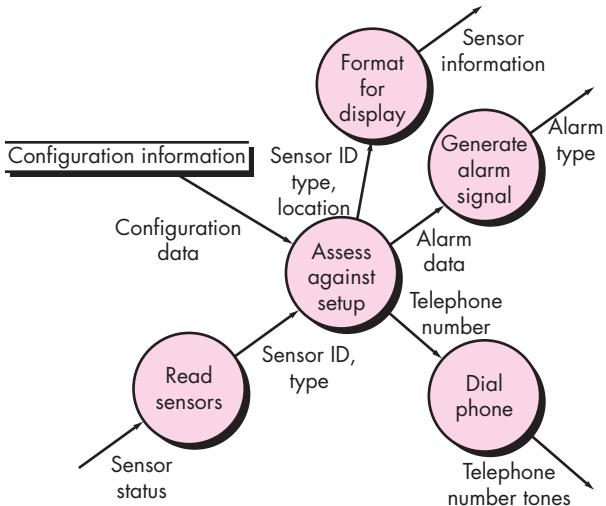


If the DFD is refined further at this time, strive to derive bubbles that exhibit high cohesion.

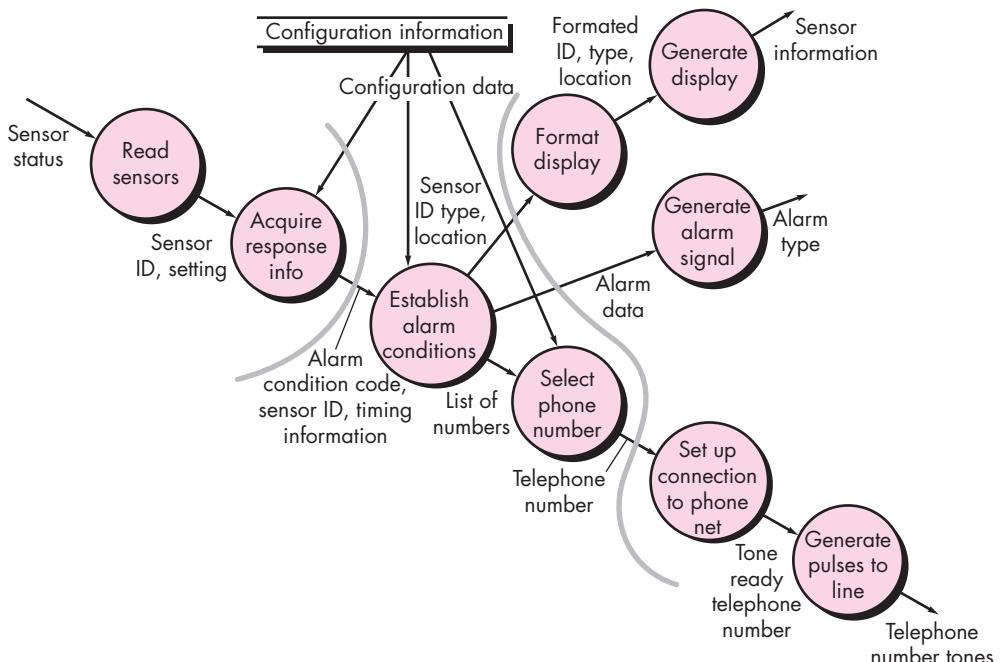
**Step 2. Review and refine data flow diagrams for the software.** Information obtained from the requirements model is refined to produce greater detail. For example, the level 2 DFD for monitor sensors (Figure 9.12) is examined, and a level 3 data flow diagram is derived as shown in Figure 9.13. At level 3, each transform in

**FIGURE 9.12**

Level 2 DFD  
that refines the  
monitor  
sensors  
transform



**FIGURE 9.13** Level 3 DFD for monitor sensors with flow boundaries




**KEY POINT**

You will often encounter both other types of data flow within the same flow-oriented model. The flows are partitioned, and program structure is derived using the appropriate mapping.


**ADVICE**

Vary the location of flow boundaries in an effort to explore alternative program structures. This takes very little time and provides important insight.

the data flow diagram exhibits relatively high cohesion (Chapter 8). That is, the process implied by a transform performs a single, distinct function that can be implemented as a component in the *SafeHome* software. Therefore, the DFD in Figure 9.13 contains sufficient detail for a “first cut” at the design of architecture for the *monitor sensors* subsystem, and we proceed without further refinement.

**Step 3. Determine whether the DFD has transform or transaction flow<sup>11</sup> characteristics.** Evaluating the DFD (Figure 9.13), we see data entering the software along one incoming path and exiting along three outgoing paths. Therefore, an overall transform characteristic will be assumed for information flow.

**Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries.** Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. In fact, alternative design solutions can be derived by varying the placement of flow boundaries. Although care should be taken when boundaries are selected, a variance of one bubble along a flow path will generally have little impact on the final program structure.

Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Figure 9.13. The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to readjust a boundary (e.g., an incoming flow boundary separating *read sensors* and *acquire response info* could be proposed). The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

**Step 5. Perform “first-level factoring.”** The program architecture derived using this mapping results in a top-down distribution of control. *Factoring* leads to a program structure in which top-level components perform decision making and low-level components perform most input, computation, and output work. Middle-level components perform some control and do moderate amounts of work.

When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture) that provides control for incoming, transform, and outgoing information processing. This first-level factoring for the *monitor sensors* subsystem is illustrated in Figure 9.14. A main controller (called *monitor sensors executive*) resides at the top of the program structure and coordinates the following subordinate control functions:

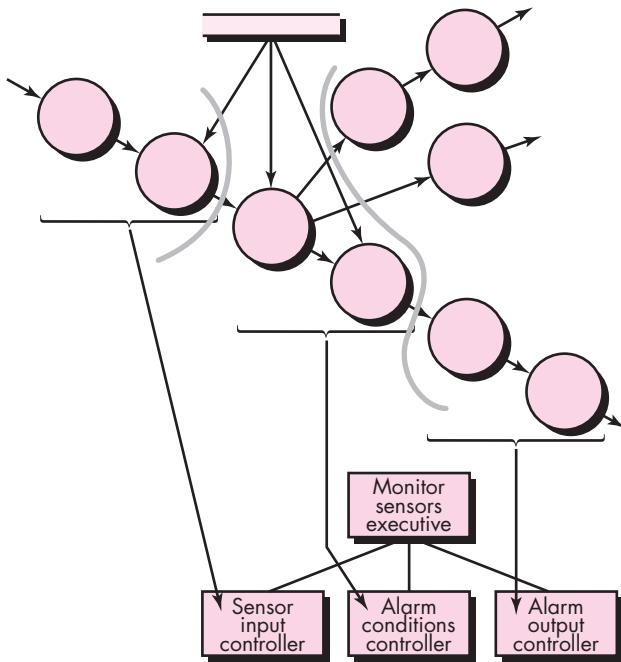
- An incoming information processing controller, called *sensor input controller*, coordinates receipt of all incoming data.

---

<sup>11</sup> In transaction flow, a single data item, called a *transaction*, causes the data flow to branch along one of a number of flow paths defined by the nature of the transaction.

**FIGURE 9.14**

First-level factoring for monitor sensors



*Don't become dogmatic at this stage. It may be necessary to establish two or more controllers for input processing or computation, based on the complexity of the system to be built. If common sense dictates this approach, do it!*

- A transform flow controller, called *alarm conditions controller*, supervises all operations on data in internalized form (e.g., a module that invokes various data transformation procedures).
- An outgoing information processing controller, called *alarm output controller*, coordinates production of output information.

Although a three-pronged structure is implied by Figure 9.14, complex flows in large systems may dictate two or more control modules for each of the generic control functions described previously. The number of modules at the first level should be limited to the minimum that can accomplish control functions and still maintain good functional independence characteristics.



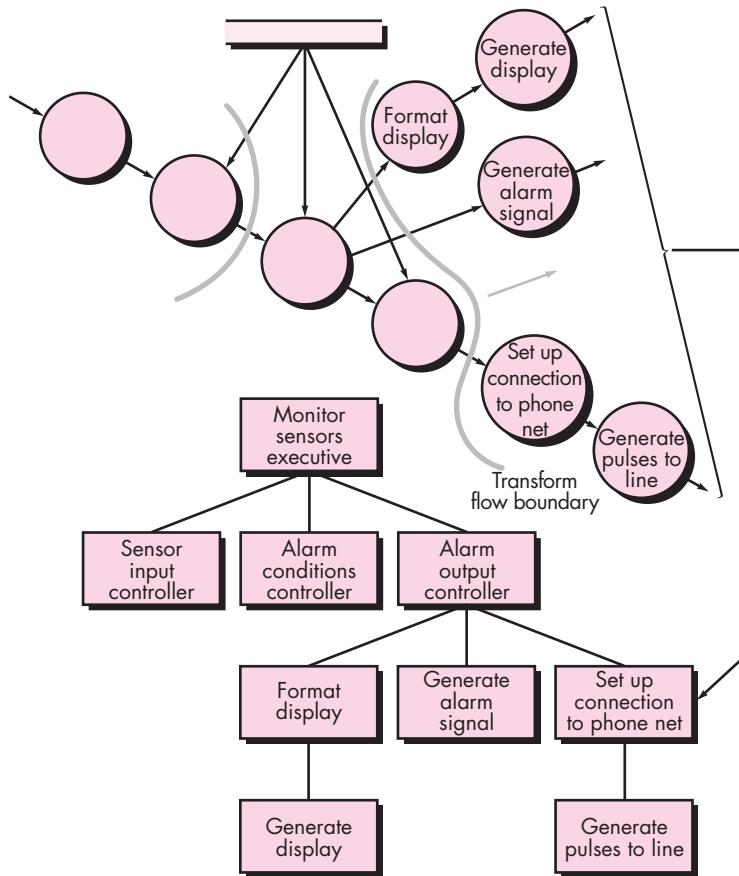
*Eliminate redundant control modules. That is, if a control module does nothing except control one other module, its control function should be imploded to a higher-level module.*

**Step 6. Perform “second-level factoring.”** Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure. The general approach to second-level factoring is illustrated in Figure 9.15.

Although Figure 9.15 illustrates a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur. Two or even three bubbles can be combined and represented as one component, or a single bubble may be expanded to two or more components. Practical considerations and measures

**FIGURE 9.15**

Second-level factoring for monitor sensors



of design quality dictate the outcome of second-level factoring. Review and refinement may lead to changes in this structure, but it can serve as a “first-iteration” design.



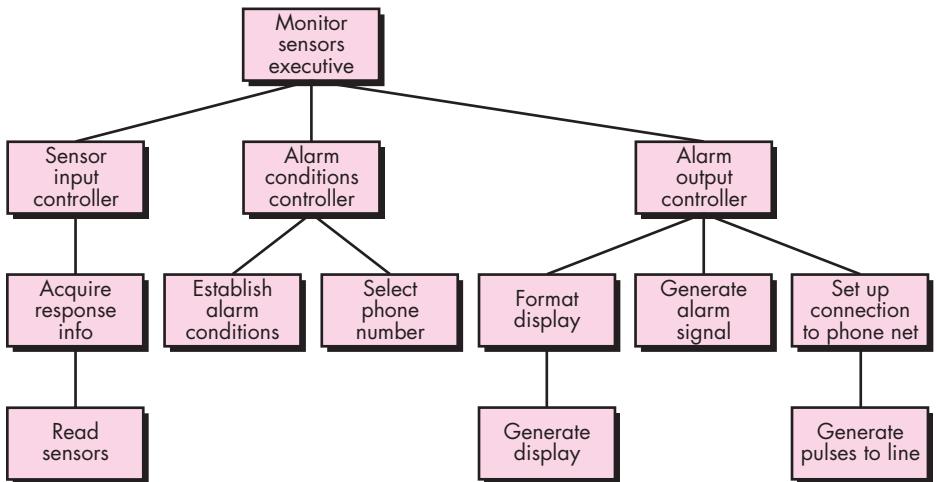
*Keep “worker” modules low in the program structure. This will lead to an architecture that is easier to maintain.*

Second-level factoring for incoming flow follows in the same manner. Factoring is again accomplished by moving outward from the transform center boundary on the incoming flow side. The transform center of *monitor sensors* subsystem software is mapped somewhat differently. Each of the data conversion or calculation transforms of the transform portion of the DFD is mapped into a module subordinate to the transform controller. A completed first-iteration architecture is shown in Figure 9.16.

The components mapped in the preceding manner and shown in Figure 9.16 represent an initial design of software architecture. Although components are named in a manner that implies function, a brief processing narrative (adapted from the process specification developed for a data transformation created during requirements modeling) should be written for each. The narrative describes the

**FIGURE 9.16**

First-iteration  
structure for  
monitor  
sensors



component interface, internal data structures, a functional narrative, and a brief discussion of restrictions and special features (e.g., file input-output, hardware-dependent characteristics, special timing requirements).

**note:**

"Make it as simple as possible. But no simpler."

**Albert Einstein**

**Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.** A first-iteration architecture can always be refined by applying concepts of functional independence (Chapter 8). Components are exploded or imploded to produce sensible factoring, separation of concerns, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

Refinements are dictated by the analysis and assessment methods described briefly in Section 9.5, as well as practical considerations and common sense. There are times, for example, when the controller for incoming data flow is totally unnecessary, when some input processing is required in a component that is subordinate to the transform controller, when high coupling due to global data cannot be avoided, or when optimal structural characteristics cannot be achieved. Software requirements coupled with human judgment is the final arbiter.

The objective of the preceding seven steps is to develop an architectural representation of software. That is, once structure is defined, we can evaluate and refine software architecture by viewing it as a whole. Modifications made at this time require little additional work, yet can have a profound impact on software quality.

You should pause for a moment and consider the difference between the design approach described and the process of "writing programs." If code is the only representation of software, you and your colleagues will have great difficulty evaluating or refining at a global or holistic level and will, in fact, have difficulty "seeing the forest for the trees."

## SAFEHOME



### Refining a First-Cut Architecture

**The scene:** Jamie's cubicle, as design modeling begins.

**The players:** Jamie and Ed—members of the SafeHome software engineering team.

#### The conversation:

[Ed has just completed a first-cut design of the monitor sensors subsystem. He stops in to ask Jamie her opinion.]

**Ed:** So here's the architecture that I derived.

[Ed shows Jamie Figure 9.16, which she studies for a few moments.]

**Jamie:** That's cool, but I think we can do a few things to make it simpler . . . and better.

**Ed:** Such as?

**Jamie:** Well, why did you use the *sensor input controller* component?

**Ed:** Because you need a controller for the mapping.

**Jamie:** Not really. The controller doesn't do much, since we're managing a single flow path for incoming data. We can eliminate the controller with no ill effects.

**Ed:** I can live with that. I'll make the change and . . .

**Jamie (smiling):** Hold up! We can also implode the components *establish alarm conditions* and *select phone number*. The transform controller you show isn't really necessary, and the small decrease in cohesion is tolerable.

**Ed:** Simplification, huh?

**Jamie:** Yep. And while we're making refinements, it would be a good idea to implode the components *format display* and *generate display*. Display formatting for the control panel is simple. We can define a new module called *produce display*.

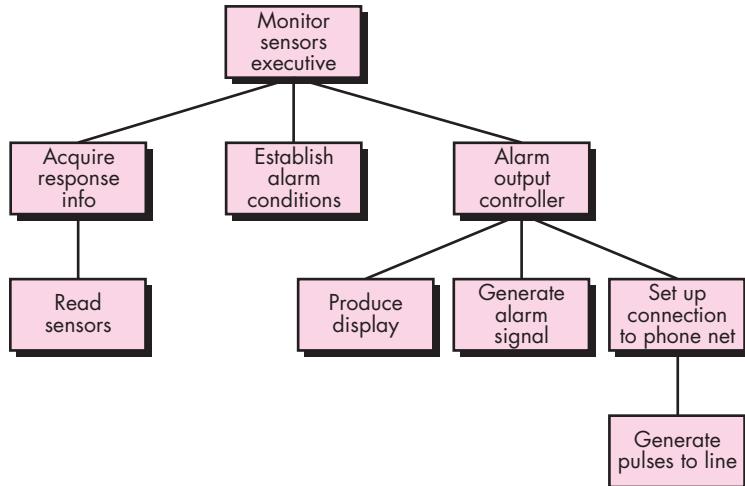
**Ed (sketching):** So this is what you think we should do?"

[Shows Jamie Figure 9.17.]

**Jamie:** It's a good start.

**FIGURE 9.17**

Refined program structure for monitor sensors



### 9.6.2 Refining the Architectural Design

What happens after the architecture has been created?

Any discussion of design refinement should be prefaced with the following comment: "Remember that an 'optimal design' that doesn't work has questionable merit." You should be concerned with developing a representation of software that will meet all functional and performance requirements and merit acceptance based on design measures and heuristics.

Refinement of software architecture during early stages of design is to be encouraged. As I discussed earlier in this chapter, alternative architectural styles may be derived, refined, and evaluated for the “best” approach. This approach to optimization is one of the true benefits derived by developing a representation of software architecture.

It is important to note that structural simplicity often reflects both elegance and efficiency. Design refinement should strive for the smallest number of components that is consistent with effective modularity and the least complex data structure that adequately serves information requirements.

## 9.7 SUMMARY

Software architecture provides a holistic view of the system to be built. It depicts the structure and organization of software components, their properties, and the connections between them. Software components include program modules and the various data representations that are manipulated by the program. Therefore, data design is an integral part of the derivation of the software architecture. Architecture highlights early design decisions and provides a mechanism for considering the benefits of alternative system structures.

A number of different architectural styles and patterns are available to the software engineer and may be applied within a given architectural genre. Each style describes a system category that encompasses a set of components that perform a function required by a system; a set of connectors that enable communication, coordination, and cooperation among components; constraints that define how components can be integrated to form the system; and semantic models that enable a designer to understand the overall properties of a system.

In a general sense, architectural design is accomplished using four distinct steps. First, the system must be represented in context. That is, the designer should define the external entities that the software interacts with and the nature of the interaction. Once context has been specified, the designer should identify a set of top-level abstractions, called archetypes, that represent pivotal elements of the system’s behavior or function. After abstractions have been defined, the design begins to move closer to the implementation domain. Components are identified and represented within the context of an architecture that supports them. Finally, specific instantiations of the architecture are developed to “prove” the design in a real-world context.

As a simple example of architectural design, the mapping method presented in this chapter uses data flow characteristics to derive a commonly used architectural style. A data flow diagram is mapped into program structure using a transform mapping approach. Transform mapping is applied to an information flow that exhibits distinct boundaries between incoming and outgoing data. The DFD is mapped into a structure that allocates control to input, processing, and output along three separately factored module hierarchies. Once an architecture has been derived, it is elaborated and then analyzed using quality criteria.

## PROBLEMS AND POINTS TO PONDER

- 9.1.** Using the architecture of a house or building as a metaphor, draw comparisons with software architecture. How are the disciplines of classical architecture and the software architecture similar? How do they differ?
- 9.2.** Present two or three examples of applications for each of the architectural styles noted in Section 9.3.1.
- 9.3.** Some of the architectural styles noted in Section 9.3.1 are hierarchical in nature and others are not. Make a list of each type. How would the architectural styles that are not hierarchical be implemented?
- 9.4.** The terms *architectural style*, *architectural pattern*, and *framework* (not discussed in this book) are often encountered in discussions of software architecture. Do some research and describe how each of these terms differs from its counterparts.
- 9.5.** Select an application with which you are familiar. Answer each of the questions posed for control and data in Section 9.3.3.
- 9.6.** Research the ATAM (using [Kaz98]) and present a detailed discussion of the six steps presented in Section 9.5.1.
- 9.7.** If you haven't done so, complete Problem 6.6. Use the design methods described in this chapter to develop a software architecture for the PHTRS.
- 9.8.** Using a data flow diagram and a processing narrative, describe a computer-based system that has distinct transform flow characteristics. Define flow boundaries and map the DFD into a software architecture using the technique described in Section 9.6.1.

## FURTHER READINGS AND INFORMATION SOURCES

The literature on software architecture has exploded over the past decade. Books by Gorton (*Essential Software Architecture*, Springer, 2006), Reekie and McAdam (*A Software Architecture Primer*, Angophora Press, 2006), Albin (*The Art of Software Architecture*, Wiley, 2003), and Bass and his colleagues (*Software Architecture in Practice*, 2d ed., Addison-Wesley, 2002) present worthwhile introductions to an intellectually challenging topic area.

Buschman and his colleagues (*Pattern-Oriented Software Architecture*, Wiley, 2007) and Kuchana (*Software Architecture Design Patterns in Java*, Auerbach, 2004) discuss pattern-oriented aspects of architectural design. Rozanski and Woods (*Software Systems Architecture*, Addison-Wesley, 2005), Fowler (*Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003), Clements and his colleagues (*Documenting Software Architecture: View and Beyond*, Addison-Wesley, 2002), Bosch [Bos00], and Hofmeister and his colleagues [Hof00] provide in-depth treatments of software architecture.

Hennessey and Patterson (*Computer Architecture*, 4th ed., Morgan-Kaufmann, 2007) take a distinctly quantitative view of software architectural design issues. Clements and his colleagues (*Evaluating Software Architectures*, Addison-Wesley, 2002) consider the issues associated with the assessment of architectural alternatives and the selection of the best architecture for a given problem domain.

Implementation-specific books on architecture address architectural design within a specific development environment or technology. Marks and Bell (*Service-Oriented Architecture*, Wiley, 2006) discuss a design approach that links business and computational resources with the requirements defined by customers. Stahl and his colleagues (*Model-Driven Software Development*, Wiley, 2006) discuss architecture within the context of domain-specific modeling approaches. Radaideh and Al-ameed (*Architecture of Reliable Web Applications Software*, GI Global, 2007) consider architectures that are appropriate for WebApps. Clements and Northrop (*Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001) address the design of architectures that

support software product lines. Shanley (*Protected Mode Software Architecture*, Addison-Wesley, 1996) provides architectural design guidance for anyone designing PC-based real-time operating systems, multitask operating systems, or device drivers.

Current software architecture research is documented yearly in the *Proceedings of the International Workshop on Software Architecture*, sponsored by the ACM and other computing organizations, and the *Proceedings of the International Conference on Software Engineering*.

A wide variety of information sources on architectural design are available on the Internet. An up-to-date list of World Wide Web references that are relevant to architectural design can be found at the SEPA website: [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm).

## CHAPTER

# 10

## COMPONENT-LEVEL DESIGN

### KEY CONCEPTS

cohesion . . . . .	286
components	
classifying . . . . .	307
adaptation . . . . .	305
composition . . . . .	305
object-oriented . . . . .	277
qualification . . . . .	304
traditional . . . . .	298
WebApp . . . . .	296
component-based development . . . . .	303
content design . . . . .	297
coupling . . . . .	288

### QUICK LOOK

**What is it?** A complete set of software components is defined during architectural design. But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to code. Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.

**Who does it?** A software engineer performs component-level design.

**Why is it important?** You have to be able to determine whether the software will work before you build it. The component-level design represents the software in a way that allows you to review the details of the design for correctness and consistency with other design representations (i.e., the data, architectural, and interface designs). It provides a means for assessing whether data structures, interfaces, and algorithms will work.

**What are the steps?** Design representations of data, architecture, and interfaces form the

**C**omponent-level design occurs after the first iteration of architectural design has been completed. At this stage, the overall data and program structure of the software has been established. The intent is to translate the design model into operational software. But the level of abstraction of the existing design model is relatively high, and the abstraction level of the operational program is low. The translation can be challenging, opening the door to the introduction of subtle errors that are difficult to find and correct in later stages of the software process. In a famous lecture, Edsger Dijkstra, a major contributor to our understanding of software design, stated [Dij72]:

Software seems to be different from many other products, where as a rule higher quality implies a higher price. Those who want really reliable software will discover that they must find a means of avoiding the majority of bugs to start with, and as a result,

foundation for component-level design. The class definition or processing narrative for each component is translated into a detailed design that makes use of diagrammatic or text-based forms that specify internal data structures, local interface detail, and processing logic. Design notation encompasses UML diagrams and supplementary forms. Procedural design is specified using a set of structured programming constructs. It is often possible to acquire existing reusable software components rather than building new ones.

**What is the work product?** The design for each component, represented in graphical, tabular, or text-based notation, is the primary work product produced during component-level design.

**How do I ensure that I've done it right?** A design review is conducted. The design is examined to determine whether data structures, interfaces, processing sequences, and logical conditions are correct and will produce the appropriate data or control transformation allocated to the component during earlier design steps.

<b>design</b>	
<b>guidelines</b>	<b>.....285</b>
<b>domain</b>	
<b>engineering</b>	<b>....303</b>
<b>tabular design</b>	
<b>notation</b>	<b>.....300</b>

the programming process will become cheaper . . . effective programmers . . . should not waste their time debugging—they should not introduce bugs to start with.

Although these words were spoken many years ago, they remain true today. As you translate the design model into source code, you should follow a set of design principles that not only perform the translation but also do not “introduce bugs to start with.”

It is possible to represent the component-level design using a programming language. In essence, the program is created using the architectural design model as a guide. An alternative approach is to represent the component-level design using some intermediate (e.g., graphical, tabular, or text-based) representation that can be translated easily into source code. Regardless of the mechanism that is used to represent the component-level design, the data structures, interfaces, and algorithms defined should conform to a variety of well-established design guidelines that help you to avoid errors as the procedural design evolves. In this chapter, I examine these design guidelines and the methods available for achieving them.

## 10.1 WHAT IS A COMPONENT?



### QUOTE:

“The details are not the details. They make the design.”

Charles Eames

A *component* is a modular building block for computer software. More formally, the *OMG Unified Modeling Language Specification* [OMG03a] defines a component as “. . . a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”

As we discussed in Chapter 9, components populate the software architecture and, as a consequence, play a role in achieving the objectives and requirements of the system to be built. Because components reside within the software architecture, they must communicate and collaborate with other components and with entities (e.g., other systems, devices, people) that exist outside the boundaries of the software.

The true meaning of the term *component* will differ depending on the point of view of the software engineer who uses it. In the sections that follow, I examine three important views of what a component is and how it is used as design modeling proceeds.

### 10.1.1 An Object-Oriented View



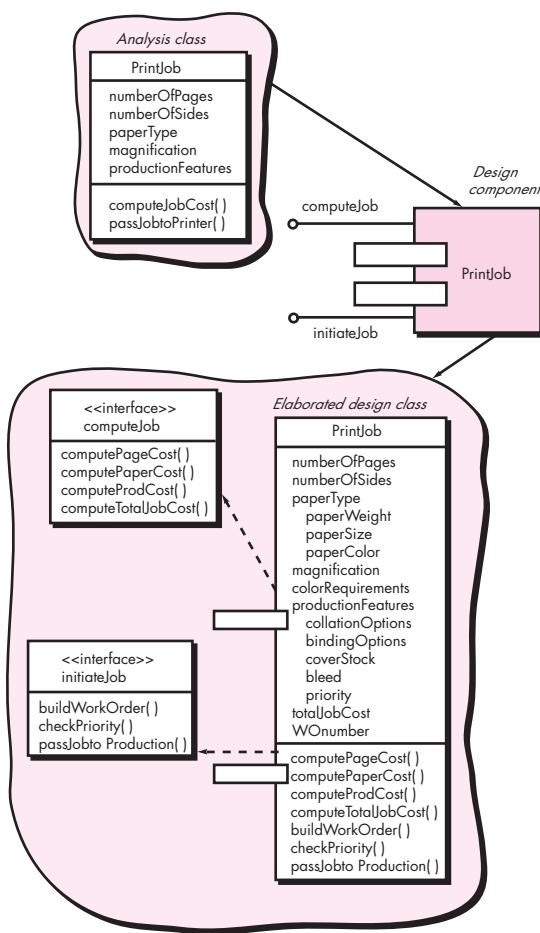
From an object-oriented viewpoint, a component is a set of collaborating classes.

In the context of object-oriented software engineering, a component contains a set of collaborating classes.<sup>1</sup> Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation. As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined. To accomplish this, you begin with the requirements model and elaborate analysis classes (for components that relate to the problem domain) and infrastructure classes (for components that provide support services for the problem domain).

<sup>1</sup> In some cases, a component may contain a single class.

**FIGURE 10.1**

Elaboration of a design component



To illustrate this process of design elaboration, consider software to be built for a sophisticated print shop. The overall intent of the software is to collect the customer's requirements at the front counter, cost a print job, and then pass the job on to an automated production facility. During requirements engineering, an analysis class called **PrintJob** was derived. The attributes and operations defined during analysis are noted at the top of Figure 10.1. During architectural design, **PrintJob** is defined as a component within the software architecture and is represented using the shorthand UML notation<sup>2</sup> shown in the middle right of the figure. Note that **PrintJob** has two interfaces, *computeJob*, which provides job costing capability, and *initiateJob*, which passes the job along to the production facility. These are represented using the "lollipop" symbols shown to the left of the component box.

2 Readers who are unfamiliar with UML notation should refer to Appendix 1.



*Recall that analysis modeling and design modeling are both iterative actions. Elaborating the original analysis class may require additional analysis steps, which are then followed with design modeling steps to represent the elaborated design class (the details of the component).*

Component-level design begins at this point. The details of the component **PrintJob** must be elaborated to provide sufficient information to guide implementation. The original analysis class is elaborated to flesh out all attributes and operations required to implement the class as the component **PrintJob**. Referring to the lower right portion of Figure 10.1, the elaborated design class **PrintJob** contains more detailed attribute information as well as an expanded description of operations required to implement the component. The interfaces *computeJob* and *initiateJob* imply communication and collaboration with other components (not shown here). For example, the operation *computePageCost()* (part of the *computeJob* interface) might collaborate with a **PricingTable** component that contains job pricing information. The *checkPriority()* operation (part of the *initiateJob* interface) might collaborate with a **JobQueue** component to determine the types and priorities of jobs currently awaiting production.

This elaboration activity is applied to every component defined as part of the architectural design. Once it is completed, further elaboration is applied to each attribute, operation, and interface. The data structures appropriate for each attribute must be specified. In addition, the algorithmic detail required to implement the processing logic associated with each operation is designed. This procedural design activity is discussed later in this chapter. Finally, the mechanisms required to implement the interface are designed. For object-oriented software, this may encompass the description of all messaging that is required to effect communication between objects within the system.

### 10.1.2 The Traditional View

In the context of traditional software engineering, a component is a functional element of a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it. A traditional component, also called a *module*, resides within the software architecture and serves one of three important roles: (1) a *control component* that coordinates the invocation of all other problem domain components, (2) a *problem domain component* that implements a complete or partial function that is required by the customer, or (3) an *infrastructure component* that is responsible for functions that support the processing required in the problem domain.

Like object-oriented components, traditional software components are derived from the analysis model. In this case, however, the data flow-oriented element of the analysis model serves as the basis for the derivation. Each transform (bubble) represented at the lowest levels of the data flow diagram is mapped (Section 9.6) into a module hierarchy. Control components (modules) reside near the top of the hierarchy (program architecture), and problem domain components tend to reside toward the bottom of the hierarchy. To achieve effective modularity, design concepts like functional independence (Chapter 8) are applied as components are elaborated.

To illustrate this process of design elaboration for traditional components, again consider software to be built for a sophisticated print shop. A set of data flow diagrams

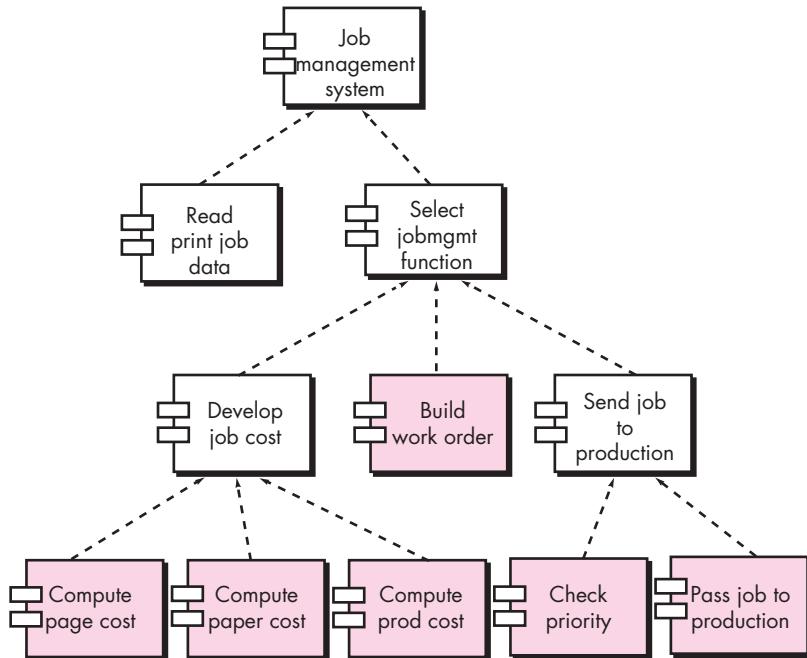


*"A complex system that works is invariably found to have evolved from a simple system that worked."*

**John Gall**

**FIGURE 10.2**

Structure chart  
for a traditional system



would be derived during requirements modeling. Assume that these are mapped into an architecture shown in Figure 10.2. Each box represents a software component. Note that the shaded boxes are equivalent in function to the operations defined for the **PrintJob** class discussed in Section 10.1.1. In this case, however, each operation is represented as a separate module that is invoked as shown in the figure. Other modules are used to control processing and are therefore control components.

During component-level design, each module in Figure 10.2 is elaborated. The module interface is defined explicitly. That is, each data or control object that flows across the interface is represented. The data structures that are used internal to the module are defined. The algorithm that allows the module to accomplish its intended function is designed using the stepwise refinement approach discussed in Chapter 8. The behavior of the module is sometimes represented using a state diagram.

To illustrate this process, consider the module *ComputePageCost*. The intent of this module is to compute the printing cost per page based on specifications provided by the customer. Data required to perform this function are: **number of pages in the document**, **total number of documents to be produced**, **one- or two-side printing**, **color requirements**, and **size requirements**. These data are passed to *ComputePageCost* via the module's interface. *ComputePageCost* uses these data to determine a page cost that is based on the size and complexity of the job—a function of all data passed to the module via the interface. Page cost is inversely proportional to the size of the job and directly proportional to the complexity of the job.



*As the design for each software component is elaborated, the focus shifts to the design of specific data structures and procedural design to manipulate the data structures. However, don't forget the architecture that must house the components or the global data structures that may serve many components.*

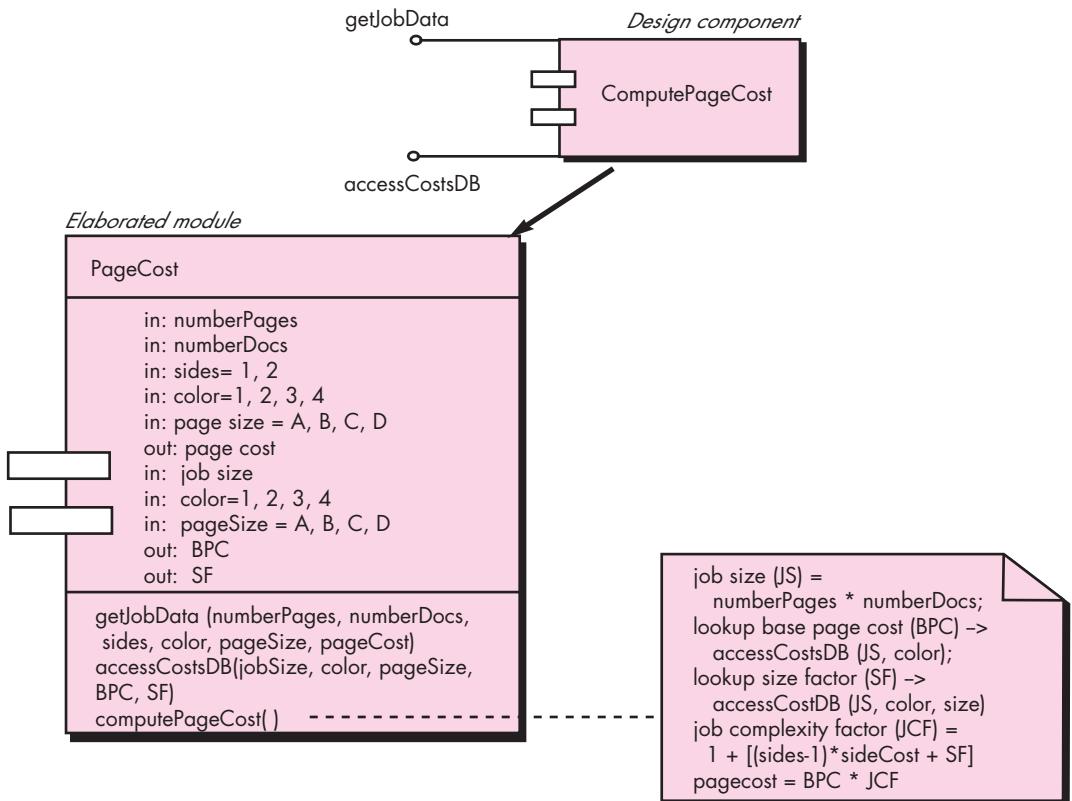
**FIGURE 10.3** Component-level design for *ComputePageCost*

Figure 10.3 represents the component-level design using a modified UML notation. The *ComputePageCost* module accesses data by invoking the module *getJobData*, which allows all relevant data to be passed to the component, and a database interface, *accessCostsDB*, which enables the module to access a database that contains all printing costs. As design continues, the *ComputePageCost* module is elaborated to provide algorithm detail and interface detail (Figure 10.3). Algorithm detail can be represented using the pseudocode text shown in the figure or with a UML activity diagram. The interfaces are represented as a collection of input and output data objects or items. Design elaboration continues until sufficient detail is provided to guide construction of the component.

### 10.1.3 A Process-Related View

The object-oriented and traditional views of component-level design presented in Sections 10.1.1 and 10.1.2 assume that the component is being designed from scratch. That is, you have to create a new component based on specifications derived from the requirements model. There is, of course, another approach.

Over the past two decades, the software engineering community has emphasized the need to build systems that make use of existing software components or design patterns. In essence, a catalog of proven design or code-level components is made available to you as design work proceeds. As the software architecture is developed, you choose components or design patterns from the catalog and use them to populate the architecture. Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration they require are all available to you. I discuss some of the important aspects of component-based software engineering (CBSE) later in Section 10.6.

**INFO**

### Component-Based Standards and Frameworks

One of the key elements that lead to the success or failure of CBSE is the availability of component-based standards, sometimes called middleware. Middleware is a collection of infrastructure components that enable problem domain components to communicate with one another across a network or within a complex system. Software engineers who want to use component-based development as their software process can choose from among the following standards:

**OMG CORBA**—[www.corba.org/](http://www.corba.org/)  
**Microsoft COM**—[www.microsoft.com/com/tech/complus.asp](http://www.microsoft.com/com/tech/complus.asp)  
**Microsoft .NET**—<http://msdn2.microsoft.com/en-us/netframework/default.aspx>  
**Sun JavaBeans**—<http://java.sun.com/products/ejb/>

The websites noted present a wide array of tutorials, white papers, tools, and general resources on these important middleware standards.

## 10.2 DESIGNING CLASS-BASED COMPONENTS

As I have already noted, component-level design draws on information developed as part of the requirements model (Chapters 6 and 7) and represented as part of the architectural model (Chapter 9). When an object-oriented software engineering approach is chosen, component-level design focuses on the elaboration of problem domain specific classes and the definition and refinement of infrastructure classes contained in the requirements model. The detailed description of the attributes, operations, and interfaces used by these classes is the design detail required as a precursor to the construction activity.

### 10.2.1 Basic Design Principles

Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied. The underlying motivation for the application of these principles is to create designs that are more amenable to change and to reduce the propagation of side effects when changes do occur. You can use these principles as a guide as each software component is developed.

**The Open-Closed Principle (OCP).** “A module [component] should be open for extension but closed for modification” [Mar00]. This statement seems to be a

**FIGURE 10.4**

Following the OCP



contradiction, but it represents one of the most important characteristics of a good component-level design. Stated simply, you should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal (code or logic-level) modifications to the component itself. To accomplish this, you create abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.

For example, assume that the *SafeHome* security function makes use of a **Detector** class that must check the status of each type of security sensor. It is likely that as time passes, the number and types of security sensors will grow. If internal processing logic is implemented as a sequence of if-then-else constructs, each addressing a different sensor type, the addition of a new sensor type will require additional internal processing logic (still another if-then-else). This is a violation of OCP.

One way to accomplish OCP for the **Detector** class is illustrated in Figure 10.4. The *sensor* interface presents a consistent view of sensors to the detector component. If a new type of sensor is added no change is required for the **Detector** class (component). The OCP is preserved.

## SAFEHOME



### *The OCP in Action*

**The scene:** Vinod's cubicle.

**The players:** Vinod and Shakira—members of the *SafeHome* software engineering team.

#### **The conversation:**

**Vinod:** I just got a call from Doug [the team manager]. He says marketing wants to add a new sensor.

**Shakira (smirking):** Not again, jeez!

**Vinod:** Yeah . . . and you're not going to believe what these guys have come up with.

**Shakira:** Amaze me.

**Vinod (laughing):** They call it a doggie angst sensor.

**Shakira:** Say what?

**Vinod:** It's for people who leave their pets home in apartments or condos or houses that are close to one

another. The dog starts to bark. The neighbor gets angry and complains. With this sensor, if the dog barks for more than, say, a minute, the sensor sets a special alarm mode that calls the owner on his or her cell phone.

**Shakira:** You're kidding me, right?

**Vinod:** Nope. Doug wants to know how much time it's going to take to add it to the security function.

**Shakira (thinking a moment):** Not much . . . look. [She shows Vinod Figure 10.4] We've isolated the actual sensor classes behind the **sensor** interface. As long as we have specs for the doggie sensor, adding it should be a piece of cake. Only thing I'll have to do is create an appropriate component . . . uh, class, for it. No change to the **Detector** component at all.

**Vinod:** So I'll tell Doug it's no big deal.

**Shakira:** Knowing Doug, he'll keep us focused and not deliver the doggie thing until the next release.

**Vinod:** That's not a bad thing, but you can implement now if he wants you to?

**Shakira:** Yeah, the way we designed the interface lets me do it with no hassle.

**Vinod (thinking a moment):** Have you ever heard of the open-closed principle?

**Shakira (shrugging):** Never heard of it.

**Vinod (smiling):** Not a problem.

**The Liskov Substitution Principle (LSP).** “*Subclasses should be substitutable for their base classes*” [Mar00]. This design principle, originally proposed by Barbara Liskov [Lis88], suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead. LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it. In the context of this discussion, a “contract” is a *precondition* that must be true before the component uses a base class and a *postcondition* that should be true after the component uses a base class. When you create derived classes, be sure they conform to the pre- and postconditions.



If you dispense with design and hack out code, just remember that code is the ultimate “concretion.” You’re violating DIP.

**Dependency Inversion Principle (DIP).** “*Depend on abstractions. Do not depend on concretions*” [Mar00]. As we have seen in the discussion of the OCP, abstractions are the place where a design can be extended without great complication. The more a component depends on other concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend.

**The Interface Segregation Principle (ISP).** “*Many client-specific interfaces are better than one general purpose interface*” [Mar00]. There are many instances in which multiple client components use the operations provided by a server class. ISP suggests that you should create a specialized interface to serve each major category of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface for that client. If multiple clients require the same operations, it should be specified in each of the specialized interfaces.

As an example, consider the **FloorPlan** class that is used for the *SafeHome* security and surveillance functions (Chapter 6). For the security functions, **FloorPlan** is used only during configuration activities and uses the operations *placeDevice()*, *showDevice()*, *groupDevice()*, and *removeDevice()* to place, show, group, and remove sensors from the floor plan. The *SafeHome* surveillance function uses the four

operations noted for security, but also requires special operations to manage cameras: *showFOV()* and *showDeviceID()*. Hence, the ISP suggests that client components from the two *SafeHome* functions have specialized interfaces defined for them. The interface for security would encompass only the operations *placeDevice()*, *showDevice()*, *groupDevice()*, and *removeDevice()*. The interface for surveillance would incorporate the operations *placeDevice()*, *showDevice()*, *groupDevice()*, and *removeDevice()*, along with *showFOV()* and *showDeviceID()*.

Although component-level design principles provide useful guidance, components themselves do not exist in a vacuum. In many cases, individual components or classes are organized into subsystems or packages. It is reasonable to ask how this packaging activity should occur. Exactly how should components be organized as the design proceeds? Martin [Mar00] suggests additional packaging principles that are applicable to component-level design:



Designing components for reuse requires more than good technical design. It also requires effective configuration control mechanisms (Chapter 22).

**The Release Reuse Equivalency Principle (REP).** “*The granule of reuse is the granule of release*” [Mar00]. When classes or components are designed for reuse, there is an implicit contract that is established between the developer of the reusable entity and the people who will use it. The developer commits to establish a release control system that supports and maintains older versions of the entity while the users slowly upgrade to the most current version. Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve.

**The Common Closure Principle (CCP).** “*Classes that change together belong together*” [Mar00]. Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioral area. When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads to more effective change control and release management.

**The Common Reuse Principle (CRP).** “*Classes that aren’t reused together should not be grouped together*” [Mar00]. When one or more classes within a package changes, the release number of the package changes. All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operates without incident. If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed. This will precipitate unnecessary integration and testing. For this reason, only classes that are reused together should be included within a package.

### 10.2.2 Component-Level Design Guidelines

In addition to the principles discussed in Section 10.2.1, a set of pragmatic design guidelines can be applied as component-level design proceeds. These guidelines apply to components, their interfaces, and the dependencies and inheritance

characteristics that have an impact on the resultant design. Ambler [Amb02b] suggests the following guidelines:



**Components.** Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model. For example, the class name **FloorPlan** is meaningful to everyone reading it regardless of technical background. On the other hand, infrastructure components or elaborated component-level classes should be named to reflect implementation-specific meaning. If a linked list is to be managed as part of the **FloorPlan** implementation, the operation *manageList()* is appropriate, even if a non-technical person might misinterpret it.<sup>3</sup>

You can choose to use stereotypes to help identify the nature of components at the detailed design level. For example, `<<infrastructure>>` might be used to identify an infrastructure component, `<<database>>` could be used to identify a database that services one or more design classes or the entire system; `<<table>>` can be used to identify a table within a database.

**Interfaces.** Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OCP). However, unfettered representation of interfaces tends to complicate component diagrams. Ambler [Amb02c] recommends that (1) lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach, when diagrams grow complex; (2) for consistency, interfaces should flow from the left-hand side of the component box; (3) only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available. These recommendations are intended to simplify the visual nature of UML component diagrams.

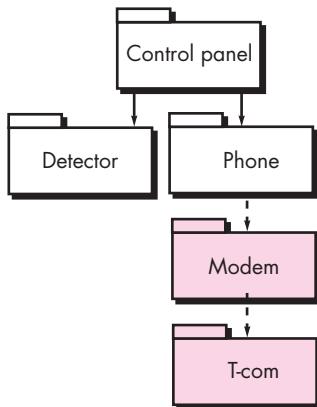
**Dependencies and Inheritance.** For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes). In addition, component interdependencies should be represented via interfaces, rather than by representation of a component-to-component dependency. Following the philosophy of the OCP, this will help to make the system more maintainable.

### 10.2.3 Cohesion

In Chapter 8, I described cohesion as the “single-mindedness” of a component. Within the context of component-level design for object-oriented systems, *cohesion*

---

<sup>3</sup> It is unlikely that someone from marketing or the customer organization (a nontechnical type) would examine detailed design information.

**FIGURE 10.5****Layer cohesion**

implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself. Lethbridge and Lagani  re [Let01] define a number of different types of cohesion (listed in order of the level of the cohesion<sup>4</sup>):

**Functional.** Exhibited primarily by operations, this level of cohesion occurs when a component performs a targeted computation and then returns a result.



*Although an understanding of the various levels of cohesion is instructive, it is more important to be aware of the general concept as you design components. Keep cohesion as high as is possible.*

**Layer.** Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers. Consider, for example, the *SafeHome* security function requirement to make an outgoing phone call if an alarm is sensed. It might be possible to define a set of layered packages as shown in Figure 10.5. The shaded packages contain infrastructure components. Access is from the control panel package downward.

**Communicational.** All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain. You should strive to achieve these levels of cohesion whenever possible. It is important to note, however, that pragmatic design and implementation issues sometimes force you to opt for lower levels of cohesion.

<sup>4</sup> In general, the higher the level of cohesion, the easier the component is to implement, test, and maintain.)

## SAFEHOME



### Cohesion in Action

**The scene:** Jamie's cubicle.

**The players:** Jamie and Ed—members of the SafeHome software engineering team who are working on the surveillance function.

#### The conversation:

**Ed:** I have a first-cut design of the **camera** component.

**Jamie:** Wanna do a quick review?

**Ed:** I guess . . . but really, I'd like your input on something.

(Jamie gestures for him to continue.)

**Ed:** We originally defined five operations for **camera**. Look . . .

*determineType()* tells me the type of camera.

*translateLocation()* allows me to move the camera around the floor plan.

*displayID()* gets the camera ID and displays it near the camera icon.

*displayView()* shows me the field of view of the camera graphically.

*displayZoom()* shows me the magnification of the camera graphically.

**Ed:** I've designed each separately, and they're pretty simple operations. So I thought it might be a good idea to combine all of the display operations into just one that's called *displayCamera()*—it'll show the ID, the view, and the zoom. Whaddaya think?

**Jamie (grimacing):** Not sure that's such a good idea.

**Ed (frowning):** Why, all of these little ops can cause headaches.

**Jamie:** The problem with combining them is we lose cohesion, you know, the *displayCamera()* op won't be single-minded.

**Ed (mildly exasperated):** So what? The whole thing will be less than 100 source lines, max. It'll be easier to implement, I think.

**Jamie:** And what if marketing decides to change the way that we represent the view field?

**Ed:** I just jump into the *displayCamera()* op and make the mod.

**Jamie:** What about side effects?

**Ed:** Whaddaya mean?

**Jamie:** Well, say you make the change but inadvertently create a problem with the ID display.

**Ed:** I wouldn't be that sloppy.

**Jamie:** Maybe not, but what if some support person two years from now has to make the mod. He might not understand the op as well as you do, and, who knows, he might be sloppy.

**Ed:** So you're against it?

**Jamie:** You're the designer . . . it's your decision . . . just be sure you understand the consequences of low cohesion.

**Ed (thinking a moment):** Maybe we'll go with separate display ops.

**Jamie:** Good decision.

### 10.2.4 Coupling

In earlier discussions of analysis and design, I noted that communication and collaboration are essential elements of any object-oriented system. There is, however, a darker side to this important (and necessary) characteristic. As the amount of communication and collaboration increases (i.e., as the degree of “connectedness” between classes increases), the complexity of the system also increases. And as complexity increases, the difficulty of implementing, testing, and maintaining software grows.

*Coupling* is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep coupling as low as is possible.



*As the design for each software component is elaborated, the focus shifts to the design of specific data structures and procedural design to manipulate the data structures. However, don't forget the architecture that must house the components or the global data structures that may serve many components.*

Class coupling can manifest itself in a variety of ways. Lethbridge and Lagani  re [Let01] define the following coupling categories:

**Content coupling.** Occurs when one component “surreptitiously modifies data that is internal to another component” [Let01]. This violates information hiding—a basic design concept.

**Common coupling.** Occurs when a number of components all make use of a global variable. Although this is sometimes necessary (e.g., for establishing default values that are applicable throughout an application), common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.

**Control coupling.** Occurs when operation *A()* invokes operation *B()* and passes a control flag to *B*. The control flag then “directs” logical flow within *B*. The problem with this form of coupling is that an unrelated change in *B* can result in the necessity to change the meaning of the control flag that *A* passes. If this is overlooked, an error will result.

**Stamp coupling.** Occurs when **ClassB** is declared as a type for an argument of an operation of **ClassA**. Because **ClassB** is now a part of the definition of **ClassA**, modifying the system becomes more complex.

**Data coupling.** Occurs when operations pass long strings of data arguments. The “bandwidth” of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.

**Routine call coupling.** Occurs when one operation invokes another. This level of coupling is common and is often quite necessary. However, it does increase the connectedness of a system.

**Type use coupling.** Occurs when component **A** uses a data type defined in component **B** (e.g., this occurs whenever “a class declares an instance variable or a local variable as having another class for its type” [Let01]). If the type definition changes, every component that uses the definition must also change.

**Inclusion or import coupling.** Occurs when component **A** imports or includes a package or the content of component **B**.

**External coupling.** Occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

Software must communicate internally and externally. Therefore, coupling is a fact of life. However, the designer should work to reduce coupling whenever possible and understand the ramifications of high coupling when it cannot be avoided.

## SAFEHOME



### Coupling in Action

**The scene:** Shakira's cubicle.  
**The players:** Vinod and Shakira—members of the SafeHome software team who are working on the security function.

#### The conversation:

**Shakira:** I had what I thought was a great idea . . . then I thought about it a little, and it seemed like a not so great idea. I finally rejected it, but I just thought I'd run it by you.

**Vinod:** Sure. What's the idea?

**Shakira:** Well, each of the sensors recognizes an alarm condition of some kind, right?

**Vinod (smiling):** That's why we call them sensors, Shakira.

**Shakira (exasperated):** Sarcasm, Vinod, you've got to work on your interpersonal skills.

**Vinod:** You were saying?

**Shakira:** Okay, anyway, I figured . . . why not create an operation within each sensor object called *makeCall()* that

would collaborate directly with the **OutgoingCall** component, well, with an interface to the **OutgoingCall** component.

**Vinod (pensive):** You mean rather than having that collaboration occur out of a component like **ControlPanel** or something?

**Shakira:** Yeah . . . but then I said to myself, that means that every sensor object will be connected to the **OutgoingCall** component, and that means that its indirectly coupled to the outside world and . . . well, I just thought it made things complicated.

**Vinod:** I agree. In this case, it's a better idea to let the sensor interface pass info to the **ControlPanel** and let it initiate the outgoing call. Besides, different sensors might result in different phone numbers. You don't want the sensor to store that information because if it changes . . .

**Shakira:** It just didn't feel right.

**Vinod:** Design heuristics for coupling tell us it's not right.

**Shakira:** Whatever . . .

## 10.3 CONDUCTING COMPONENT-LEVEL DESIGN

### QUOTE

"If I had more time, I would have written a shorter letter."

Blaise Pascal

Earlier in this chapter I noted that component-level design is elaborative in nature. You must transform information from requirements and architectural models into a design representation that provides sufficient detail to guide the construction (coding and testing) activity. The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

**Step 1. Identify all design classes that correspond to the problem domain.** Using the requirements and architectural model, each analysis class and architectural component is elaborated as described in Section 10.1.1.

**Step 2. Identify all design classes that correspond to the infrastructure domain.** These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point. As we have noted earlier, classes and components in this category include GUI components (often available as reusable components), operating system components, and object and data management components.

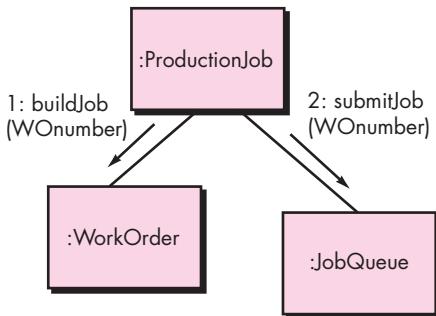
**Step 3. Elaborate all design classes that are not acquired as reusable components.** Elaboration requires that all interfaces, attributes, and operations

### ADVICE

If you're working in a non-OO environment, the first three steps focus on refinement of data objects and processing functions (transforms) identified as part of the requirements model.

**FIGURE 10.6**

Collaboration diagram with messaging



necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

### **Step 3a. Specify message details when classes or components collaborate.**

The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. As component-level design proceeds, it is sometimes useful to show the details of these collaborations by specifying the structure of messages that are passed between objects within a system. Although this design activity is optional, it can be used as a precursor to the specification of interfaces that show how components within the system communicate and collaborate.

Figure 10.6 illustrates a simple collaboration diagram for the printing system discussed earlier. Three objects, **ProductionJob**, **WorkOrder**, and **JobQueue**, collaborate to prepare a print job for submission to the production stream. Messages are passed between objects as illustrated by the arrows in the figure. During requirements modeling the messages are specified as shown in the figure. However, as design proceeds, each message is elaborated by expanding its syntax in the following manner [Ben02]:

```
[guard condition] sequence expression (return value) :=
  message name (argument list)
```

where a **[guard condition]** is written in Object Constraint Language (OCL)<sup>5</sup> and specifies any set of conditions that must be met before the message can be sent; **sequence expression** is an integer value (or other ordering indicator, e.g., 3.1.2) that indicates the sequential order in which a message is sent; **(return value)** is the name of the information that is returned by the operation invoked by the message; **message name** identifies the operation that is to be invoked, and **(argument list)** is the list of attributes that are passed to the operation.

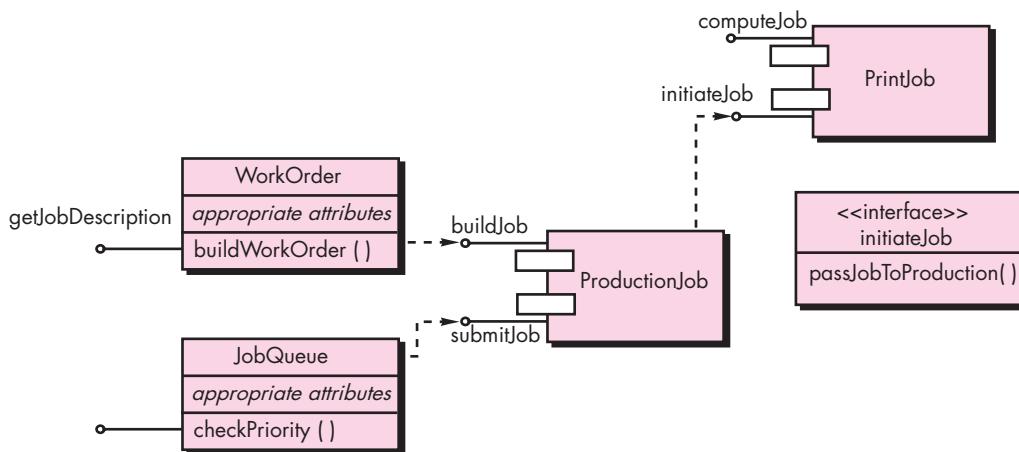
5 OCL is discussed briefly in Appendix 1.

**Step 3b. Identify appropriate interfaces for each component.** Within the context of component-level design, a UML interface is “a group of externally visible (i.e., public) operations. The interface contains no internal structure, it has no attributes, no associations . . .” [Ben02]. Stated more formally, an interface is the equivalent of an abstract class that provides a controlled connection between design classes. The elaboration of interfaces is illustrated in Figure 10.1. In essence, operations defined for the design class are categorized into one or more abstract classes. Every operation within the abstract class (the interface) should be cohesive; that is, it should exhibit processing that focuses on one limited function or subfunction.

Referring to Figure 10.1, it can be argued that the interface *initiateJob* does not exhibit sufficient cohesion. In actuality, it performs three different subfunctions—building a work order, checking job priority, and passing a job to production. The interface design should be refactored. One approach might be to reexamine the design classes and define a new class **WorkOrder** that would take care of all activities associated with the assembly of a work order. The operation *buildWorkOrder()* becomes a part of that class. Similarly, we might define a class **JobQueue** that would incorporate the operation *checkPriority()*. A class **ProductionJob** would encompass all information associated with a production job to be passed to the production facility. The interface *initiateJob* would then take the form shown in Figure 10.7. The interface *initiateJob* is now cohesive, focusing on one function. The interfaces associated with **ProductionJob**, **WorkOrder**, and **JobQueue** are similarly single-minded.

**Step 3c. Elaborate attributes and define data types and data structures required to implement them.** In general, data structures and types used to define attributes are defined within the context of the programming language that is to be

**FIGURE 10.7** Refactoring interfaces and class definitions for PrintJob



used for implementation. UML defines an attribute's data type using the following syntax:

```
name : type-expression = initial-value {property string}
```

where **name** is the attribute name, **type expression** is the data type, **initial value** is the value that the attribute takes when an object is created, and **property-string** defines a property or characteristic of the attribute.

During the first component-level design iteration, attributes are normally described by name. Referring once again to Figure 10.1, the attribute list for **PrintJob** lists only the names of the attributes. However, as design elaboration proceeds, each attribute is defined using the UML attribute format noted. For example, **paperType-weight** is defined in the following manner:

```
paperType-weight: string = "A" { contains 1 of 4 values - A, B, C, or D}
```

which defines **paperType-weight** as a string variable initialized to the value A that can take on one of four values from the set {A,B,C, D}.

If an attribute appears repeatedly across a number of design classes, and it has a relatively complex structure, it is best to create a separate class to accommodate the attribute.

**Step 3d. Describe processing flow within each operation in detail.** This may be accomplished using a programming language-based pseudocode or with a UML activity diagram. Each software component is elaborated through a number of iterations that apply the stepwise refinement concept (Chapter 8).

The first iteration defines each operation as part of the design class. In every case, the operation should be characterized in a way that ensures high cohesion; that is, the operation should perform a single targeted function or subfunction. The next iteration does little more than expand the operation name. For example, the operation *computePaperCost()* noted in Figure 10.1 can be expanded in the following manner:

```
computePaperCost (weight, size, color): numeric
```

This indicates that *computePaperCost()* requires the attributes **weight**, **size**, and **color** as input and returns a value that is numeric (actually a dollar value) as output.

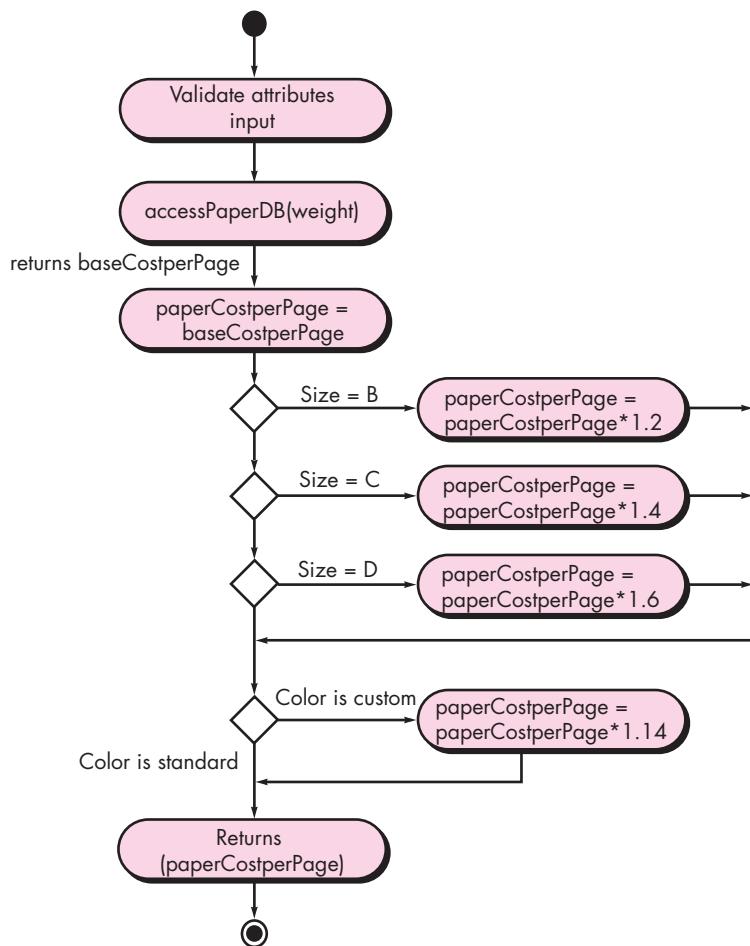


*Use stepwise elaboration as you refine the component design. Always ask, "Is there a way this can be simplified and yet still accomplish the same result?"*

If the algorithm required to implement *computePaperCost()* is simple and widely understood, no further design elaboration may be necessary. The software engineer who does the coding will provide the detail necessary to implement the operation. However, if the algorithm is more complex or arcane, further design elaboration is required at this stage. Figure 10.8 depicts a UML activity diagram for *computePaperCost()*. When activity diagrams are used for component-level design specification, they are generally represented at a level of abstraction that is somewhat higher than source code. An alternative approach—the use of pseudocode for design specification—is discussed in Section 10.5.3.

**FIGURE 10.8**

UML activity diagram for `computePaperCost()`



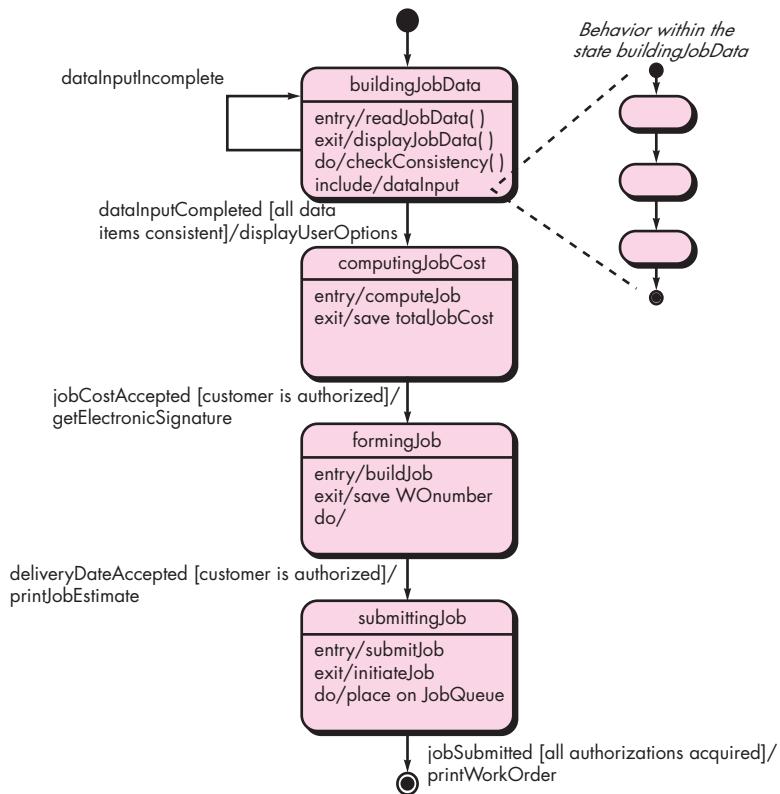
**Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.** Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

**Step 5. Develop and elaborate behavioral representations for a class or component.** UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes. During component-level design, it is sometimes necessary to model the behavior of a design class.

The dynamic behavior of an object (an instantiation of a design class as the program executes) is affected by events that are external to it and the current state

**FIGURE 10.9**

Statechart  
fragment for  
PrintJob class



(mode of behavior) of the object. To understand the dynamic behavior of an object, you should examine all use cases that are relevant to the design class throughout its life. These use cases provide information that helps you to delineate the events that affect the object and the states in which the object resides as time passes and events occur. The transitions between states (driven by events) are represented using a UML statechart [Ben02] as illustrated in Figure 10.9.

The transition from one state (represented by a rectangle with rounded corners) to another occurs as a consequence of an event that takes the form:

**Event-name (parameter-list) [guard-condition] / action expression**

where **event-name** identifies the event, **parameter-list** incorporates data that are associated with the event, **guard-condition** is written in Object Constraint Language (OCL) and specifies a condition that must be met before the event can occur, and **action expression** defines an action that occurs as the transition takes place.

Referring to Figure 10.9, each state may define *entry/* and *exit/* actions that occur as transition into the state occurs and as transition out of the state occurs, respectively. In most cases, these actions correspond to operations that are relevant to the class that is being modeled. The *do/* indicator provides a mechanism for indicating activities that

occur while in the state, and the *include/* indicator provides a means for elaborating the behavior by embedding more statechart detail within the definition of a state.

It is important to note that the behavioral model often contains information that is not immediately obvious in other design models. For example, careful examination of the statechart in Figure 10.9 indicates that the dynamic behavior of the **PrintJob** class is contingent upon two customer approvals as costs and schedule data for the print job are derived. Without approvals (the guard condition ensures that the customer is authorized to approve) the print job cannot be submitted because there is no way to reach the *submittingJob* state.

**Step 6. Elaborate deployment diagrams to provide additional implementation detail.** Deployment diagrams (Chapter 8) are used as part of architectural design and are represented in descriptor form. In this form, major system functions (often represented as subsystems) are represented within the context of the computing environment that will house them.

During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components. However, components generally are not represented individually within a component diagram. The reason for this is to avoid diagrammatic complexity. In some cases, deployment diagrams are elaborated into instance form at this time. This means that the specific hardware and operating system environment(s) that will be used is (are) specified and the location of component packages within this environment is indicated.

**Step 7. Refactor every component-level design representation and always consider alternatives.** Throughout this book, I have emphasized that design is an iterative process. The first component-level model you create will not be as complete, consistent, or accurate as the *n*th iteration you apply to the model. It is essential to refactor as design work is conducted.

In addition, you should not suffer from tunnel vision. There are always alternative design solutions, and the best designers consider all (or most) of them before settling on the final design model. Develop alternatives and consider each carefully, using the design principles and concepts presented in Chapter 8 and in this chapter.

## 10.4 COMPONENT-LEVEL DESIGN FOR WEBAPPS

The boundary between content and function is often blurred when Web-based systems and applications (WebApps) are considered. Therefore, it is reasonable to ask: What is a WebApp component?

In the context of this chapter, a WebApp component is (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end user or (2) a cohesive package of content and functionality that provides the

end user with some required capability. Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

#### 10.4.1 Content Design at the Component Level

Content design at the component level focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end user. As an example, consider a Web-based video surveillance capability within **SafeHomeAssured.com**. Among many capabilities, the user can select and control any of the cameras represented as part of a floor plan, require video-capture thumbnail images from all the cameras, and display streaming video from any one camera. In addition, the user can control pan and zoom for a camera using appropriate control icons.

A number of potential content components can be defined for the video surveillance capability: (1) the content objects that represent the space layout (the floor plan) with additional icons representing the location of sensors and video cameras, (2) the collection of thumbnail video captures (each a separate data object), and (3) the streaming video window for a specific camera. Each of these components can be separately named and manipulated as a package.

Consider a floor plan that depicts four cameras placed strategically throughout a house. Upon user request, a video frame is captured from each camera and is identified as a dynamically generated content object, **VideoCaptureN**, where *N* identifies cameras 1 to 4. A content component, named **Thumbnail-Images**, combines all four **VideoCaptureN** content objects and displays them on the video surveillance page.

The formality of content design at the component level should be tuned to the characteristics of the WebApp to be built. In many cases, content objects need not be organized as components and can be manipulated individually. However, as the size and complexity (of the WebApp, content objects, and their interrelationships) grows, it may be necessary to organize content in a way that allows easier reference and design manipulation.<sup>6</sup> In addition, if content is highly dynamic (e.g., the content for an online auction site), it becomes important to establish a clear structural model that incorporates content components.

#### 10.4.2 Functional Design at the Component Level

Modern Web applications deliver increasingly sophisticated processing functions that (1) perform localized processing to generate content and navigation capability in a dynamic fashion, (2) provide computation or data processing capability that is appropriate for the WebApp's business domain, (3) provide sophisticated database query and access, or (4) establish data interfaces with external corporate systems. To

---

<sup>6</sup> Content components can also be reused in other WebApps.

achieve these (and many other) capabilities, you will design and construct WebApp functional components that are similar in form to software components for conventional software.

WebApp functionality is delivered as a series of components developed in parallel with the information architecture to ensure that they are consistent. In essence you begin by considering both the requirements model and the initial information architecture and then examining how functionality affects the user's interaction with the application, the information that is presented, and the user tasks that are conducted.

During architectural design, WebApp content and functionality are combined to create a functional architecture. A *functional architecture* is a representation of the functional domain of the WebApp and describes the key functional components in the WebApp and how these components interact with each other.

For example, the pan and zoom functions for the **SafeHomeAssured.com** video surveillance capability are implemented as part of a **CameraControl** component. Alternatively, pan and zoom can be implemented as the operations, *pan()* and *zoom()*, which are part of a **Camera** class. In either case, the functionality implied by pan and zoom must be implemented as modules within **SafeHomeAssured.com**.

## 10.5 DESIGNING TRADITIONAL COMPONENTS



Structured programming is a design technique that constrains logic flow to three constructs: sequence, condition, and repetition.

The foundations of component-level design for traditional software components<sup>7</sup> were formed in the early 1960s and were solidified with the work of Edsger Dijkstra and his colleagues ([Boh66], [Dij65], [Dij76b]). In the late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed. The constructs emphasized "maintenance of functional domain." That is, each construct had a predictable logical structure and was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.

The constructs are sequence, condition, and repetition. *Sequence* implements processing steps that are essential in the specification of any algorithm. *Condition* provides the facility for selected processing based on some logical occurrence, and *repetition* allows for looping. These three constructs are fundamental to *structured programming*—an important component-level design technique.

The structured constructs were proposed to limit the procedural design of software to a small number of predictable logical structures. Complexity metrics (Chapter 23) indicate that the use of the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability. The use of

<sup>7</sup> A traditional software component implements an element of processing that addresses a function or subfunction in the problem domain or some capability in the infrastructure domain. Often called modules, procedures, or subroutines, traditional components do not encapsulate data in the same way that object-oriented components do.

a limited number of logical constructs also contributes to a human understanding process that psychologists call *chunking*. To understand this process, consider the way in which you are reading this page. You do not read individual letters but rather recognize patterns or chunks of letters that form words or phrases. The structured constructs are logical chunks that allow a reader to recognize procedural elements of a module, rather than reading the design or code line by line. Understanding is enhanced when readily recognizable logical patterns are encountered.

Any program, regardless of application area or technical complexity, can be designed and implemented using only the three structured constructs. It should be noted, however, that dogmatic use of only these constructs can sometimes cause practical difficulties. Section 10.5.1 considers this issue in further detail.

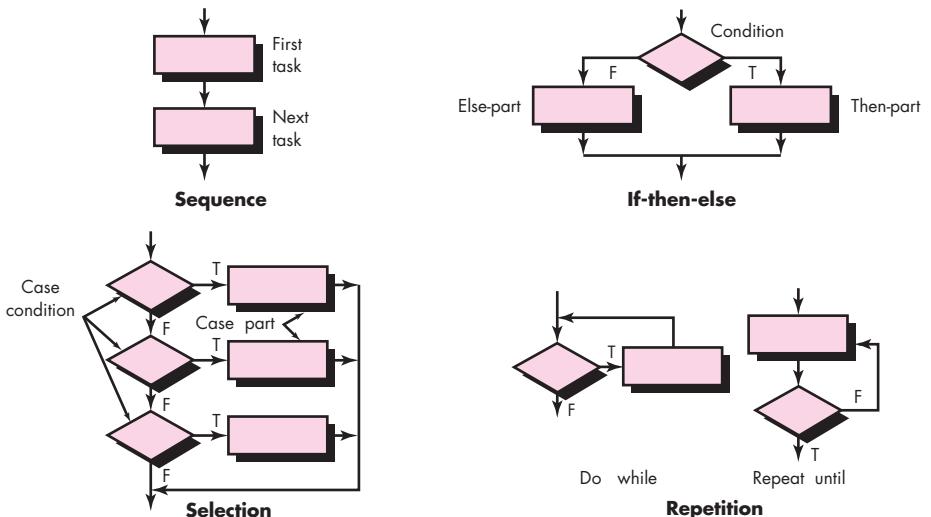
### 10.5.1 Graphical Design Notation

"A picture is worth a thousand words," but it's rather important to know which picture and which 1000 words. There is no question that graphical tools, such as the UML activity diagram or the flowchart, provide useful pictorial patterns that readily depict procedural detail. However, if graphical tools are misused, the wrong picture may lead to the wrong software.

The activity diagram allows you to represent sequence, condition, and repetition—all elements of structured programming—and is a descendent of an earlier pictorial design representation (still used widely) called a *flowchart*. A flowchart, like an activity diagram, is quite simple pictorially. A box is used to indicate a processing step. A diamond represents a logical condition, and arrows show the flow of control. Figure 10.10 illustrates three structured constructs. The *sequence* is

**FIGURE 10.10**

Flowchart constructs



represented as two processing boxes connected by a line (arrow) of control. *Condition*, also called *if-then-else*, is depicted as a decision diamond that, if true, causes *then-part* processing to occur, and if false, invokes *else-part* processing. *Repetition* is represented using two slightly different forms. The *do while* tests a condition and executes a loop task repetitively as long as the condition holds true. A *repeat until* executes the loop task first and then tests a condition and repeats the task until the condition fails. The *selection* (or *select-case*) construct shown in the figure is actually an extension of the *if-then-else*. A parameter is tested by successive decisions until a true condition occurs and a *case part* processing path is executed.

In general, the dogmatic use of only the structured constructs can introduce inefficiency when an escape from a set of nested loops or nested conditions is required. More important, additional complication of all logical tests along the path of escape can cloud software control flow, increase the possibility of error, and have a negative impact on readability and maintainability. What can you do?

You're left with two options: (1) The procedural representation is redesigned so that the "escape branch" is not required at a nested location in the flow of control or (2) the structured constructs are violated in a controlled manner; that is, a constrained branch out of the nested flow is designed. Option 1 is obviously the ideal approach, but option 2 can be accommodated without violating the spirit of structured programming.

### 10.5.2 Tabular Design Notation



*Use a decision table when a complex set of conditions and actions are encountered within a component.*

In many software applications, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions. *Decision tables* [Hur83] provide a notation that translates actions and conditions (described in a processing narrative or a use case) into a tabular form. The table is difficult to misinterpret and may even be used as a machine-readable input to a table-driven algorithm.

Decision table organization is illustrated in Figure 10.11. Referring to the figure, the table is divided into four sections. The upper left-hand quadrant contains a list of all conditions. The lower left-hand quadrant contains a list of all actions that are possible based on combinations of conditions. The right-hand quadrants form a matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination. Therefore, each column of the matrix may be interpreted as a *processing rule*. The following steps are applied to develop a decision table:



1. List all actions that can be associated with a specific procedure (or component).
2. List all conditions (or decisions made) during execution of the procedure.
3. Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
4. Define rules by indicating what actions occur for a set of conditions.

**FIGURE 10.11**

Decision table  
nomenclature

Conditions	1	2	3	4	5	6	Rules
Regular customer	T	T					
Silver customer			T	T			
Gold customer					T	T	
Special discount	F	T	F	T	F	T	
Actions							
No discount	✓						
Apply 8 percent discount			✓	✓			
Apply 15 percent discount					✓	✓	
Apply additional x percent discount	✓		✓			✓	

To illustrate the use of a decision table, consider the following excerpt from an informal use case that has just been proposed for the print shop system:

Three types of customers are defined: a regular customer, a silver customer, and a gold customer (these types are assigned by the amount of business the customer does with the print shop over a 12 month period). A regular customer receives normal print rates and delivery. A silver customer gets an 8 percent discount on all quotes and is placed ahead of all regular customers in the job queue. A gold customer gets a 15 percent reduction in quoted prices and is placed ahead of both regular and silver customers in the job queue. A special discount of  $x$  percent in addition to other discounts can be applied to any customer's quote at the discretion of management.

Figure 10.11 illustrates a decision table representation of the preceding informal use case. Each of the six rules indicates one of six viable conditions. As a general rule, the decision table can be used effectively to supplement other procedural design notation.

### 10.5.3 Program Design Language

*Program design language* (PDL), also called *structured English* or *pseudocode*, incorporates the logical structure of a programming language with the free-form expressive ability of a natural language (e.g., English). Narrative text (e.g., English) is embedded within a programming language-like syntax. Automated tools (e.g., [Cai03]) can be used to enhance the application of PDL.

A basic PDL syntax should include constructs for component definition, interface description, data declaration, block structuring, condition constructs, repetition constructs, and input-output (I/O) constructs. It should be noted that PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt handling, interprocess synchronization, and many other features. The application design for which PDL is to be used should dictate the final form for the design language. The format and semantics for some of these PDL constructs are presented in the example that follows.

To illustrate the use of PDL, consider a procedural design for the *SafeHome* security function discussed in earlier chapters. The system monitors alarms for fire, smoke, burglar, water, and temperature (e.g., the heating system fails while the homeowner is away during winter) and produces an alarm bell and calls a monitoring service, generating a voice-synthesized message.

Recall that PDL is *not* a programming language. You can adapt as required without worry about syntax errors. However, the design for the monitoring software would have to be reviewed (do you see any problems?) and further refined before code could be written. The following PDL<sup>8</sup> provides an elaboration of the procedural design for an early version of an alarm management component.

```

component alarmManagement;
  The intent of this component is to manage control panel switches and input from sensors by
  type and to act on any alarm condition that is encountered.
    set default values for systemStatus (returned value), all data items
    initialize all system ports and reset all hardware
    check controlPanelSwitches (cps)
      if cps = "test" then invoke alarm set to "on"
      if cps = "alarmOff" then invoke alarm set to "off"
      if cps = "newBoundingValue" then invoke keyboardInput
      if cps = "burglarAlarmOff" invoke deactivateAlarm;
      •
      •
      •
      default for cps = none
    reset all signalValues and switches
    do for all sensors
      invoke checkSensor procedure returning signalValue
      if signalValue > bound [alarmType]
        then phoneMessage = message [alarmType]
          set alarmBell to "on" for alarmTimeSeconds
          set system status = "alarmCondition"
          parbegin
            invoke alarm procedure with "on", alarmTimeSeconds;
            invoke phone procedure set to alarmType, phoneNumber
          endpar
        else skip
      endif
    enddofor
  end alarmManagement

```

---

<sup>8</sup> The level of detail represented by the PDL is defined locally. Some people prefer a more natural language-oriented description, while others prefer something that is close to code.

Note that the designer for the **alarmManagement** component has used the construct **parbegin ... parend** that specifies a parallel block. All tasks specified within the **parbegin** block are executed in parallel. In this case, implementation details are not considered.

## 10.6 COMPONENT-BASED DEVELOPMENT

In the software engineering context, reuse is an idea both old and new. Programmers have reused ideas, abstractions, and processes since the earliest days of computing, but the early approach to reuse was ad hoc. Today, complex, high-quality computer-based systems must be built in very short time periods and demand a more organized approach to reuse.

*Component-based software engineering* (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable software “components.” Clements [Cle95] describes CBSE in the following way:

[CBSE] embodies the “buy, don’t build” philosophy espoused by Fred Brooks and others. In the same way that early subroutines liberated the programmer from thinking about details, [CBSE] shifts the emphasis from programming software to composing software systems. Implementation has given way to integration as the focus.

But a number of questions arise. Is it possible to construct complex systems by assembling them from a catalog of reusable software components? Can this be accomplished in a cost- and time-effective manner? Can appropriate incentives be established to encourage software engineers to reuse rather than reinvent? Is management willing to incur the added expense associated with creating reusable software components? Can the library of components necessary to accomplish reuse be created in a way that makes it accessible to those who need it? Can components that do exist be found by those who need them?

Increasingly, the answer to each of these questions is “yes.” In the rest of this section, I examine some of the issues that must be considered to make CBSE successful within a software engineering organization.

### 10.6.1 Domain Engineering

The intent of *domain engineering* is to identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain.<sup>9</sup> The overall goal is to establish mechanisms that enable software engineers to share these components—to reuse them—during work on new and existing systems. Domain engineering includes three major activities—analysis, construction, and dissemination.

**“Domain engineering is about finding commonalities among systems to identify components that can be applied to many systems . . .”**

**Paul Clements**

<sup>9</sup> In Chapter 9 we referred to architectural genres that identify specific application domains.



The analysis process we discuss in this section focuses on reusable components. However, the analysis of complete COTS systems (e.g., e-commerce Apps, sales force automation Apps) can also be a part of domain analysis.

The overall approach to *domain analysis* is often characterized within the context of object-oriented software engineering. The steps in the process are defined as:

1. Define the domain to be investigated.
2. Categorize the items extracted from the domain.
3. Collect a representative sample of applications in the domain.
4. Analyze each application in the sample and define analysis classes.
5. Develop a requirements model for the classes.

It is important to note that domain analysis is applicable to any software engineering paradigm and may be applied for conventional as well as object-oriented development.

### 10.6.2 Component Qualification, Adaptation, and Composition

Domain engineering provides the library of reusable components that are required for component-based software engineering. Some of these reusable components are developed in-house, others can be extracted from existing applications, and still others may be acquired from third parties.

Unfortunately, the existence of reusable components does not guarantee that these components can be integrated easily or effectively into the architecture chosen for a new application. It is for this reason that a sequence of component-based development actions is applied when a component is proposed for use.

**Component Qualification.** Component qualification ensures that a candidate component will perform the function required, will properly “fit” into the architectural style (Chapter 9) specified for the system, and will exhibit the quality characteristics (e.g., performance, reliability, usability) that are required for the application.

An interface description provides useful information about the operation and use of a software component, but it does not provide all of the information required to determine if a proposed component can, in fact, be reused effectively in a new application. Among the many factors considered during component qualification are [Bro96]:



- Application programming interface (API).
- Development and integration tools required by the component.
- Run-time requirements, including resource usage (e.g., memory or storage), timing or speed, and network protocol.
- Service requirements, including operating system interfaces and support from other components.
- Security features, including access controls and authentication protocol.
- Embedded design assumptions, including the use of specific numerical or nonnumerical algorithms.
- Exception handling.

Each of these factors is relatively easy to assess when reusable components that have been developed in-house are proposed. If good software engineering practices were applied during the development of a component, answers to the questions implied by the list can be developed. However, it is much more difficult to determine the internal workings of commercial off-the-shelf (COTS) or third-party components because the only available information may be the interface specification itself.

**Component Adaptation.** In an ideal setting, domain engineering creates a library of components that can be easily integrated into an application architecture. The implication of “easy integration” is that (1) consistent methods of resource management have been implemented for all components in the library, (2) common activities such as data management exist for all components, and (3) interfaces within the architecture and with the external environment have been implemented in a consistent manner.



*In addition to assessing whether the cost of adaptation for reuse is justified, you should also assess whether achieving required functionality and performance can be done cost effectively.*

In reality, even after a component has been qualified for use within an application architecture, conflicts may occur in one or more of the areas just noted. To avoid these conflicts, an adaptation technique called *component wrapping* [Bro96] is sometimes used. When a software team has full access to the internal design and code for a component (often not the case unless open-source COTS components are used), *white-box wrapping* is applied. Like its counterpart in software testing (Chapter 18), white-box wrapping examines the internal processing details of the component and makes code-level modifications to remove any conflict. *Gray-box wrapping* is applied when the component library provides a component extension language or API that enables conflicts to be removed or masked. *Black-box wrapping* requires the introduction of pre- and postprocessing at the component interface to remove or mask conflicts. You must determine whether the effort required to adequately wrap a component is justified or whether a custom component (designed to eliminate the conflicts encountered) should be engineered instead.

**Component Composition.** The component composition task assembles qualified, adapted, and engineered components to populate the architecture established for an application. To accomplish this, an infrastructure must be established to bind the components into an operational system. The infrastructure (usually a library of specialized components) provides a model for the coordination of components and specific services that enable components to coordinate with one another and perform common tasks.

Because the potential impact of reuse and CBSE on the software industry is enormous, a number of major companies and industry consortia have proposed standards for component software.<sup>10</sup>

---

<sup>10</sup> Greg Olsen [Ols06] provides an excellent discussion of past and present industry efforts to make CBSE a reality.

**WebRef**

The latest information on CORBA can be obtained at [www.omg.org](http://www.omg.org).

**WebRef**

The latest information on COM and .NET can be obtained at [www.microsoft.com/COM](http://www.microsoft.com/COM) and [msdn2.microsoft.com/en-us/default.aspx](http://msdn2.microsoft.com/en-us/default.aspx).

**WebRef**

The latest information on JavaBeans can be obtained at [java.sun.com/products/javabeans/docs/](http://java.sun.com/products/javabeans/docs/).

**OMG/CORBA.** The Object Management Group has published a *common object request broker architecture* (OMG/CORBA). An object request broker (ORB) provides a variety of services that enable reusable components (objects) to communicate with other components, regardless of their location within a system.

**Microsoft COM and .NET.** Microsoft has developed a *component object model* (COM) that provides a specification for using components produced by various vendors within a single application running under the Windows operating system. From the point of view of the application, "the focus is not on how [COM objects are] implemented, only on the fact that the object has an interface that it registers with the system, and that it uses the component system to communicate with other COM objects" [Har98a]. The Microsoft .NET framework encompasses COM and provides a reusable class library that covers a wide array of application domains.

**Sun JavaBeans Components.** The JavaBeans component system is a portable, platform-independent CBSE infrastructure developed using the Java programming language. The JavaBeans component system encompasses a set of tools, called the *Bean Development Kit* (BDK), that allows developers to (1) analyze how existing Beans (components) work, (2) customize their behavior and appearance, (3) establish mechanisms for coordination and communication, (4) develop custom Beans for use in a specific application, and (5) test and evaluate Bean behavior.

None of these standards dominate the industry. Although many developers have standardized on one, it is likely that large software organizations may choose to use a standard based on the application categories and platforms that are chosen.

### 10.6.3 Analysis and Design for Reuse

Although the CBSE process encourages the use of existing software components, there are times when new software components must be developed and integrated with existing COTS and in-house components. Because these new components become members of the in-house library of reusable components, they should be engineered for reuse.

Design concepts such as abstraction, hiding, functional independence, refinement, and structured programming, along with object-oriented methods, testing, software quality assurance (SQA), and correctness verification methods (Chapter 21), all contribute to the creation of software components that are reusable. In this subsection, I consider the reuse-specific issues that are complementary to solid software engineering practices.

The requirements model is analyzed to determine those elements that point to existing reusable components. Elements of the requirements model are compared to

descriptions of reusable components in a process that is sometimes referred to as “specification matching” [Bel95]. If specification matching points to an existing component that fits the needs of the current application, you can extract the component from a reuse library (repository) and use it in the design of a new system. If components cannot be found (i.e., there is no match), a new component is created. It is at this point—when you begin to create a new component—that *design for reuse* (DFR) should be considered.

As we have already noted, DFR requires that you apply solid software design concepts and principles (Chapter 8). But the characteristics of the application domain must also be considered. Binder [Bin93] suggests a number of key issues<sup>11</sup> that form a basis for design for reuse:



*DFR can be quite difficult when components must be interfaced or integrated with legacy systems or with multiple systems whose architecture and interfacing protocols are inconsistent.*

**Standard data.** The application domain should be investigated and standard global data structures (e.g., file structures or a complete database) should be identified. All design components can then be characterized to make use of these standard data structures.

**Standard interface protocols.** Three levels of interface protocol should be established: the nature of intramodular interfaces, the design of external technical (nonhuman) interfaces, and the human-computer interface.

**Program templates.** An architectural style (Chapter 9) is chosen and can serve as a template for the architectural design of a new software.

Once standard data, interfaces, and program templates have been established, you have a framework in which to create the design. New components that conform to this framework have a higher probability for subsequent reuse.

#### 10.6.4 Classifying and Retrieving Components

Consider a large university library. Hundreds of thousands of books, periodicals, and other information resources are available for use. But to access these resources, a categorization scheme must be developed. To navigate this large volume of information, librarians have defined a classification scheme that includes a Library of Congress classification code, keywords, author names, and other index entries. All enable the user to find the needed resource quickly and easily.

Now, consider a large component repository. Tens of thousands of reusable software components reside in it. But how do you find the one that you need? To answer this question, another question arises: How do we describe software components in unambiguous, classifiable terms? These are difficult questions, and no definitive answer has yet been developed. In this section I explore current directions that will enable future software engineers to navigate reuse libraries.

---

<sup>11</sup> In general, DFR preparations should be undertaken as part of domain engineering.

A reusable software component can be described in many ways, but an ideal description encompasses what Tracz [Tra95] has called the *3C model*—concept, content, and context. The *concept* of a software component is “a description of what the component does” [Whi95]. The interface to the component is fully described and the semantics—represented within the context of pre- and postconditions—is identified. The concept should communicate the intent of the component. The *content* of a component describes how the concept is realized. In essence, the content is information that is hidden from casual users and need be known only to those who intend to modify or test the component. The *context* places a reusable software component within its domain of applicability. That is, by specifying conceptual, operational, and implementation features, the context enables a software engineer to find the appropriate component to meet application requirements.

To be of use in a pragmatic setting, concept, content, and context must be translated into a concrete specification scheme. Dozens of papers and articles have been written about classification schemes for reusable software components (e.g., see [Cec06] for an overview of current trends).

Classification enables you to find and retrieve candidate reusable components, but a reuse environment must exist to integrate these components effectively. A reuse environment exhibits the following characteristics:



- A component database capable of storing software components and the classification information necessary to retrieve them.
- A library management system that provides access to the database.
- A software component retrieval system (e.g., an object request broker) that enables a client application to retrieve components and services from the library server.
- CBSE tools that support the integration of reused components into a new design or implementation.

Each of these functions interact with or is embodied within the confines of a reuse library.

The *reuse library* is one element of a larger software repository (Chapter 22) and provides facilities for the storage of software components and a wide variety of reusable work products (e.g., specifications, designs, patterns, frameworks, code fragments, test cases, user guides). The library encompasses a database and the tools that are necessary to query the database and retrieve components from it. The component classification scheme serves as the basis for library queries.

Queries are often characterized using the context element of the 3C model described earlier in this section. If an initial query results in a voluminous list of candidate components, the query is refined to narrow the list. Concept and content information are then extracted (after candidate components are found) to assist you in selecting the proper component.

### WebRef

A comprehensive collection of resources on CBSE can be found at [www.cbd-hq.com/](http://www.cbd-hq.com/).



### CBSE

**Objective:** To aid in modeling, design, review, and integration of software components as part of a larger system.

**Mechanics:** Tools mechanics vary. In general, CBSE tools assist in one or more of the following capabilities: specification and modeling of the software architecture, browsing and selection of available software components; integration of components.

#### Representative Tools<sup>12</sup>

*ComponentSource* ([www.componentsource.com](http://www.componentsource.com)) provides a wide array of COTS software components (and tools) supported within many different component standards.

### SOFTWARE TOOLS

*Component Manager*, developed by Flashline ([www.flashline.com](http://www.flashline.com)), "is an application that enables, promotes, and measures software component reuse."

*Select Component Factory*, developed by Select Business Solutions ([www.selectbs.com](http://www.selectbs.com)), "is an integrated set of products for software design, design review, service/component management, requirements management and code generation."

*Software Through Pictures-ACD*, distributed by Aonix ([www.aonix.com](http://www.aonix.com)), enables comprehensive modeling using UML for the OMG model driven architecture—an open, vendor-neutral approach for CBSE.

## 10.7 SUMMARY

The component-level design process encompasses a sequence of activities that slowly reduces the level of abstraction with which software is represented. Component-level design ultimately depicts the software at a level of abstraction that is close to code.

Three different views of component-level design may be taken, depending on the nature of the software to be developed. The object-oriented view focuses on the elaboration of design classes that come from both the problem and infrastructure domain. The traditional view refines three different types of components or modules: control modules, problem domain modules, and infrastructure modules. In both cases, basic design principles and concepts that lead to high-quality software are applied. When considered from a process viewpoint, component-level design draws on reusable software components and design patterns that are pivotal elements of component-based software engineering.

A number of important principles and concepts guide the designer as classes are elaborated. Ideas encompassed in the Open-Closed Principle and the Dependency Inversion Principle and concepts such as coupling and cohesion guide the software engineer in building testable, implementable, and maintainable software components. To conduct component-level design in this context, classes are elaborated by specifying messaging details, identifying appropriate interfaces, elaborating attributes and defining data structures to implement them, describing processing flow

<sup>12</sup> Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

within each operation, and representing behavior at a class or component level. In every case, design iteration (refactoring) is an essential activity.

Traditional component-level design requires the representation of data structures, interfaces, and algorithms for a program module in sufficient detail to guide in the generation of programming language source code. To accomplish this, the designer uses one of a number of design notations that represent component-level detail in either graphical, tabular, or text-based formats.

Component-level design for WebApps considers both content and functionality as it is delivered by a Web-based system. Content design at the component level focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end user. Functional design for WebApps focuses on processing functions that manipulate content, perform computations, query and access a database, and establish interfaces with other systems. All component-level design principles and guidelines apply.

Structured programming is a procedural design philosophy that constrains the number and type of logical constructs used to represent algorithmic detail. The intent of structured programming is to assist the designer in defining algorithms that are less complex and therefore easier to read, test, and maintain.

Component-based software engineering identifies, constructs, catalogs, and disseminates a set of software components in a particular application domain. These components are then qualified, adapted, and integrated for use in a new system. Reusable components should be designed within an environment that establishes standard data structures, interface protocols, and program architectures for each application domain.

## PROBLEMS AND POINTS TO PONDER

**10.1.** The term *component* is sometimes a difficult one to define. First provide a generic definition, and then provide more explicit definitions for object-oriented and traditional software. Finally, pick three programming languages with which you are familiar and illustrate how each defines a component.

**10.2.** Why are control components necessary in traditional software and generally not required in object-oriented software?

**10.3.** Describe the OCP in your own words. Why is it important to create abstractions that serve as an interface between components?

**10.4.** Describe the DIP in your own words. What might happen if a designer depends too heavily on concretions?

**10.5.** Select three components that you have developed recently and assess the types of cohesion that each exhibits. If you had to define the primary benefit of high cohesion, what would it be?

**10.6.** Select three components that you have developed recently and assess the types of coupling that each exhibits. If you had to define the primary benefit of low coupling, what would it be?

**10.7.** Is it reasonable to say that problem domain components should never exhibit external coupling? If you agree, what types of component would exhibit external coupling?

**10.8.** Develop (1) an elaborated design class, (2) interface descriptions, (3) an activity diagram for one of the operations within the class, and (4) a detailed statechart diagram for one of the *SafeHome* classes that we have discussed in earlier chapters.

**10.9.** Are stepwise refinement and refactoring the same thing? If not, how do they differ?

**10.10.** What is a WebApp component?

**10.11.** Select a small portion of an existing program (approximately 50 to 75 source lines). Isolate the structured programming constructs by drawing boxes around them in the source code. Does the program excerpt have constructs that violate the structured programming philosophy? If so, redesign the code to make it conform to structured programming constructs. If not, what do you notice about the boxes that you've drawn?

**10.12.** All modern programming languages implement the structured programming constructs. Provide examples from three programming languages.

**10.13.** Select a small coded component and represent it using (1) an activity diagram, (2) a flowchart, (3) a decision table, and (4) PDL.

**10.14.** Why is “chunking” important during the component-level design review process?

## FURTHER READINGS AND INFORMATION SOURCES

Many books on component-based development and component reuse have been published in recent years. Apperly and his colleagues (*Service- and Component-Based Development*, Addison-Wesley, 2003), Heineman and Councill (*Component Based Software Engineering*, Addison-Wesley, 2001), Brown (*Large Scale Component-Based Development*, Prentice-Hall, 2000), Allen (*Realizing e-Business with Components*, Addison-Wesley, 2000), Herzum and Sims (*Business Component Factory*, Wiley, 1999), Allen, Frost, and Yourdon (*Component-Based Development for Enterprise Systems: Applying the Select Perspective*, Cambridge University Press, 1998) cover all important aspects of the CBSE process. Cheesman and Daniels (*UML Components*, Addison-Wesley, 2000) discuss CBSE with a UML emphasis.

Gao and his colleagues (*Testing and Quality Assurance for Component-Based Software*, Artech House, 2006) and Gross (*Component-Based Software Testing with UML*, Springer, 2005) discuss testing and SQA issues for component-based systems.

Dozens of books describing the industry's component-based standards have been published in recent years. These address the inner workings of the standards themselves but also consider many important CBSE topics.

The work of Linger, Mills, and Witt (*Structured Programming—Theory and Practice*, Addison-Wesley, 1979) remains a definitive treatment of the subject. The text contains a good PDL as well as detailed discussions of the ramifications of structured programming. Other books that focus on procedural design issues for traditional systems include those by Robertson (*Simple Program Design*, 3d ed., Course Technology, 2000), Farrell (*A Guide to Programming Logic and Design*, Course Technology, 1999), Bentley (*Programming Pearls*, 2d ed., Addison-Wesley, 1999), and Dahl (*Structured Programming*, Academic Press, 1997).

Relatively few recent books have been dedicated solely to component-level design. In general, programming language books address procedural design in some detail but always in the context of the language that is introduced by the book. Hundreds of titles are available.

A wide variety of information sources on component-level design are available on the Internet. An up-to-date list of World Wide Web references that are relevant to component-level design can be found at the SEPA website: [www.mhhe.com/engcs/compisci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compisci/pressman/professional/olc/ser.htm).

## CHAPTER

# 11

## USER INTERFACE DESIGN

### KEY CONCEPTS

accessibility	...334
command	
labeling	.....333
control	.....313
design	
evaluation	....342
error handling	....333
golden rules	...313
help facilities	..332
interface analysis	....320
consistent	...316
design	.....328
models	....317
internationalization	.....334
memory load	..314

We live in a world of high-technology products, and virtually all of them—consumer electronics, industrial equipment, corporate systems, military systems, personal computer software, and WebApps—require human interaction. If a product is to be successful, it must exhibit good *usability*—a qualitative measure of the ease and efficiency with which a human can employ the functions and features offered by the high-technology product.

Whether an interface has been designed for a digital music player or the weapons control system for a fighter aircraft, usability matters. If interface mechanisms have been well designed, the user glides through the interaction using a smooth rhythm that allows work to be accomplished effortlessly. But if the interface is poorly conceived, the user moves in fits and starts, and the end result is frustration and poor work efficiency.

For the first three decades of the computing era, usability was not a dominant concern among those who built software. In his classic book on design, Donald Norman [Nor88] argued that it was time for a change in attitude:

To make technology that fits human beings, it is necessary to study human beings. But now we tend to study only the technology. As a result, people are required to conform to technology. It is time to reverse this trend, time to make technology that conforms to people.

### QUICK LOOK

**What is it?** User interface design creates an effective communication medium between a human and a computer. Following a set of interface

design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

**Who does it?** A software engineer designs the user interface by applying an iterative process that draws on predefined design principles.

**Why is it important?** If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits, the content it delivers, or the functionality it offers. The interface has to be right because it molds a user's perception of the software.

**What are the steps?** User interface design begins with the identification of user, task, and environ-

mental requirements. Once user tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. These form the basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Tools are used to prototype and ultimately implement the design model, and the result is evaluated for quality.

**What is the work product?** User scenarios are created and screen layouts are generated. An interface prototype is developed and modified in an iterative fashion.

**How do I ensure that I've done it right?** An interface prototype is "test driven" by the users, and feedback from the test drive is used for the next iterative modification of the prototype.

principles and guidelines .....	336
process .....	319
response time .....	332
task analysis .....	322
task elaboration .....	324
usability .....	317
user analysis .....	321
WebApp interface design .....	335

As technologists studied human interaction, two dominant issues arose. First, a set of *golden rules* (discussed in Section 11.1) were identified. These applied to all human interaction with technology products. Second, a set of *interaction mechanisms* were defined to enable software designers to build systems that properly implemented the golden rules. These interaction mechanisms, collectively called the *graphical user interface* (GUI), have eliminated some of the most egregious problems associated with human interfaces. But even in a “Windows world,” we all have encountered user interfaces that are difficult to learn, difficult to use, confusing, counterintuitive, unforgiving, and in many cases, totally frustrating. Yet, someone spent time and energy building each of these interfaces, and it is not likely that the builder created these problems purposely.

## 11.1 THE GOLDEN RULES

In his book on interface design, Theo Mandel [Man97] coins three *golden rules*:

1. Place the user in control.
2. Reduce the user’s memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

### 11.1.1 Place the User in Control

#### quote:

“It’s better to design the user experience than rectify it.”

Jon Meads

During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface.

“What I really would like,” said the user solemnly, “is a system that reads my mind. It knows what I want to do before I need to do it and makes it very easy for me to get it done. That’s all, just that.”

My first reaction was to shake my head and smile, but I paused for a moment. There was absolutely nothing wrong with the user’s request. She wanted a system that reacted to her needs and helped her get things done. She wanted to control the computer, not have the computer control her.

Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction. But for whom?

As a designer, you may be tempted to introduce constraints and limitations to simplify the implementation of the interface. The result may be an interface that is easy to build, but frustrating to use. Mandel [Man97] defines a number of design principles that allow the user to maintain control:

**Define interaction modes in a way that does not force a user into unnecessary or undesired actions.** An interaction mode is the current state of the interface. For example, if *spell check* is selected in a word-processor menu, the software

moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

**Provide for flexible interaction.** Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multitouch screen, or voice recognition commands. But every action is not amenable to every interaction mechanism. Consider, for example, the difficulty of using keyboard command (or voice input) to draw a complex shape.

**Allow user interaction to be interruptible and undoable.** Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to “undo” any action.

**Streamline interaction as skill levels advance and allow the interaction to be customized.** Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a “macro” mechanism that enables an advanced user to customize the interface to facilitate interaction.



### quote:

“I have always wished that my computer would be as easy to use as my telephone. My wish has come true. I no longer know how to use my telephone.”

Bjarne  
Stroustrup  
(originator of  
C++)

**Hide technical internals from the casual user.** The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology. In essence, the interface should never require that the user interact at a level that is “inside” the machine (e.g., a user should never be required to type operating system commands from within application software).

**Design for direct interaction with objects that appear on the screen.** The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to “stretch” an object (scale it in size) is an implementation of direct manipulation.

#### 11.1.2 Reduce the User’s Memory Load

The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user’s memory. Whenever possible, the system should “remember” pertinent information and assist the user with an interaction scenario that assists recall. Mandel [Man97] defines design principles that enable an interface to reduce the user’s memory load:

**Reduce demand on short-term memory.** When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results.

This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

**Establish meaningful defaults.** The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a “reset” option should be available, enabling the redefinition of original default values.

**Define shortcuts that are intuitive.** When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).

**The visual layout of the interface should be based on a real-world metaphor.** For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

**Disclose information in a progressive fashion.** The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick. An example, common to many word-processing applications, is the underlining function. The function itself is one of a number of functions under a *text style* menu. However, every underlining capability is not listed. The user must pick underlining; then all underlining options (e.g., single underline, double underline, dashed underline) are presented.

## SAFEHOME



### Violating a UI Golden Rule

**The scene:** Vinod’s cubicle, as user interface design begins.

**The players:** Vinod and Jamie, members of the *SafeHome* software engineering team.

#### The conversation:

**Jamie:** I’ve been thinking about the surveillance function interface.

**Vinod (smiling):** Thinking is good.

**Jamie:** I think maybe we can simplify matters some.

**Vinod:** Meaning?

**Jamie:** Well, what if we eliminate the floor plan entirely. It’s flashy, but it’s going to take serious development effort. Instead we just ask the user to specify the camera he wants to see and then display the video in a video window.

**Vinod:** How does the homeowner remember how many cameras are set up and where they are?

**Jamie (mildly irritated):** He’s the homeowner; he should know.

**Vinod:** But what if he doesn’t?

**Jamie:** He should.

**Vinod:** That's not the point . . . what if he forgets?

**Jamie:** Uh, we could provide a list of operational cameras and their locations.

**Vinod:** That's possible, but why should he have to ask for a list?

**Jamie:** Okay, we provide the list whether he asks or not.

**Vinod:** Better. At least he doesn't have to remember stuff that we can give him.

**Jamie (thinking for a moment):** But you like the floor plan, don't you?

**Vinod:** Uh huh.

**Jamie:** Which one will marketing like, do you think?

**Vinod:** You're kidding, right?

**Jamie:** No.

**Vinod:** Duh . . . the one with the flash . . . they love sexy product features . . . they're not interested in which is easier to build.

**Jamie (sighing):** Okay, maybe I'll prototype both.

**Vinod:** Good idea . . . then we let the customer decide.

### 11.1.3 Make the Interface Consistent

 **Quote:**

"Things that look different should act different. Things that look the same should act the same."

**Larry Marine**

The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to design rules that are maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined and implemented. Mandel [Man97] defines a set of design principles that help make the interface consistent:

**Allow the user to put the current task into a meaningful context.** Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

**Maintain consistency across a family of applications.** A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

**If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.** Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion.

The interface design principles discussed in this and the preceding sections provide you with basic guidance. In the sections that follow, you'll learn about the interface design process itself.



### Usability

In an insightful paper on usability, Larry Constantine [Con95] asks a question that has significant bearing on the subject: "What do users want, anyway?" He answers this way:

What users really want are good tools. All software systems, from operating systems and languages to data entry and decision support applications, are just tools. End users want from the tools we engineer for them much the same as we expect from the tools we use. They want systems that are easy to learn and that help them do their work. They want software that doesn't slow them down, that doesn't trick or confuse them, that doesn't make it easier to make mistakes or harder to finish the job.

Constantine argues that usability is not derived from aesthetics, state-of-the-art interaction mechanisms, or built-in interface intelligence. Rather, it occurs when the architecture of the interface fits the needs of the people who will be using it.

A formal definition of usability is somewhat illusive. Donahue and his colleagues [Don99] define it in the following manner: "Usability is a measure of how well a computer system . . . facilitates learning; helps learners remember what they've learned; reduces the likelihood of errors; enables them to be efficient, and makes them satisfied with the system."

The only way to determine whether "usability" exists within a system you are building is to conduct usability

### INFO

assessment or testing. Watch users interact with the system and answer the following questions [Con95]:

- Is the system usable without continual help or instruction?
- Do the rules of interaction help a knowledgeable user to work efficiently?
- Do interaction mechanisms become more flexible as users become more knowledgeable?
- Has the system been tuned to the physical and social environment in which it will be used?
- Is the user aware of the state of the system? Does the user know where she is at all times?
- Is the interface structured in a logical and consistent manner?
- Are interaction mechanisms, icons, and procedures consistent across the interface?
- Does the interaction anticipate errors and help the user correct them?
- Is the interface tolerant of errors that are made?
- Is the interaction simple?

If each of these questions is answered "yes," it is likely that usability has been achieved.

Among the many measurable benefits derived from a usable system are [Don99]: increased sales and customer satisfaction, competitive advantage, better reviews in the media, better word of mouth, reduced support costs, improved end-user productivity, reduced training costs, reduced documentation costs, reduced likelihood of litigation from unhappy customers.

## 11.2 USER INTERFACE ANALYSIS AND DESIGN

### WebRef

An excellent source of UI design information can be found at [www.useit.com](http://www.useit.com).

The overall process for analyzing and designing a user interface begins with the creation of different models of system function (as perceived from the outside). You begin by delineating the human- and computer-oriented tasks that are required to achieve system function and then considering the design issues that apply to all interface designs. Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.

### 11.2.1 Interface Analysis and Design Models

Four different models come into play when a user interface is to be analyzed and designed. A human engineer (or the software engineer) establishes a *user model*, the software engineer creates a *design model*, the end user develops a mental image that is often called the user's *mental model* or the *system perception*, and the implementers

**Quote:**

"If there's a 'trick' to it, the UI is broken."

**Douglas Anderson**



Even a novice user wants shortcuts; even knowledgeable, frequent users sometimes need guidance. Give them what they need.



The user's mental model shapes how the user perceives the interface and whether the UI meets the user's needs.

of the system create an *implementation model*. Unfortunately, each of these models may differ significantly. Your role, as an interface designer, is to reconcile these differences and derive a consistent representation of the interface.

The user model establishes the profile of end users of the system. In his introductory column on "user-centric design," Jeff Patton [Pat07] notes:

The truth is, designers and developers—myself included—often think about users. However, in the absence of a strong mental model of specific users, we self-substitute. Self-substitution isn't user centric—it's self-centric.

To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality" [Shn04]. In addition, users can be categorized as:

*Novices.* No syntactic knowledge<sup>1</sup> of the system and little semantic knowledge<sup>2</sup> of the application or computer usage in general.

*Knowledgeable, intermittent users.* Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.

*Knowledgeable, frequent users.* Good semantic and syntactic knowledge that often leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction.

The user's *mental model* (system perception) is the image of the system that end users carry in their heads. For example, if the user of a particular word processor were asked to describe its operation, the system perception would guide the response. The accuracy of the description will depend upon the user's profile (e.g., novices would provide a sketchy response at best) and overall familiarity with software in the application domain. A user who understands word processors fully but has worked with the specific word processor only once might actually be able to provide a more complete description of its function than the novice who has spent weeks trying to learn the system.

The *implementation model* combines the outward manifestation of the computer-based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively. To accomplish this "melding" of the models, the design model must have been

- 
- 1 In this context, *syntactic knowledge* refers to the mechanics of interaction that are required to use the interface effectively.
  - 2 *Semantic knowledge* refers to the underlying sense of the application—an understanding of the functions that are performed, the meaning of input and output, and the goals and objectives of the system.

developed to accommodate the information contained in the user model, and the implementation model must accurately reflect syntactic and semantic information about the interface.

**Q** **vote:**

"... pay attention to what users do, not what they say."

Jakob Nielsen

**Q** **vote:**

"It's better to design the user experience than rectify it."

Jon Meads

The models described in this section are "abstractions of what the user is doing or thinks he is doing or what somebody else thinks he ought to be doing when he uses an interactive system" [Mon84]. In essence, these models enable the interface designer to satisfy a key element of the most important principle of user interface design: "Know the user, know the tasks."

### 11.2.2 The Process

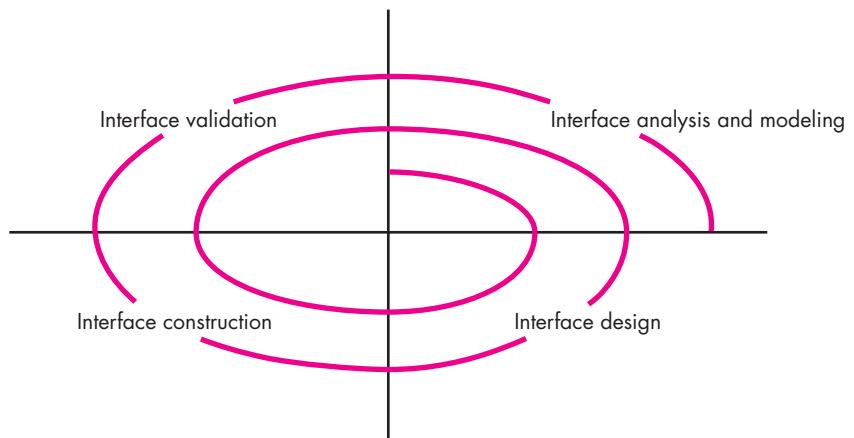
The analysis and design process for user interfaces is iterative and can be represented using a spiral model similar to the one discussed in Chapter 2. Referring to Figure 11.1, the user interface analysis and design process begins at the interior of the spiral and encompasses four distinct framework activities [Man97]: (1) interface analysis and modeling, (2) interface design, (3) interface construction, and (4) interface validation. The spiral shown in Figure 11.1 implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed.

*Interface analysis* focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, you work to understand the system perception (Section 11.2.1) for each class of users.

Once general requirements have been defined, a more detailed *task analysis* is conducted. Those tasks that the user performs to accomplish the goals of the system

**FIGURE 11.1**

The user interface design process



are identified, described, and elaborated (over a number of iterative passes through the spiral). Task analysis is discussed in more detail in Section 11.3. Finally, analysis of the user environment focuses on the physical work environment. Among the questions to be asked are



- Where will the interface be located physically?
- Will the user be sitting, standing, or performing other tasks unrelated to the interface?
- Does the interface hardware accommodate space, light, or noise constraints?
- Are there special human factors considerations driven by environmental factors?

The information gathered as part of the analysis action is used to create an analysis model for the interface. Using this model as a basis, the design action commences.

The goal of *interface design* is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system. Interface design is discussed in more detail in Section 11.4.

*Interface construction* normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit (Section 11.5) may be used to complete the construction of the interface.

*Interface validation* focuses on (1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements; (2) the degree to which the interface is easy to use and easy to learn, and (3) the users' acceptance of the interface as a useful tool in their work.

As I have already noted, the activities described in this section occur iteratively. Therefore, there is no need to attempt to specify every detail (for the analysis or design model) on the first pass. Subsequent passes through the process elaborate task detail, design information, and the operational features of the interface.

### 11.3 INTERFACE ANALYSIS<sup>3</sup>

A key tenet of all software engineering process models is this: *understand the problem before you attempt to design a solution*. In the case of user interface design, understanding the problem means understanding (1) the people (end users) who will interact with the system through the interface, (2) the tasks that end users must

<sup>3</sup> It is reasonable to argue that this section should be placed in Chapter 5, 6, or 7, since requirements analysis issues are discussed there. It has been positioned here because interface analysis and design are intimately connected to one another, and the boundary between the two is often fuzzy.

perform to do their work, (3) the content that is presented as part of the interface, and (4) the environment in which these tasks will be conducted. In the sections that follow, I examine each of these elements of interface analysis with the intent of establishing a solid foundation for the design tasks that follow.

### 11.3.1 User Analysis

The phrase “user interface” is probably all the justification needed to spend some time understanding the user before worrying about technical matters. Earlier I noted that each user has a mental image of the software that may be different from the mental image developed by other users. In addition, the user’s mental image may be vastly different from the software engineer’s design model. The only way that you can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use the system. Information from a broad array of sources can be used to accomplish this:



*Above all, spend time talking to actual users, but be careful. One strong opinion doesn’t necessarily mean that the majority of users will agree.*



How do we learn about the demographics and characteristics of end users?

**User Interviews.** The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups.

**Sales input.** Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.

**Marketing input.** Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.

**Support input.** Support staff talks with users on a daily basis. They are the most likely source of information on what works and what doesn’t, what users like and what they dislike, what features generate questions and what features are easy to use.

The following set of questions (adapted from [Hac98]) will help you to better understand the users of a system:

- Are users trained professionals, technicians, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?

- Do users work normal office hours or do they work until the job is done?
- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology that sits behind the interface?

Once these questions are answered, you'll know who the end users are, what is likely to motivate and please them, how they can be grouped into different user classes or profiles, what their mental models of the system are, and how the user interface must be characterized to meet their needs.

### 11.3.2 Task Analysis and Modeling

The goal of task analysis is to answer the following questions:



The user's goal is to accomplish one or more tasks via the UI. To accomplish this, the UI must provide mechanisms that allow the user to achieve her goal.

- What work will the user perform in specific circumstances?
- What tasks and subtasks will be performed as the user does the work?
- What specific problem domain objects will the user manipulate as work is performed?
- What is the sequence of work tasks—the workflow?
- What is the hierarchy of tasks?

To answer these questions, you must draw upon techniques that I have discussed earlier in this book, but in this instance, these techniques are applied to the user interface.

**Use cases.** In earlier chapters you learned that the use case describes the manner in which an actor (in the context of user interface design, an actor is always a person) interacts with a system. When used as part of task analysis, the use case is developed to show how an end user performs some specific work-related task. In most instances, the use case is written in an informal style (a simple paragraph) in the first-person. For example, assume that a small software company wants to build a computer-aided design system explicitly for interior designers. To get a better understanding of how they do their work, actual interior designers are asked to describe a specific design function. When asked: "How do you decide where to put furniture in a room?" an interior designer writes the following informal use case:

I begin by sketching the floor plan of the room, the dimensions and the location of windows and doors. I'm very concerned about light as it enters the room, about the view out of the windows (if it's beautiful, I want to draw attention to it), about the running length of an unobstructed wall, about the flow of movement through the room. I then look at the list of furniture my customer and I have chosen—tables, chairs, sofa, cabinets, the list of

accents—lamps, rugs, paintings, sculpture, plants, smaller pieces, and my notes on any desires my customer has for placement. I then draw each item from my lists using a template that is scaled to the floor plan. I label each item I draw and use pencil because I always move things. I consider a number of alternative placements and decide on the one I like best. Then, I draw a rendering (a 3-D picture) of the room to give my customer a feel for what it'll look like.

This use case provides a basic description of one important work task for the computer-aided design system. From it, you can extract tasks, objects, and the overall flow of the interaction. In addition, other features of the system that would please the interior designer might also be conceived. For example, a digital photo could be taken looking out each window in a room. When the room is rendered, the actual outside view could be represented through each window.

## SAFEHOME



### Use Cases for UI Design

**The scene:** Vinod's cubicle, as user interface design continues.

**The players:** Vinod and Jamie, members of the *SafeHome* software engineering team.

#### The conversation:

**Jamie:** I pinned down our marketing contact and had her write a use case for the surveillance interface.

**Vinod:** From whose point of view?

**Jamie:** The homeowner, who else is there?

**Vinod:** There's also the system administrator role, even if it's the homeowner playing the role, it's a different point of view. The "administrator" sets the system up, configures stuff, lays out the floor plan, places the cameras . . .

**Jamie:** All I had her do was play the role of the homeowner when he wants to see video.

**Vinod:** That's okay. It's one of the major behaviors of the surveillance function interface. But we're going to have to examine the system administration behavior as well.

**Jamie (irritated):** You're right.

[Jamie leaves to find the marketing person. She returns a few hours later.]

**Jamie:** I was lucky, I found her and we worked through the administrator use case together. Basically, we're going to define "administration" as one function that's applicable to all other *SafeHome* functions. Here's what we came up with.

[Jamie shows the informal use case to Vinod.]

**Informal use case:** I want to be able to set or edit the system layout at any time. When I set up the system, I select an administration function. It asks me whether I want to do a new setup or whether I want to edit an existing setup. If I select a new setup, the system displays a drawing screen that will enable me to draw the floor plan onto a grid. There will be icons for walls, windows, and doors so that drawing is easy. I just stretch the icons to their appropriate lengths. The system will display the lengths in feet or meters (I can select the measurement system). I can select from a library of sensors and cameras and place them on the floor plan. I get to label each, or the system will do automatic labeling. I can establish settings for sensors and cameras from appropriate menus. If I select edit, I can move sensors or cameras, add new ones or delete existing ones, edit the floor plan, and edit the settings for cameras and sensors. In every case, I expect the system to do consistency checking and to help me avoid mistakes.

**Vinod (after reading the scenario):** Okay, there are probably some useful design patterns [Chapter 12] or reusable components for GUIs for drawing programs. I'll betcha 50 bucks we can implement some or most of the administrator interface using them.

**Jamie:** Agreed. I'll check it out.

**Task elaboration.** In Chapter 8, I discussed stepwise elaboration (also called functional decomposition or stepwise refinement) as a mechanism for refining the processing tasks that are required for software to accomplish some desired function. Task analysis for interface design uses an elaborative approach to assist in understanding the human activities the user interface must accommodate.



*Task elaboration is quite useful, but it can also be dangerous. Just because you have elaborated a task, do not assume that there isn't another way to do it, and that the other way will be tried when the UI is implemented.*

Task analysis can be applied in two ways. As I have already noted, an interactive, computer-based system is often used to replace a manual or semimanual activity. To understand the tasks that must be performed to accomplish the goal of the activity, you must understand the tasks that people currently perform (when using a manual approach) and then map these into a similar (but not necessarily identical) set of tasks that are implemented in the context of the user interface. Alternatively, you can study an existing specification for a computer-based solution and derive a set of user tasks that will accommodate the user model, the design model, and the system perception.

Regardless of the overall approach to task analysis, you must first define and classify tasks. I have already noted that one approach is stepwise elaboration. For example, let's reconsider the computer-aided design system for interior designers discussed earlier. By observing an interior designer at work, you notice that interior design comprises a number of major activities: furniture layout (note the use case discussed earlier), fabric and material selection, wall and window coverings selection, presentation (to the customer), costing, and shopping. Each of these major tasks can be elaborated into subtasks. For example, using information contained in the use case, furniture layout can be refined into the following tasks: (1) draw a floor plan based on room dimensions, (2) place windows and doors at appropriate locations, (3a) use furniture templates to draw scaled furniture outlines on the floor plan, (3b) use accents templates to draw scaled accents on the floor plan, (4) move furniture outlines and accent outlines to get the best placement, (5) label all furniture and accent outlines, (6) draw dimensions to show location, and (7) draw a perspective-rendering view for the customer. A similar approach could be used for each of the other major tasks.

Subtasks 1 to 7 can each be refined further. Subtasks 1 to 6 will be performed by manipulating information and performing actions within the user interface. On the other hand, subtask 7 can be performed automatically in software and will result in little direct user interaction.<sup>4</sup> The design model of the interface should accommodate each of these tasks in a way that is consistent with the user model (the profile of a "typical" interior designer) and system perception (what the interior designer expects from an automated system).

**Object elaboration.** Rather than focusing on the tasks that a user must perform, you can examine the use case and other information obtained from the user and extract the physical objects that are used by the interior designer. These objects can

<sup>4</sup> However, this may not be the case. The interior designer might want to specify the perspective to be drawn, the scaling, the use of color, and other information. The use case related to drawing perspective renderings would provide the information you need to address this task.



Although object elaboration is useful, it should not be used as a stand-alone approach. The user's voice must be considered during task analysis.

be categorized into classes. Attributes of each class are defined, and an evaluation of the actions applied to each object provide a list of operations. For example, the furniture template might translate into a class called **Furniture** with attributes that might include **size**, **shape**, **location**, and others. The interior designer would *select* the object from the **Furniture** class, *move* it to a position on the floor plan (another object in this context), *draw* the furniture outline, and so forth. The tasks *select*, *move*, and *draw* are operations. The user interface analysis model would not provide a literal implementation for each of these operations. However, as the design is elaborated, the details of each operation are defined.

**Workflow analysis.** When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply *workflow analysis*. This technique allows you to understand how a work process is completed when several people (and roles) are involved. Consider a company that intends to fully automate the process of prescribing and delivering prescription drugs. The entire process<sup>5</sup> will revolve around a Web-based application that is accessible by physicians (or their assistants), pharmacists, and patients. Workflow can be represented effectively with a UML swimlane diagram (a variation on the activity diagram).

We consider only a small part of the work process: the situation that occurs when a patient asks for a refill. Figure 11.2 presents a swimlane diagram that indicates the tasks and decisions for each of the three roles noted earlier. This information may have been elicited via interview or from use cases written by each actor. Regardless, the flow of events (shown in the figure) enables you to recognize a number of key interface characteristics:

## A pink speech bubble icon containing the word "QUOTE:" in white capital letters.

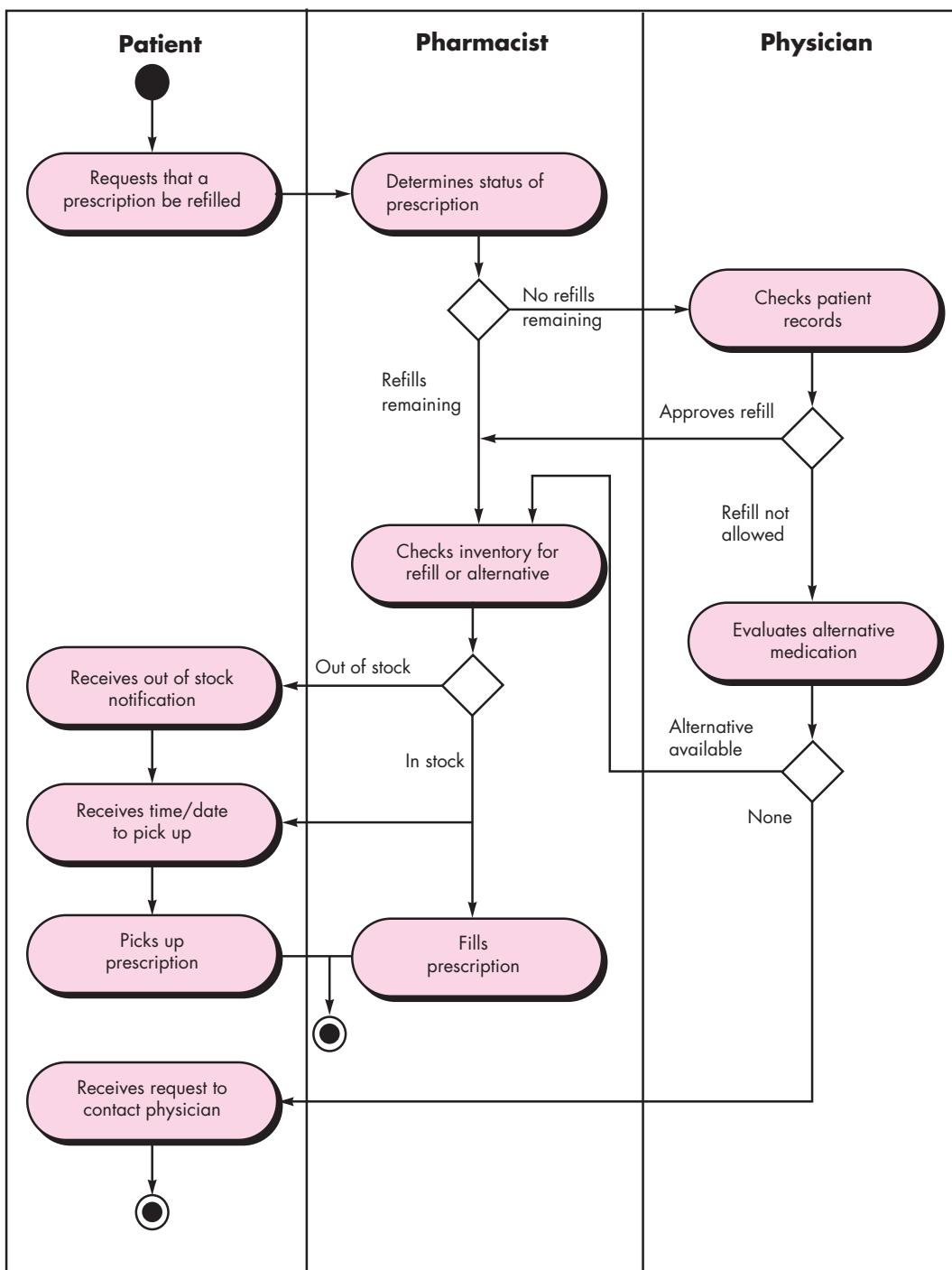
"It is far better to adapt the technology to the user than to force the user to adapt to the technology."

Larry Marine

1. Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different than the one defined for pharmacists or physicians.
2. The interface design for pharmacists and physicians must accommodate access to and display of information from secondary information sources (e.g., access to inventory for the pharmacist and access to information about alternative medications for the physician).
3. Many of the activities noted in the swimlane diagram can be further elaborated using task analysis and/or object elaboration (e.g., *Fills prescription* could imply a mail-order delivery, a visit to a pharmacy, or a visit to a special drug distribution center).

**Hierarchical representation.** A process of elaboration occurs as you begin to analyze the interface. Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of

<sup>5</sup> This example has been adapted from [Hac98].

**FIGURE 11.2** Swimlane diagram for prescription refill function

each task identified for the user. For example, consider the following user task and subtask hierarchy.

**User task: Requests that a prescription be refilled**

- Provide identifying information.
  - Specify name.
  - Specify userid.
  - Specify PIN and password.
- Specify prescription number.
- Specify date refill is required.

To complete the task, three subtasks are defined. One of these subtasks, *provide identifying information*, is further elaborated in three additional sub-subtasks.

### 11.3.3 Analysis of Display Content

The user tasks identified in Section 11.3.2 lead to the presentation of a variety of different types of content. For modern applications, display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or video files). The analysis modeling techniques discussed in Chapters 6 and 7 identify the output data objects that are produced by an application. These data objects may be (1) generated by components (unrelated to the interface) in other parts of an application, (2) acquired from data stored in a database that is accessible from the application, or (3) transmitted from systems external to the application in question.

During this interface analysis step, the format and aesthetics of the content (as it is displayed by the interface) are considered. Among the questions that are asked and answered are:



- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right-hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data?
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color be used to enhance understanding?
- How will error messages and warnings be presented to the user?

The answers to these (and other) questions will help you to establish requirements for content presentation.

#### 11.3.4 Analysis of the Work Environment

Hackos and Redish [Hac98] discuss the importance of work environment analysis when they state:

People do not perform their work in isolation. They are influenced by the activity around them, the physical characteristics of the workplace, the type of equipment they are using, and the work relationships they have with other people. If the products you design do not fit into the environment, they may be difficult or frustrating to use.

In some applications the user interface for a computer-based system is placed in a “user-friendly location” (e.g., proper lighting, good display height, easy keyboard access), but in others (e.g., a factory floor or an airplane cockpit), lighting may be suboptimal, noise may be a factor, a keyboard or mouse may not be an option, display placement may be less than ideal. The interface designer may be constrained by factors that mitigate against ease of use.

In addition to physical environmental factors, the workplace culture also comes into play. Will system interaction be measured in some manner (e.g., time per transaction or accuracy of a transaction)? Will two or more people have to share information before an input can be provided? How will support be provided to users of the system? These and many related questions should be answered before the interface design commences.

### 11.4 INTERFACE DESIGN STEPS

**note:**

“Interactive design [is] a seamless blend of graphic arts, technology, and psychology.”

**Brad Wieners**

Once interface analysis has been completed, all tasks (or objects and actions) required by the end user have been identified in detail and the interface design activity commences. Interface design, like all software engineering design, is an iterative process. Each user interface design step occurs a number of times, elaborating and refining information developed in the preceding step.

Although many different user interface design models (e.g., [Nor86], [Nie00]) have been proposed, all suggest some combination of the following steps:

1. Using information developed during interface analysis (Section 11.3), define interface objects and actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
3. Depict each interface state as it will actually look to the end user.
4. Indicate how the user interprets the state of the system from information provided through the interface.

In some cases, you can begin with sketches of each interface state (i.e., what the user interface looks like under various circumstances) and then work backward to define objects, actions, and other important design information. Regardless of the sequence of design tasks, you should (1) always follow the golden rules discussed

in Section 11.1, (2) model how the interface will be implemented, and (3) consider the environment (e.g., display technology, operating system, development tools) that will be used.

### 11.4.1 Applying Interface Design Steps

The definition of interface objects and the actions that are applied to them is an important step in interface design. To accomplish this, user scenarios are parsed in much the same way as described in Chapter 6. That is, a use case is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified. A *source object* (e.g., a report icon) is dragged and dropped onto a *target object* (e.g., a printer icon). The implication of this action is to create a hard-copy report. An *application object* represents application-specific data that are not directly manipulated as part of screen interaction. For example, a mailing list is used to store names for a mailing. The list itself might be sorted, merged, or purged (menu-based actions), but it is not dragged and dropped via user interaction.

When you are satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed. Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items are conducted. If a real-world metaphor is appropriate for the application, it is specified at this time, and the layout is organized in a manner that complements the metaphor.

To provide a brief illustration of the design steps noted previously, consider a user scenario for the *SafeHome* system (discussed in earlier chapters). A preliminary use case (written by the homeowner) for the interface follows:

**Preliminary use case:** I want to gain access to my *SafeHome* system from any remote location via the Internet. Using browser software operating on my notebook computer (while I'm at work or traveling), I can determine the status of the alarm system, arm or disarm the system, reconfigure security zones, and view different rooms within the house via preinstalled video cameras.

To access *SafeHome* from a remote location, I provide an identifier and a password. These define levels of access (e.g., all users may not be able to reconfigure the system) and provide security. Once validated, I can check the status of the system and change the status by arming or disarming *SafeHome*. I can reconfigure the system by displaying a floor plan of the house, viewing each of the security sensors, displaying each currently configured zone, and modifying zones as required. I can view the interior of the house via strategically placed video cameras. I can pan and zoom each camera to provide different views of the interior.

Based on this use case, the following homeowner tasks, objects, and data items are identified:

- *accesses the SafeHome system*
- *enters an **ID** and **password** to allow remote access*
- *checks **system status***
- *arms or disarms SafeHome system*
- *displays **floor plan** and **sensor locations***
- *displays **zones** on floor plan*
- *changes **zones** on floor plan*
- *displays **video camera locations** on floor plan*
- *selects **video camera** for viewing*
- *views **video images** (four frames per second)*
- *pans or zooms the **video camera***

Objects (boldface) and actions (italics) are extracted from this list of homeowner tasks. The majority of objects noted are application objects. However, **video camera location** (a source object) is dragged and dropped onto **video camera** (a target object) to create a **video image** (a window with video display).



Although automated tools can be useful in developing layout prototypes, sometimes a pencil and paper are all that are needed.

A preliminary sketch of the screen layout for video monitoring is created (Figure 11.3).<sup>6</sup> To invoke the video image, a video camera location icon, C, located in the floor plan displayed in the monitoring window is selected. In this case a camera location in the living room (LR) is then dragged and dropped onto the video camera icon in the upper left-hand portion of the screen. The video image window appears, displaying streaming video from the camera located in the LR. The zoom and pan control slides are used to control the magnification and direction of the video image. To select a view from another camera, the user simply drags and drops a different camera location icon into the camera icon in the upper left-hand corner of the screen.

The layout sketch shown would have to be supplemented with an expansion of each menu item within the menu bar, indicating what actions are available for the video monitoring mode (state). A complete set of sketches for each homeowner task noted in the user scenario would be created during the interface design.

### 11.4.2 User Interface Design Patterns

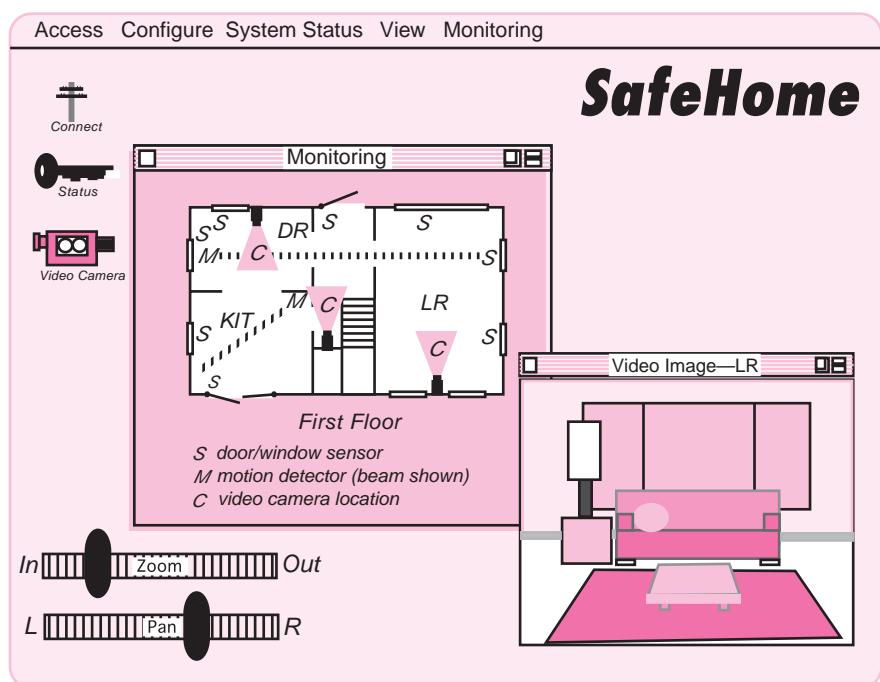
Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. As I noted earlier in this book, a design pattern is

---

<sup>6</sup> Note that this differs somewhat from the implementation of these features in earlier chapters. This might be considered a first draft design and represents one alternative that might be considered.

**FIGURE 11.3**

Preliminary screen layout



an abstraction that prescribes a design solution to a specific, well-bounded design problem.

As an example of a commonly encountered interface design problem, consider a situation in which a user must enter one or more calendar dates, sometimes months in advance. There are many possible solutions to this simple problem, and a number of different patterns that might be proposed. Laakso [Laa00] suggests a pattern called **CalendarStrip** that produces a continuous, scrollable calendar in which the current date is highlighted and future dates may be selected by picking them from the calendar. The calendar metaphor is well known to every user and provides an effective mechanism for placing a future date in context.

A vast array of interface design patterns has been proposed over the past decade. A more detailed discussion of user interface design patterns is presented in Chapter 12. In addition, Erickson [Eri08] provides pointers to many Web-based collections.

#### 11.4.3 Design Issues

As the design of a user interface evolves, four common design issues almost always surface: system response time, user help facilities, error information handling, and

#### WebRef

A wide variety of UI design patterns has been proposed. For pointers to a variety of patterns sites, visit [www.hcipatterns.org](http://www.hcipatterns.org).

command labeling. Unfortunately, many designers do not address these issues until relatively late in the design process (sometimes the first inkling of a problem doesn't occur until an operational prototype is available). Unnecessary iteration, project delays, and end-user frustration often result. It is far better to establish each as a design issue to be considered at the beginning of software design, when changes are easy and costs are low.

**Response time.** System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.

System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress are inevitable. *Variability* refers to the deviation from average response time, and in many ways, it is the most important response time characteristic. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long. For example, a 1-second response to a command will often be preferable to a response that varies from 0.1 to 2.5 seconds. When variability is significant, the user is always off balance, always wondering whether something "different" has occurred behind the scenes.



### Quote:

"A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools."

Douglas Adams

**Help facilities.** Almost every user of an interactive, computer-based system requires help now and then. In some cases, a simple question addressed to a knowledgeable colleague can do the trick. In others, detailed research in a multivolume set of "user manuals" may be the only option. In most cases, however, modern software provides online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface.

A number of design issues [Rub88] must be addressed when a help facility is considered:

- Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for all functions.
- How will the user request help? Options include a help menu, a special function key, or a HELP command.
- How will help be represented? Options include a separate window, a reference to a printed document (less than ideal), or a one- or two-line suggestion produced in a fixed screen location.
- How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or control sequence.

- How will help information be structured? Options include a “flat” structure in which all information is accessed through a keyword, a layered hierarchy of information that provides increasing detail as the user proceeds into the structure, or the use of hypertext.

**note:**

"The interface from hell—to correct this error and continue, enter any 11-digit prime number ..."

Author unknown

**Error handling.** Error messages and warnings are “bad news” delivered to users of interactive systems when something has gone awry. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration. There are few computer users who have not encountered an error of the form: *“Application XXX has been forced to quit because an error of type 1023 has been encountered.”* Somewhere, an explanation for error 1023 must exist; otherwise, why would the designers have added the identification? Yet, the error message provides no real indication of what went wrong or where to look to get additional information. An error message presented in this manner does nothing to assuage user anxiety or to help correct the problem.

In general, every error message or warning produced by an interactive system should have the following characteristics:

 **What characteristics should a “good” error message have?**

- The message should describe the problem in jargon that the user can understand.
- The message should provide constructive advice for recovering from the error.
- The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).
- The message should be accompanied by an audible or visual cue. That is, a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the “error color.”
- The message should be “nonjudgmental.” That is, the wording should never place blame on the user.

Because no one really likes bad news, few users will like an error message no matter how well designed. But an effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur.

**Menu and command labeling.** The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. Today, the use of window-oriented, point-and-pick interfaces has reduced reliance on typed commands, but some power-users continue to prefer a command-oriented mode of interaction. A number of design

issues arise when typed commands or menu labels are provided as a mode of interaction:

- Will every menu option have a corresponding command?
- What form will commands take? Options include a control sequence (e.g., alt-P), function keys, or a typed word.
- How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?
- Can commands be customized or abbreviated by the user?
- Are menu labels self-explanatory within the context of the interface?
- Are submenus consistent with the function implied by a master menu item?

As I noted earlier in this chapter, conventions for command usage should be established across all applications. It is confusing and often error-prone for a user to type alt-D when a graphics object is to be duplicated in one application and alt-D when a graphics object is to be deleted in another. The potential for error is obvious.

#### WebRef

Guidelines for developing accessible software can be found at [www3.ibm.com/able/guidelines/software/accesssoftware.html](http://www3.ibm.com/able/guidelines/software/accesssoftware.html).

**Application accessibility.** As computing applications become ubiquitous, software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. *Accessibility* for users (and software engineers) who may be physically challenged is an imperative for ethical, legal, and business reasons. A variety of accessibility guidelines (e.g., [W3C03])—many designed for Web applications but often applicable to all types of software—provide detailed suggestions for designing interfaces that achieve varying levels of accessibility. Others (e.g., [App08], [Mic08]) provide specific guidelines for “assistive technology” that addresses the needs of those with visual, hearing, mobility, speech, and learning impairments.

**Internationalization.** Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages. Too often, interfaces are designed for one locale and language and then jury-rigged to work in other countries. The challenge for interface designers is to create “globalized” software. That is, user interfaces should be designed to accommodate a generic core of functionality that can be delivered to all who use the software. *Localization* features enable the interface to be customized for a specific market.

A variety of internationalization guidelines (e.g., [IBM03]) are available to software engineers. These guidelines address broad design issues (e.g., screen layouts may differ in various markets) and discrete implementation issues (e.g., different alphabets may create specialized labeling and spacing requirements). The *Unicode* standard [Uni03] has been developed to address the daunting challenge of managing dozens of natural languages with hundreds of characters and symbols.

## SOFTWARE TOOLS



### User Interface Development

**Objective:** These tools enable a software engineer to create a sophisticated GUI with relatively little custom software development. The tools provide access to reusable components and make the creation of an interface a matter of selecting from predefined capabilities that are assembled using the tool.

**Mechanics:** Modern user interfaces are constructed using a set of reusable components that are coupled with some custom components developed to provide specialized features. Most user interface development tools enable a software engineer to create an interface using “drag and drop” capability. That is, the developer selects from many predefined capabilities (e.g., forms builders, interaction mechanisms, command processing capability) and places these capabilities within the content of the interface to be created.

#### Representative Tools:<sup>7</sup>

*LegaSuite GUI*, developed by Seagull Software ([www.seagullsoftware.com](http://www.seagullsoftware.com)), enabled the creation of browser-based GUIs and provide facilities for reengineering antiquated interfaces.

*Motif Common Desktop Environment*, developed by The Open Group ([www.osf.org/tech/desktop/cde/](http://www.osf.org/tech/desktop/cde/)), is an integrated graphical user interface for open systems desktop computing. It delivers a single, standard graphical interface for the management of data and files (the graphical desktop) and applications.

*Altia Design 8.0*, developed by Altia ([www.altia.com](http://www.altia.com)), is a tool for creating GUIs on a variety of different platforms (e.g., automotive, handheld, industrial).

## 11.5 WEBAPP INTERFACE DESIGN

Every user interface—whether it is designed for a WebApp, a traditional software application, a consumer product, or an industrial device—should exhibit the usability characteristics that were discussed earlier in this chapter. Dix [Dix99] argues that you should design a WebApp interface so that it answers three primary questions for the end user:



If it is likely that users may enter your WebApp at various locations and levels in the content hierarchy, be sure to design every page with navigation features that will lead the user to other points of interest.

*Where am I?* The interface should (1) provide an indication of the WebApp that has been accessed<sup>8</sup> and (2) inform the user of her location in the content hierarchy.

*What can I do now?* The interface should always help the user understand his current options—what functions are available, what links are live, what content is relevant?

*Where have I been, where am I going?* The interface must facilitate navigation. Hence, it must provide a “map” (implemented in a way that is easy to understand) of where the user has been and what paths may be taken to move elsewhere within the WebApp.

An effective WebApp interface must provide answers for each of these questions as the end user navigates through content and functionality.

<sup>7</sup> Tools noted here do not represent an endorsement, but rather a sampling of tools in this category.

<sup>8</sup> Each of us has bookmarked a Web page, only to revisit it later and have no indication of the website or the context for the page (as well as no way to move to another location within the site).

### 11.5.1 Interface Design Principles and Guidelines

**note:**

"If a site is perfectly usable but it lacks an elegant and appropriate design style, it will fail."

Curt Cloninger

**KEY POINT**

A good WebApp interface is understandable and forgiving, providing the user with a sense of control.

Is there a set of basic principles that can be applied as you design a GUI?

The user interface of a WebApp is its "first impression." Regardless of the value of its content, the sophistication of its processing capabilities and services, and the overall benefit of the WebApp itself, a poorly designed interface will disappoint the potential user and may, in fact, cause the user to go elsewhere. Because of the sheer volume of competing WebApps in virtually every subject area, the interface must "grab" a potential user immediately.

Bruce Tognazzi [Tog01] defines a set of fundamental characteristics that all interfaces should exhibit and in doing so, establishes a philosophy that should be followed by every WebApp interface designer:

Effective interfaces are visually apparent and forgiving, instilling in their users a sense of control. Users quickly see the breadth of their options, grasp how to achieve their goals, and do their work.

Effective interfaces do not concern the user with the inner workings of the system. Work is carefully and continuously saved, with full option for the user to undo any activity at any time.

Effective applications and services perform a maximum of work, while requiring a minimum of information from users.

In order to design WebApp interfaces that exhibit these characteristics, Tognazzi [Tog01] identifies a set of overriding design principles:<sup>9</sup>

**Anticipation.** *A WebApp should be designed so that it anticipates the user's next move.* For example, consider a customer support WebApp developed by a manufacturer of computer printers. A user has requested a content object that presents information about a printer driver for a newly released operating system. The designer of the WebApp should anticipate that the user might request a download of the driver and should provide navigation facilities that allow this to happen without requiring the user to search for this capability.

**Communication.** *The interface should communicate the status of any activity initiated by the user.* Communication can be obvious (e.g., a text message) or subtle (e.g., an image of a sheet of paper moving through a printer to indicate that printing is under way). The interface should also communicate user status (e.g., the user's identification) and her location within the WebApp content hierarchy.

**Consistency.** *The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout) should be consistent throughout the WebApp.* For example, if underlined blue text implies a navigation link, content should never incorporate blue underlined text that does not imply a link. In addition, an object, say a yellow triangle, used to

---

<sup>9</sup> Tognazzi's original principles have been adapted and extended for use this book. See [Tog01] for further discussion of these principles.

indicate a caution message before the user invokes a particular function or action, should not be used for other purposes elsewhere in the WebApp. Finally, every feature of the interface should respond in a manner that is consistent with user expectations.<sup>10</sup>

**Controlled autonomy.** *The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.* For example, navigation to secure portions of the WebApp should be controlled by userID and password, and there should be no navigation mechanism that enables a user to circumvent these controls.

**vote:**

"The best journey is the one with the fewest steps. Shorten the distance between the user and their goal."

Author unknown

**WebRef**

A search on the Web will uncover many available libraries, e.g., Java API packages, interfaces, and classes at [java.sun.com](http://java.sun.com) or COM, DCOM, and Type Libraries at [msdn.Microsoft.com](http://msdn.Microsoft.com).

**Efficiency.** *The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the developer who designs and builds it or the client-server environment that executes it.* Tognazzi [Tog01] discusses this when he writes: "This simple truth is why it is so important for everyone involved in a software project to appreciate the importance of making user productivity goal one and to understand the vital difference between building an efficient system and empowering an efficient user."

**Flexibility.** *The interface should be flexible enough to enable some users to accomplish tasks directly and others to explore the WebApp in a somewhat random fashion.* In every case, it should enable the user to understand where he is and provide the user with functionality that can undo mistakes and retrace poorly chosen navigation paths.

**Focus.** *The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.* In all hypermedia there is a tendency to route the user to loosely related content. Why? Because it's very easy to do! The problem is that the user can rapidly become lost in many layers of supporting information and lose sight of the original content that she wanted in the first place.

**Fitt's law.** *"The time to acquire a target is a function of the distance to and size of the target"* [Tog01]. Based on a study conducted in the 1950s [Fit54], Fitt's law "is an effective method of modeling rapid, aimed movements, where one appendage (like a hand) starts at rest at a specific start position, and moves to rest within a target area" [Zha02]. If a sequence of selections or standardized inputs (with many different options within the sequence) is defined by a user task, the first selection (e.g., mouse pick) should be physically close to the next selection. For example, consider a WebApp home page interface at an e-commerce site that sells consumer electronics.

Each user option implies a set of follow-on user choices or actions. For example, the "buy a product" option requires that the user enter a product category followed by the product name. The product category (e.g., audio equipment, televisions, DVD

---

<sup>10</sup> Tognazzi [Tog01] notes that the only way to be sure that user expectations are properly understood is through comprehensive user testing (Chapter 20).

players) appears as a pull-down menu as soon as “buy a product” is picked. Therefore, the next choice is immediately obvious (it is nearby) and the time to acquire it is negligible. If, on the other hand, the choice appeared on a menu that was located on the other side of the screen, the time for the user to acquire it (and then make the choice) would be far too long.

**Human interface objects.** *A vast library of reusable human interface objects has been developed for WebApps. Use them.* Any interface object that can be “seen, heard, touched or otherwise perceived” [Tog01] by an end user can be acquired from any one of a number of object libraries.

**Latency reduction.** *Rather than making the user wait for some internal operation to complete (e.g., downloading a complex graphical image), the WebApp should use multitasking in a way that lets the user proceed with work as if the operation has been completed.* In addition to reducing latency, delays must be acknowledged so that the user understands what is happening. This includes (1) providing audio feedback when a selection does not result in an immediate action by the WebApp, (2) displaying an animated clock or progress bar to indicate that processing is under way, and (3) providing some entertainment (e.g., an animation or text presentation) while lengthy processing occurs.

**Learnability.** *A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.* In general the interface should emphasize a simple, intuitive design that organizes content and functionality into categories that are obvious to the user.



Metaphors are an excellent idea because they mirror real-world experience. Just be sure that the metaphor you choose is well known to end users.

**Metaphors.** *An interface that uses an interaction metaphor is easier to learn and easier to use, as long as the metaphor is appropriate for the application and the user.* A metaphor should call on images and concepts from the user’s experience, but it does not need to be an exact reproduction of a real-world experience. For example, an e-commerce site that implements automated bill paying for a financial institution, uses a checkbook metaphor (not surprisingly) to assist the user in specifying and scheduling bill payments. However, when a user “writes” a check, he need not enter the complete payee name but can pick from a list of payees or have the system select based on the first few typed letters. The metaphor remains intact, but the user gets an assist from the WebApp.

**Maintain work product integrity.** *A work product (e.g., a form completed by the user, a user-specified list) must be automatically saved so that it will not be lost if an error occurs.* Each of us has experienced the frustration associated with completing a lengthy WebApp form only to have the content lost because of an error (made by us, by the WebApp, or in transmission from client to server). To avoid this, a WebApp should be designed to autosave all user-specified data. The interface should support this function and provide the user with an easy mechanism for recovering “lost” information.

**Readability.** All information presented through the interface should be readable by young and old. The interface designer should emphasize readable type styles, font sizes, and color background choices that enhance contrast.

**Track state.** When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off. In general, cookies can be designed to store state information. However, cookies are a controversial technology, and other design solutions may be more palatable for some users.

**Visible navigation.** A well-designed WebApp interface provides “the illusion that users are in the same place, with the work brought to them” [Tog01]. When this approach is used, navigation is not a user concern. Rather, the user retrieves content objects and selects functions that are displayed and executed through the interface.

## SAFEHOME



### Interface Design Review

**The scene:** Doug Miller’s office.

**The players:** Doug Miller (manager of the SafeHome software engineering group) and Vinod Raman, a member of the SafeHome product software engineering team.

#### The conversation:

**Doug:** Vinod, have you and the team had a chance to review the **SafeHomeAssured.com** e-commerce interface prototype?

**Vinod:** Yeah . . . we all went through it from a technical point of view, and I have a bunch of notes. I e-mailed ‘em to Sharon [manager of the WebApp team for the outsourcing vendor for the SafeHome e-commerce website] yesterday.

**Doug:** You and Sharon can get together and discuss the small stuff . . . give me a summary of the important issues.

**Vinod:** Overall, they’ve done a good job, nothing ground breaking, but it’s a typical e-commerce interface, decent aesthetics, reasonable layout, they’ve hit all the important functions . . .

**Doug (smiling ruefully):** But?

**Vinod:** Well, there are a few things . . .

**Doug:** Such as . . .

**Vinod (showing Doug a sequence of story-boards for the interface prototype):** Here’s the major functions menu that’s displayed on the home page:

**Learn about SafeHome.**

**Describe your home.**

**Get SafeHome component recommendations.**

**Purchase a SafeHome system.**

**Get technical support.**

The problem isn’t with these functions. They’re all okay, but the level of abstraction isn’t right.

**Doug:** They’re all major functions, aren’t they?

**Vinod:** They are, but here’s the thing . . . you can purchase a system by inputting a list of components . . . no real need to describe the house if you don’t want to. I’d suggest only four menu options on the home page:

**Learn about SafeHome.**

**Specify the SafeHome system you need.**

**Purchase a SafeHome system.**

**Get technical support.**

When you select **Specify the SafeHome system you need**, you’ll then have the following options:

**Select SafeHome components.**

**Get SafeHome component recommendations.**

If you’re a knowledgeable user, you’ll select components from a set of categorized pull-down menus for sensors, cameras, control panels, etc. If you need help, you’ll ask for a recommendation and that will require that you describe your house. I think it’s a bit more logical.

**Doug:** I agree. Have you talked with Sharon about this?

**Vinod:** No, I want to discuss this with marketing first; then I’ll give her a call.

Nielsen and Wagner [Nie96] suggest a few pragmatic interface design guidelines (based on their redesign of a major WebApp) that provide a nice complement to the principles suggested earlier in this section:

- Reading speed on a computer monitor is approximately 25 percent slower than reading speed for hardcopy. Therefore, do not force the user to read voluminous amounts of text, particularly when the text explains the operation of the WebApp or assists in navigation.
- Avoid “under construction” signs—an unnecessary link is sure to disappoint.
- Users prefer not to scroll. Important information should be placed within the dimensions of a typical browser window.
- Navigation menus and head bars should be designed consistently and should be available on all pages that are available to the user. The design should not rely on browser functions to assist in navigation.
- Aesthetics should never supersede functionality. For example, a simple button might be a better navigation option than an aesthetically pleasing, but vague image or icon whose intent is unclear.
- Navigation options should be obvious, even to the casual user. The user should not have to search the screen to determine how to link to other content or services.

A well-designed interface improves the user’s perception of the content or services provided by the site. It need not necessarily be flashy, but it should always be well structured and ergonomically sound.

### 11.5.2 Interface Design Workflow for WebApps

Earlier in this chapter I noted that user interface design begins with the identification of user, task, and environmental requirements. Once user tasks have been identified, user scenarios (use cases) are created and analyzed to define a set of interface objects and actions.

Information contained within the requirements model forms the basis for the creation of a screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Tools are then used to prototype and ultimately implement the interface design model. The following tasks represent a rudimentary workflow for WebApp interface design:

1. **Review information contained in the requirements model and refine as required.**
2. **Develop a rough sketch of the WebApp interface layout.** An interface prototype (including the layout) may have been developed as part of the requirements modeling activity. If the layout already exists, it should be reviewed and refined as required. If the interface layout has not been

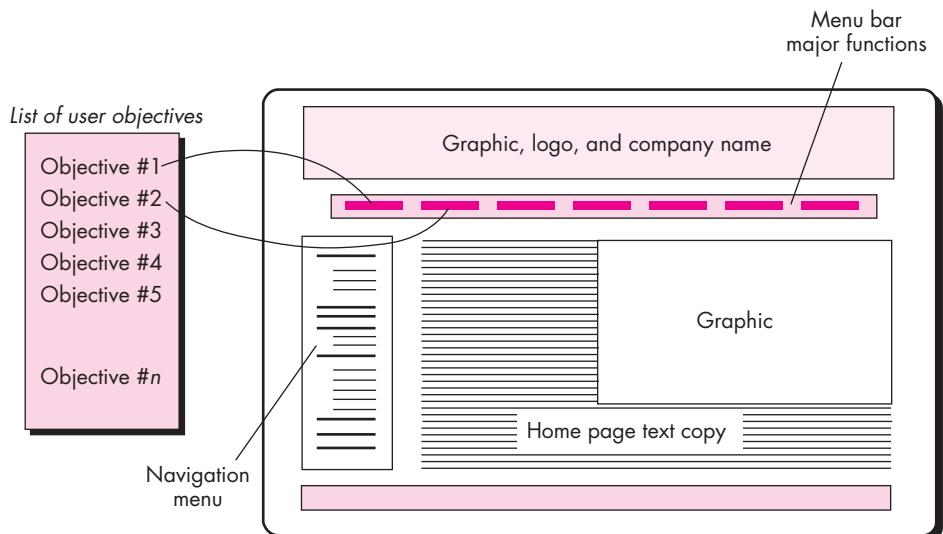
#### Quote:

“People have very little patience for poorly designed WWW sites.”

Jakob Nielsen  
and  
Annette Wagner

**FIGURE 11.4**

Mapping user objectives into interface actions



developed, you should work with stakeholders to develop it at this time. A schematic first-cut layout sketch is shown in Figure 11.4.

- 3. Map user objectives into specific interface actions.** For the vast majority of WebApps, the user will have a relatively small set of primary objectives. These should be mapped into specific interface actions as shown in Figure 11.4. In essence, you must answer the following question: "How does the interface enable the user to accomplish each objective?"
- 4. Define a set of user tasks that are associated with each action.** Each interface action (e.g., "buy a product") is associated with a set of user tasks. These tasks have been identified during requirements modeling. During design, they must be mapped into specific interactions that encompass navigation issues, content objects, and WebApp functions.
- 5. Storyboard screen images for each interface action.** As each action is considered, a sequence of storyboard images (screen images) should be created to depict how the interface responds to user interaction. Content objects should be identified (even if they have not yet been designed and developed), WebApp functionality should be shown, and navigation links should be indicated.
- 6. Refine interface layout and storyboards using input from aesthetic design.** In most cases, you'll be responsible for rough layout and storyboarding, but the aesthetic look and feel for a major commercial site is often developed by artistic, rather than technical, professionals. Aesthetic design (Chapter 13) is integrated with the work performed by the interface designer.

7. **Identify user interface objects that are required to implement the interface.** This task may require a search through an existing object library to find those reusable objects (classes) that are appropriate for the WebApp interface. In addition, any custom classes are specified at this time.
8. **Develop a procedural representation of the user's interaction with the interface.** This optional task uses UML sequence diagrams and/or activity diagrams (Appendix 1) to depict the flow of activities (and decisions) that occur as the user interacts with the WebApp.
9. **Develop a behavioral representation of the interface.** This optional task makes use of UML state diagrams (Appendix 1) to represent state transitions and the events that cause them. Control mechanisms (i.e., the objects and actions available to the user to alter a WebApp state) are defined.
10. **Describe the interface layout for each state.** Using design information developed in Tasks 2 and 5, associate a specific layout or screen image with each WebApp state described in Task 8.
11. **Refine and review the interface design model.** Review of the interface should focus on usability.

It is important to note that the final task set you choose should be adapted to the special requirements of the application that is to be built.

## 11.6 DESIGN EVALUATION

Once you create an operational user interface prototype, it must be evaluated to determine whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal “test drive,” in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end users.

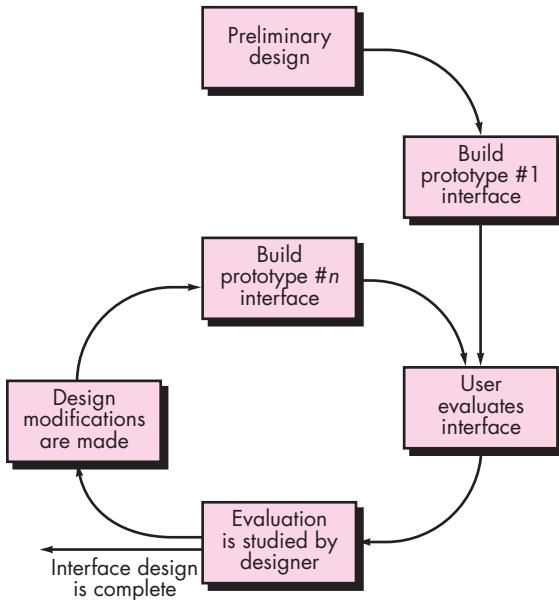
The user interface evaluation cycle takes the form shown in Figure 11.5. After the design model has been completed, a first-level prototype is created. The prototype is evaluated by the user,<sup>11</sup> who provides you with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), you can extract information from these data (e.g., 80 percent of all users did not like the mechanism for saving data files). Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary.

---

<sup>11</sup> It is important to note that experts in ergonomics and interface design may also conduct reviews of the interface. These reviews are called *heuristic evaluations* or *cognitive walkthroughs*.

**FIGURE 11.5**

The interface design evaluation cycle



The prototyping approach is effective, but is it possible to evaluate the quality of a user interface before a prototype is built? If you identify and correct potential problems early, the number of loops through the evaluation cycle will be reduced and development time will shorten. If a design model of the interface has been created, a number of evaluation criteria [Mor81] can be applied during early design reviews:

1. The length and complexity of the requirements model or written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
2. The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.
3. The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.
4. Interface style, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

Once the first prototype is built, you can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect qualitative data, questionnaires can be distributed to users of the prototype. Questions can be: (1) simple yes/no response, (2) numeric response, (3) scaled (subjective) response, (4) Likert

scales (e.g., strongly agree, somewhat agree), (5) percentage (subjective) response, or (6) open-ended.

If quantitative data are desired, a form of time-study analysis can be conducted. Users are observed during interaction, and data—such as number of tasks correctly completed over a standard time period, frequency of actions, sequence of actions, time spent “looking” at the display, number and types of errors, error recovery time, time spent using help, and number of help references per standard time period—are collected and used as a guide for interface modification.

A complete discussion of user interface evaluation methods is beyond the scope of this book. For further information, see [Hac98] and [Sto05].

## 11.7 SUMMARY

The user interface is arguably the most important element of a computer-based system or product. If the interface is poorly designed, the user’s ability to tap the computational power and informational content of an application may be severely hindered. In fact, a weak interface may cause an otherwise well-designed and solidly implemented application to fail.

Three important principles guide the design of effective user interfaces: (1) place the user in control, (2) reduce the user’s memory load, and (3) make the interface consistent. To achieve an interface that abides by these principles, an organized design process must be conducted.

The development of a user interface begins with a series of analysis tasks. User analysis defines the profiles of various end users and is gathered from a variety of business and technical sources. Task analysis defines user tasks and actions using either an elaborative or object-oriented approach, applying use cases, task and object elaboration, workflow analysis, and hierarchical task representations to fully understand the human-computer interaction. Environmental analysis identifies the physical and social structures in which the interface must operate.

Once tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. This provides a basis for the creation of a screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Design issues such as response time, command and action structure, error handling, and help facilities are considered as the design model is refined. A variety of implementation tools are used to build a prototype for evaluation by the user.

Like interface design for conventional software, the design of WebApp interfaces describes the structure and organization of the user interface and includes a representation of screen layout, a definition of the modes of interaction, and a description of navigation mechanisms. A set of interface design principles and an interface design workflow guide a WebApp designer when layout and interface control mechanisms are designed.

The user interface is the window into the software. In many cases, the interface molds a user's perception of the quality of the system. If the "window" is smudged, wavy, or broken, the user may reject an otherwise powerful computer-based system.

## PROBLEMS AND POINTS TO PONDER

**11.1.** Describe the worst interface that you have ever worked with and critique it relative to the concepts introduced in this chapter. Describe the best interface that you have ever worked with and critique it relative to the concepts introduced in this chapter.

**11.2.** Develop two additional design principles that "place the user in control."

**11.3.** Develop two additional design principles that "reduce the user's memory load."

**11.4.** Develop two additional design principles that "make the interface consistent."

**11.5.** Consider one of the following interactive applications (or an application assigned by your instructor):

- a. A desktop publishing system
- b. A computer-aided design system
- c. An interior design system (as described in Section 11.3.2)
- d. An automated course registration system for a university
- e. A library management system
- f. An Internet-based polling booth for public elections
- g. A home banking system
- h. An interactive application assigned by your instructor

Develop a user model, design model, mental model, and an implementation model, for any one of these systems.

**11.6.** Perform a detailed task analysis for any one of the systems listed in Problem 11.5 Use either an elaborative or object-oriented approach.

**11.7.** Add at least five additional questions to the list developed for content analysis in Section 11.3.3.

**11.8.** Continuing Problem 11.5, define interface objects and actions for the application you have chosen. Identify each object type.

**11.9.** Develop a set of screen layouts with a definition of major and minor menu items for the system you chose in Problem 11.5.

**11.10.** Develop a set of screen layouts with a definition of major and minor menu items for the *SafeHome* system. You may elect to take a different approach than the one shown for the screen layout in Figure 11.3.

**11.11.** Describe your approach to user help facilities for the task analysis design model and task analysis you have performed as part of Problems 11.5 through 11.8.

**11.12.** Provide a few examples that illustrate why response time variability can be an issue.

**11.13.** Develop an approach that would automatically integrate error messages and a user help facility. That is, the system would automatically recognize the error type and provide a help window with suggestions for correcting it. Perform a reasonably complete software design that considers appropriate data structures and algorithms.

**11.14.** Develop an interface evaluation questionnaire that contains 20 generic questions that would apply to most interfaces. Have 10 classmates complete the questionnaire for an interactive system that you all use. Summarize the results and report them to your class.

## FURTHER READINGS AND INFORMATION SOURCES

Although his book is not specifically about human-computer interfaces, much of what Donald Norman (*The Design of Everyday Things*, reissue edition, Currency/Doubleday, 1990) has to say about the psychology of effective design applies to the user interface. It is recommended reading for anyone who is serious about doing high-quality interface design.

Graphical user interfaces are ubiquitous in the modern world of computing. Whether it's an ATM, a mobile phone, an electronic dashboard in an automobile, a website, or a business application, the user interface provides a window into the software. It is for this reason that books addressing interface design abound. Butow (*User Interface Design for Mere Mortals*, Addison-Wesley, 2007), Galitz (*The Essential Guide to User Interface Design*, 3d ed., Wiley, 2007), Lehikoinen and his colleagues (*Personal Content Experience: Managing Digital Life in the Mobile Age*, Wiley-Interscience, 2007), Cooper and his colleagues (*About Face 3: The Essentials of Interaction Design*, 3d ed., Wiley, 2007), Ballard (*Designing the Mobile User Experience*, Wiley, 2007), Nielsen (*Coordinating User Interfaces for Consistency*, Morgan-Kaufmann, 2006), Lauesen (*User Interface Design: A Software Engineering Perspective*, Addison-Wesley, 2005), Barfield (*The User Interface: Concepts and Design*, Bosko Books, 2004) all discuss usability, user interface concepts, principles, and design techniques and contain many useful examples.

Older books by Beyer and Holtzblatt (*Contextual Design: A Customer Centered Approach to Systems Design*, Morgan-Kaufmann, 2002), Raskin (*The Humane Interface*, Addison-Wesley, 2000), Constantine and Lockwood (*Software for Use*, ACM Press, 1999), and Mayhew (*The Usability Engineering Lifecycle*, Morgan-Kaufmann, 1999) present treatments that provide additional design guidelines and principles as well as suggestions for interface requirements elicitation, design modeling, implementation, and testing.

Johnson (*GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*, Morgan-Kaufmann, 2000) provides useful guidance for those that learn more effectively by examining counterexamples. An enjoyable book by Cooper (*The Inmates Are Running the Asylum*, Sams Publishing, 1999) discusses why high-tech products drive us crazy and how to design ones that don't.

A wide variety of information sources on user interface design are available on the Internet. An up-to-date list of World Wide Web references that are relevant to user interface design can be found at the SEPA website: [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm).

## PATTERN-BASED DESIGN

# 12

### KEY CONCEPTS

design mistakes	.... 359
forces	..... 349
frameworks	... 352
granularity	.... 369
pattern languages	.... 353
patterns	
architectural	. 360
behavioral	.... 351

**E**ach of us has encountered a design problem and silently thought: *I wonder if anyone has developed a solution for this?* The answer is almost always—yes! The problem is finding the solution; ensuring that it does, in fact, fit the problem you've encountered; understanding the constraints that may restrict the manner in which the solution is applied; and finally, translating the proposed solution into your design environment.

But what if the solution were codified in some manner? What if there was a standard way of describing a problem (so you could look it up), and an organized method for representing the solution to the problem? It turns out that software problems have been codified and described using a standardized template, and solutions to them (along with constraints) have been proposed. Called *design patterns*,

### QUICK LOOK

**What is it?** Pattern-based design creates a new application by finding a set of proven solutions to a clearly delineated set of problems. Each problem and its solution is described by a design pattern that has been cataloged and vetted by other software engineers who have encountered the problem and implemented the solution while designing other applications. Each design pattern provides you with a proven approach to one part of the problem to be solved.

**Who does it?** A software engineer examines each problem encountered for a new application and then attempts to find a relevant solution by searching one or more patterns repositories.

**Why is it important?** Have you ever heard the phrase “reinventing the wheel”? It happens all the time in software development, and it’s a waste of time and energy. By using existing design patterns, you can acquire a proven solution for a specific problem. As each pattern is applied, solutions are integrated and the application to be built moves closer to a complete design.

**What are the steps?** The requirements model is examined in order to isolate the hierarchical set of problems to be solved. The problem space is partitioned so that subsets of problems associated with specific software functions and features can be identified. Problems can also be organized by type: architectural, component-level, algorithmic, user interface, etc. Once a subset of problems is defined, one or more pattern repositories are searched to determine if an existing design pattern, represented at an appropriate level of abstraction, exists. Patterns that are applicable are adapted to the specific needs of the software to be built. Custom problem solving is applied in situations for which no patterns can be found.

**What is the work product?** A design model that depicts the architectural structure, user interface, and component-level detail is developed.

**How do I ensure that I’ve done it right?** As each design pattern is translated into some element of the design model, work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

<b>component-level</b>	.....	<b>362</b>
<b>generative</b>	.....	<b>350</b>
<b>creational</b>	.....	<b>350</b>
<b>structural</b>	.....	<b>351</b>
<b>user interface</b>	.....	<b>366</b>
<b>WebApps</b>	.....	<b>368</b>

this codified method for describing problems and their solution allows the software engineering community to capture design knowledge in a way that enables it to be reused.

The early history of software patterns begins not with a computer scientist but a building architect, Christopher Alexander, who recognized that a recurring set of problems were encountered whenever a building was designed. He characterized these recurring problems and their solutions as *patterns*, describing them in the following manner [Ale77]:

Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use the solution a million times over without ever doing it the same way twice.

Alexander's ideas were first translated into the software world in books by Gamma [Gam95], Buschmann [Bus96], and their many colleagues.<sup>1</sup> Today, dozens of pattern repositories exist, and pattern-based design can be applied in many different application domains.

## 12.1 DESIGN PATTERNS



**Forces** are those characteristics of the problem and attributes of the solution that constrain the way in which the design can be developed.

A *design pattern* can be characterized as “a three-part rule which expresses a relation between a certain context, a problem, and a solution” [Ale79]. For software design, *context* allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment. A set of requirements, including limitations and constraints, acts as a *system of forces* that influences how the problem can be interpreted within its context and how the solution can be effectively applied.

To better understand these concepts, consider a situation<sup>2</sup> in which a person must travel between New York and Los Angeles. In this context, travel will occur within an industrialized country (the United States), using an existing transportation infrastructure (e.g., roads, airlines, railways). The system of forces that will affect the way in which the travel problem is solved will include: how quickly the person wants to get from New York to LA, whether the trip will include site-seeing or stopovers, how much money the person can spend, whether the trip is intended to accomplish a specific purpose, and the personal vehicles the person has at her disposal. Given these forces, the problem (traveling from New York to LA) can be better defined. For example, investigation (requirements gathering) indicates that the person has very little money, owns only a bicycle (and is an avid cyclist), wants to make the trip to raise money for her favorite charity, and has plenty of time to spare. The solution to the problem, given the context and the system of forces, might be a cross-country

<sup>1</sup> Earlier discussions of software patterns do exist, but these two classic books were the first cohesive treatments of the subject.

<sup>2</sup> This example has been adapted from [Cor98].

bike trip. If the forces were different (e.g., travel time must be minimized and the purpose of the trip is a business meeting), another solution might be more appropriate.

**quote:**

"Our responsibility is to do what we can, learn what we can, improve the solutions, and pass them on."

**Richard P.  
Feynman**

It is reasonable to argue that most problems have multiple solutions, but that a solution is effective only if it is appropriate within the context of the existing problem. It is the system of forces that causes a designer to choose a specific solution. The intent is to provide a solution that best satisfies the system of forces, even when these forces are contradictory. Finally, every solution has consequences that may have an impact on other aspects of the software and may themselves become part of the system of forces for other problems to be solved within the larger system.

Coplien [Cop05] characterizes an effective design pattern in the following way:

- *It solves a problem:* Patterns capture solutions, not just abstract principles or strategies.
- *It is a proven concept:* Patterns capture solutions with a track record, not theories or speculation.
- *The solution isn't obvious:* Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns *generate* a solution to a problem indirectly—a necessary approach for the most difficult problems of design.
- *It describes a relationship:* Patterns don't just describe modules, but describe deeper system structures and mechanisms.
- *The pattern has a significant human component (minimize human intervention).* All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

Stated even more pragmatically, a good design pattern captures hard-earned, pragmatic design knowledge in a way that enables others to reuse that knowledge “a million times over without ever doing it the same way twice.” A design pattern saves you from “reinventing the wheel,” or worse, inventing a “new wheel” that is slightly out of round, too small for its intended use, and too narrow for the ground it will roll over. Design patterns, if used effectively, will invariably make you a better software designer.

### 12.1.1 Kinds of Patterns

One of the reasons that software engineers are interested in (and intrigued by) design patterns is that human beings are inherently good at pattern recognition. If we weren’t, we’d be frozen in space and time—unable to learn from past experience, unwilling to venture forward because of our inability to recognize situations that might lead to high risk, unhinged by a world that seems to have no regularity or logical consistency. Luckily, none of this occurs because we do recognize patterns in virtually every aspect of our lives.

In the real world, the patterns we recognize are learned over a lifetime of experience. We recognize them instantly and inherently understand what they mean and how they might be used. Some of these patterns provide us with insight into

recurring phenomenon. For example, you're on your way home from work on the interstate when your navigation system (or car radio) informs you that a serious accident has occurred on the interstate in the opposing direction. You're 4 miles from the accident, but already you begin to see traffic slowing, recognizing a pattern that we'll call **RubberNecking**. People in the travel lanes moving in your direction are slowing at the sight of the accident to get a better view of what happened on the opposite side of the highway. The **RubberNecking** pattern yields remarkably predictable results (a traffic jam), but it does nothing more than describe a phenomenon. In patterns jargon, it might be called a *nongenerative* pattern because it describes a context and a problem but it does not provide any clear-cut solution.

### KEY POINT

A "generative" pattern describes the problem, a context, and forces, but it also describes a pragmatic solution to the problem.

When software design patterns are considered, we strive to identify and document *generative* patterns. That is, we identify a pattern that describes an important and repeatable aspect of a system and that provides us with a way to build that aspect within a system of forces that are unique to a given context. In an ideal setting, a collection of generative design patterns could be used to "generate" an application or computer-based system whose architecture enables it to adapt to change. Sometimes called *generativity*, "the successive application of several patterns, each encapsulating its own problem and forces, unfolds a larger solution which emerges indirectly as a result of the smaller solutions" [App00].

Design patterns span a broad spectrum of abstraction and application. *Architectural patterns* describe broad-based design problems that are solved using a structural approach. *Data patterns* describe recurring data-oriented problems and the data modeling solutions that can be used to solve them. *Component patterns* (also referred to as *design patterns*) address problems associated with the development of subsystems and components, the manner in which they communicate with one another, and their placement within a larger architecture. *Interface design patterns* describe common user interface problems and their solution with a system of forces that includes the specific characteristics of end users. *WebApp patterns* address a problem set that is encountered when building WebApps and often incorporates many of the other patterns categories just mentioned. At a lower level of abstraction, *idioms* describe how to implement all or part of a specific algorithm or data structure for a software component within the context of a specific programming language.

### Is there a way to categorize pattern types?

In their seminal book on design patterns, Gamma and his colleagues<sup>3</sup> [Gam95] focus on three types of patterns that are particularly relevant to object-oriented design: *creational patterns*, *structural patterns*, and *behavioral patterns*.

*Creational patterns* focus on the "creation, composition, and representation" of objects. Gamma and his colleagues [Gam95] note that creational patterns "encapsulate knowledge about which concrete classes the system uses" but at the same time "hide how instances of these classes are created and put together." Creational patterns provide mechanisms that make the instantiation of objects

---

<sup>3</sup> Gamma and his colleagues [Gam95] are often referred to as the "Gang of Four" (GoF) in patterns literature.

easier within a system and enforce “constraints on the type and number of objects that can be created within a system” [Maa07].

*Structural patterns* focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure. In essence, they help to establish relationships between entities within a system. For example, structural patterns that focus on class-oriented issues might provide inheritance mechanisms that lead to more effective program interfaces. Structural patterns that focus on objects suggest techniques for combining objects within other objects or integrating objects into a larger structure.

*Behavioral patterns* address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects.

## INFO



### Creational, Structural, and Behavioral Patterns

A wide variety of design patterns that fit into creational, structural, and behavioral categories have been proposed and can be found on the Web. Wikipedia ([www.wikipedia.org](http://www.wikipedia.org)) notes the following sampling:

#### Creational Patterns

- **Abstract factory pattern:** centralize decision of what factory to instantiate.
- **Factory method pattern:** centralize creation of an object of a specific type choosing one of several implementations.
- **Builder pattern:** separate the construction of a complex object from its representation so that the same construction process can create different representations.
- **Prototype pattern:** used when the inherent cost of creating a new object in the standard way (e.g., using the “new” keyword) is prohibitively expensive for a given application.
- **Singleton pattern:** restrict instantiation of a class to one object.

#### Structural Patterns

- **Adapter pattern:** “adapts” one interface for a class into one that a client expects.
- **Aggregate pattern:** a version of the composite pattern with methods for aggregation of children.
- **Bridge pattern:** decouple an abstraction from its implementation so that the two can vary independently.
- **Composite pattern:** a tree structure of objects where every object has the same interface.

- **Container pattern:** create objects for the sole purpose of holding other objects and managing them.
- **Proxy pattern:** a class functioning as an interface to another thing.
- **Pipes and filters:** a chain of processes where the output of each process is the input of the next.

#### Behavioral Patterns

- **Chain of responsibility pattern:** Command objects are handled or passed on to other objects by logic-containing processing objects.
- **Command pattern:** Command objects encapsulate an action and its parameters.
- **Event listener:** Data are distributed to objects that are registered to receive them.
- **Interpreter pattern:** Implement a specialized computer language to rapidly solve a specific set of problems.
- **Iterator pattern:** Iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Mediator pattern:** Provides a unified interface to a set of interfaces in a subsystem.
- **Visitor pattern:** A way to separate an algorithm from an object.
- **Single-serving visitor pattern:** Optimize the implementation of a visitor that is allocated, used only once, and then deleted.
- **Hierarchical visitor pattern:** Provide a way to visit every node in a hierarchical data structure such as a tree.

Comprehensive descriptions of each of these patterns can be obtained via links at [www.wikipedia.org](http://www.wikipedia.org).



A *framework* is a reusable “mini-architecture” that serves as a foundation from which other design patterns can be applied.

### 12.1.2 Frameworks

Patterns themselves may not be sufficient to develop a complete design. In some cases it may be necessary to provide an implementation-specific skeletal infrastructure, called a *framework*, for design work. That is, you can select a “*Reusable mini-architecture* that provides the generic structure and behavior for a family of software abstractions, along with a context . . . which specifies their collaboration and use within a given domain” [Amb98].

A framework is not an architectural pattern, but rather a skeleton with a collection of “plug points” (also called *hooks* and *slots*) that enable it to be adapted to a specific problem domain. The plug points enable you to integrate problem-specific classes or functionality within the skeleton. In an object-oriented context, a framework is a collection of cooperating classes.

Gamma and his colleagues [Gam95] describe the differences between design patterns and frameworks in the following manner:

1. *Design patterns are more abstract than frameworks.* Frameworks can be embodied in code, but only examples of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly. . . .
2. *Design patterns are smaller architectural elements than frameworks.* A typical framework contains several design patterns but the reverse is never true.
3. *Design patterns are less specialized than frameworks.* Frameworks always have a particular application domain. In contrast, design patterns can be used in nearly any kind of application. While more specialized design patterns are certainly possible, even these wouldn’t dictate an application architecture.

In essence, the designer of a framework will argue that one reusable mini-architecture is applicable to all software to be developed within a limited domain of application. To be most effective, frameworks are applied with no changes. Additional design elements may be added, but only via the plug points that allow the designer to flesh out the framework skeleton.

### 12.1.3 Describing a Pattern

Pattern-based design begins with the recognition of patterns within the application you intend to build, continues with a search to determine whether others have addressed the pattern, and concludes with the application of an appropriate pattern to the problem at hand. The second of these three tasks is often the most difficult. How do you find patterns that fit your needs?

An answer to this question must rely on effective communication of the problem the pattern addresses, the context in which the pattern resides, the system of forces that mold the context, and the solution that is proposed. To communicate this information unambiguously, a standard form or template for pattern descriptions is required. Although a number of different pattern templates have been proposed,

almost all contain a major subset of the content suggested by Gamma and his colleagues [Gam95]. A simplified pattern template is shown in the sidebar.



### Design Pattern Template

**Pattern name**—describes the essence of the pattern in a short but expressive name  
**Problem**—describes the problem that the pattern addresses  
**Motivation**—provides an example of the problem  
**Context**—describes the environment in which the problem resides including the application domain  
**Forces**—lists the system of forces that affect the manner in which the problem must be solved; includes a discussion of limitations and constraints that must be considered  
**Solution**—provides a detailed description of the solution proposed for the problem

### INFO

**Intent**—describes the pattern and what it does  
**Collaborations**—describes how other patterns contribute to the solution  
**Consequences**—describes the potential trade-offs that must be considered when the pattern is implemented and the consequences of using the pattern  
**Implementation**—identifies special issues that should be considered when implementing the pattern  
**Known uses**—provides examples of actual uses of the design pattern in real applications  
**Related patterns**—cross-references related design patterns

### vote:

"Patterns are half-baked—meaning you always have to finish them yourself and adapt them to your own environment."

**Martin Fowler**

The names of design patterns should be chosen with care. One of the key technical problems in pattern-based design is the inability to find existing patterns when hundreds or thousands of candidate patterns exist. The search for the "right" pattern is aided immeasurably by a meaningful pattern name.

A pattern template provides a standardized means for describing a design pattern. Each of the template entries represents characteristics of the design pattern that can be searched (e.g., via a database) so that the appropriate pattern can be found.

#### 12.1.4 Pattern Languages and Repositories

When we use the term *language*, the first thing that comes to mind is either a natural language (e.g., English, Spanish, Chinese) or a programming language (e.g., C++, Java). In both cases the language has a syntax and semantics that are used to communicate ideas or procedural instructions in an effective manner.

When the term *language* is used in the context of design patterns, it takes on a slightly different meaning. A *pattern language* encompasses a collection of patterns, each described using a standardized template (Section 12.1.3) and interrelated to show how these patterns collaborate to solve problems across an application domain.<sup>4</sup>

In a natural language, words are organized into sentences that impart meaning. The structure of sentences is described by the language's syntax. In a pattern language, design patterns are organized in a way that provides a "structured method of describing good design practices within a particular domain."<sup>5</sup>

<sup>4</sup> Christopher Alexander originally proposed pattern languages for building architecture and urban planning. Today, pattern languages have been developed for everything from the social sciences to the software engineering process.

<sup>5</sup> This Wikipedia description can be found at [http://en.wikipedia.org/wiki/Pattern\\_language](http://en.wikipedia.org/wiki/Pattern_language).



If you can't find a pattern language that addresses your problem domain, look for analogies in another set of patterns.

### WebRef

For a listing of useful patterns languages see [c2.com/ppr/titles.html](http://c2.com/ppr/titles.html). Additional information can be obtained at [hillside.net/patterns/](http://hillside.net/patterns/).

In a way, a pattern language is analogous to a hypertext instruction manual for problem solving in a specific application domain. The problem domain under consideration is first described hierarchically, beginning with broad design problems associated with the domain and then refining each of the broad problems into lower levels of abstraction. In a software context, broad design problems tend to be architectural in nature and address the overall structure of the application and the data or content that serve it. Architectural problems are refined to lower levels of abstraction, leading to design patterns that solve subproblems and collaborate with one another at the component (or class) level. Rather than a sequential list of patterns, a pattern language represents an interconnected collection in which the user can begin with a broad design problem and "burrow down" to uncover specific problems and their solutions.

Dozens of pattern languages have been proposed for software design [Hil08]. In most cases, the design patterns that are part of pattern language are stored in a Web-accessible patterns repository (e.g., [Boo08], [Cha03], [HPR02]). The repository provides an index of all design patterns and contains hypermedia links that enable the user to understand the collaborations between patterns.

## 12.2 PATTERN-BASED SOFTWARE DESIGN

The best designers in any field have an uncanny ability to see patterns that characterize a problem and corresponding patterns that can be combined to create a solution. The software developers at Microsoft [Mic04] discuss this when they write:

While pattern-based design is relatively new in the field of software development, industrial technology has used pattern-based design for decades, perhaps even centuries. Catalogs of mechanisms and standard configurations provide design elements that are used to engineer automobiles, aircraft, machine tools, and robots. Applying pattern-based design to software development promises the same benefits to software as it does to industrial technology: predictability, risk mitigation, and increased productivity.

Throughout the design process, you should look for every opportunity to apply existing design patterns (when they meet the needs of the design) rather than creating new ones.

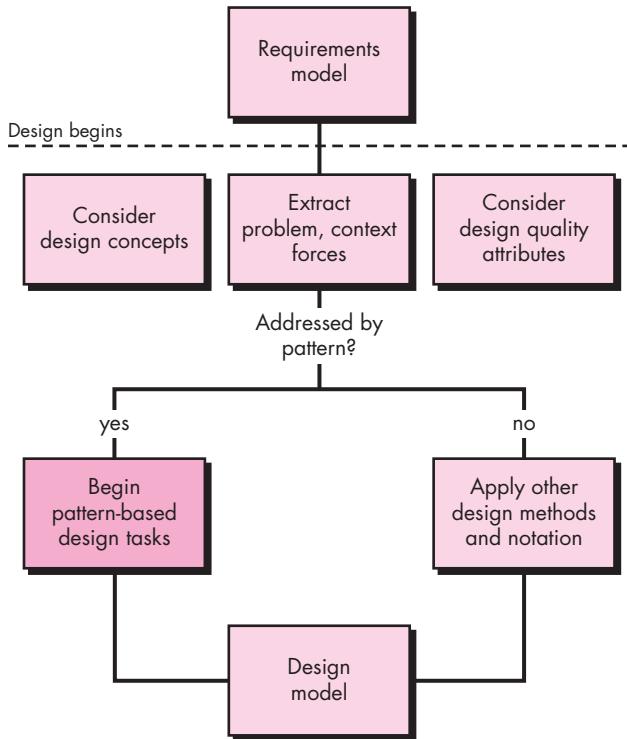
### 12.2.1 Pattern-Based Design in Context

Pattern-based design is not used in a vacuum. The concepts and techniques discussed for architectural, component-level, and user interface design (Chapters 9 through 11) are all used in conjunction with a pattern-based approach.

In Chapter 8, I noted that a set of quality guidelines and attributes serve as the basis for all software design decisions. The decisions themselves are influenced by a set of fundamental design concepts (e.g., separation of concerns, stepwise refinement, functional independence) that are achieved using heuristics that have evolved over many decades, and best practices (e.g., techniques, modeling notation) that

**FIGURE 12.1**

**Pattern-based design in context**



have been proposed to make design easier to perform and more effective as a basis for construction.

The role of pattern-based design in all of this is illustrated in Figure 12.1. A software designer begins with a requirements model (either explicit or implied) that presents an abstract representation of the system. The requirements model describes the problem set, establishes the context, and identifies the system of forces that hold sway. It may imply the design in an abstract manner, but the requirements model does little to represent the design explicitly.

As you begin your work as a designer, it's always important to keep quality attributes in mind. These attributes (e.g., a design must implement all explicit requirements addressed in the requirements model) establish a way to assess software quality but do little to help you actually achieve it. The design you create should exhibit the fundamental design concepts discussed in Chapter 8. Therefore, you should apply proven techniques for translating the abstractions contained in the requirements model into a more concrete form that is the software design. To accomplish this, you'll use the methods and modeling tools available for architectural, component-level, and interface design. But only when you're faced with a problem, context, and system of forces that have not been solved before. If a solution already exists, use it! And that means applying a pattern-based design approach.

### 12.2.2 Thinking in Patterns

In an excellent book on pattern-based design, Shalloway and Trott [Sha05] comment on a “new way of thinking” when one uses patterns as part of the design activity:

I had to open my mind to a new way of thinking. And when I did so, I heard [Christopher] Alexander say that “good software design cannot be achieved simply by adding together performing parts.”

Good design begins by considering context—the big picture. As context is evaluated, you extract a hierarchy of problems that must be solved. Some of these problems will be global in nature, while others will address specific features and functions of the software. All will be affected by a system of forces that will influence the nature of the solution that is proposed.

Shalloway and Trott [Sha05] suggest the following approach<sup>6</sup> that enables a designer to think in patterns:



1. Be sure you understand the big picture—the context in which the software to be built resides. The requirements model should communicate this to you.
2. Examining the big picture, extract the patterns that are present at that level of abstraction.
3. Begin your design with “big picture” patterns that establish a context or skeleton for further design work.
4. “Work inward from the context” [Sha05] looking for patterns at lower levels of abstraction that contribute to the design solution.
5. Repeat steps 1 to 4 until the complete design is fleshed out.
6. Refine the design by adapting each pattern to the specifics of the software you’re trying to build.

It’s important to note that patterns are not independent entities. Design patterns that are present at a high level of abstraction will invariably influence the manner in which other patterns are applied at lower levels of abstraction. In addition, patterns often collaborate with one another. The implication—when you select an architectural pattern, it may very well influence the component-level design patterns you choose. Likewise, when you select a specific interface design pattern, you are sometimes forced to use other patterns that collaborate with it.

To illustrate, consider the **SafeHomeAssured.com** WebApp. If you consider the big picture, the WebApp must address a number of fundamental problems such as:

- How to provide information about *SafeHome* products and services
- How to sell *SafeHome* products and services to customers
- How to establish Internet-based monitoring and control of an installed security system

<sup>6</sup> Based on the work of Christopher Alexander [Ale79].

Each of these fundamental problems can be further refined into a set of subproblems. For example *How to sell via the Internet* implies an **E-commerce** pattern that itself implies a large number of patterns at lower levels of abstraction. The **E-commerce** pattern (likely, an architectural pattern) implies mechanisms for setting up a customer account, displaying the products to be sold, selecting products for purchase, and so forth. Hence, if you think in patterns, it is important to determine whether a pattern for setting up an account exists. If **SetUpAccount** is available as a viable pattern for the problem context, it may collaborate with other patterns such as **BuildInputForm**, **ManageFormsInput**, and **ValidateFormsEntry**. Each of these patterns delineates problems to be solved and solutions that may be applied.

### 12.2.3 Design Tasks

The following design tasks are applied when a pattern-based design philosophy is used:



- 1. Examine the requirements model and develop a problem hierarchy.** Describe each problem and subproblem by isolating the problem, the context, and the system of forces that apply. Work from broad problems (high level of abstraction) to smaller subproblems (at lower levels of abstraction).
- 2. Determine if a reliable pattern language has been developed for the problem domain.** As I noted in Section 12.1.4, a pattern language addresses problems associated with a specific application domain. The *SafeHome* software team would look for a pattern language developed specifically for home security products. If that level of pattern language specificity could not be found, the team would partition the *SafeHome* software problem into a series of generic problem domains (e.g., digital device monitoring problems, user interface problems, digital video management problems) and search for appropriate pattern languages.
- 3. Beginning with a broad problem, determine whether one or more architectural patterns is available for it.** If an architectural pattern is available, be certain to examine all collaborating patterns. If the pattern is appropriate, adapt the design solution proposed and build a design model element that adequately represents it. As I noted in Section 12.2.2, a broad problem for the **SafeHomeAssured.com** WebApp is addressed with an **E-commerce** pattern. This pattern will suggest a specific architecture for addressing e-commerce requirements.
- 4. Using the collaborations provided for the architectural pattern, examine subsystem or component-level problems and search for appropriate patterns to address them.** It may be necessary to search through other pattern repositories as well as the list of patterns that corresponds to the architectural solution. If an appropriate pattern is found, adapt

the design solution proposed and build a design model element that adequately represents it. Be certain to apply step 7.

- 5. Repeat steps 2 through 5 until all broad problems have been addressed.** The implication is to begin with the big picture and elaborate to solve problems at increasingly more detailed levels.
- 6. If user interface design problems have been isolated (this is almost always the case), search the many user interface design pattern repositories for appropriate patterns.** Proceed in a manner similar to steps 3, 4, and 5.
- 7. Regardless of its level of abstraction, if a pattern language and/or patterns repository or individual pattern shows promise, compare the problem to be solved against the existing pattern(s) presented.** Be certain to examine context and forces to ensure that the pattern does, in fact, provide a solution that is amenable to the problem.
- 8. Be certain to refine the design as it is derived from patterns using design quality criteria as a guide.**

Although this design approach is top-down, real-life design solutions are sometimes more complex. Gillis [Gil06] comments on this when he writes:

Design patterns in software engineering are meant to be used in a deductive, rationalistic fashion. So you have this general problem or requirement, X, design pattern Y solves X, therefore use Y. Now, when I reflect on my own process—and I've got reason to believe that I'm not alone here—I find that it's more organic than that, more inductive than deductive, more bottom-up than top-down.

Obviously, there's a balance to be achieved. When a project is in the initial bootstrap phase and I'm trying to make the jump from abstract requirements to a concrete design solution, I'll often perform a sort of breadth-first search... I've found design patterns to be helpful, allowing me to quickly frame up the design problem in concrete terms.

In addition, the pattern-based approach must be used in conjunction with other software design concepts and techniques.

#### 12.2.4 Building a Pattern-Organizing Table



Entries in the table can be supplemented with an indication of the relative applicability of the pattern.

As pattern-based design proceeds, you may encounter trouble organizing and categorizing candidate patterns from multiple pattern languages and repositories. To help organize your evaluation of candidate patterns, Microsoft [Mic04] suggests the creation of a *pattern-organizing table* that takes the general form shown in Figure 12.2.

A pattern-organizing table can be implemented as a spreadsheet model using the form shown in the figure. An abbreviated list of problem statements, organized by data/content, architecture, component-level, and user interface issues, is presented in the left-hand (shaded) column. Four pattern types—database, application,

**FIGURE 12.2**

**A pattern-organizing table**

Source: Adapted from [Mic04].

	Database	Application	Implementation	Infrastructure
<b>Data/Content</b>				
Problem statement ...	PatternName(s)		PatternName(s)	
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...	PatternName(s)			PatternName(s)
<b>Architecture</b>				
Problem statement ...		PatternName(s)		
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...				
<b>Component-level</b>				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...				PatternName(s)
Problem statement ...		PatternName(s)	PatternName(s)	
<b>User interface</b>				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	

implementation, and infrastructure—are listed across the top row. The names of candidate patterns are noted in the cells of the table.

To provide entries for the organizing table, you'll search through pattern languages and repositories for patterns that address a particular problem statement. When one or more candidate patterns is found, it is entered in the row corresponding to the problem statement and the column that corresponds to the pattern type. The name of the pattern is entered as a hyperlink to the URL of the Web address that contains a complete description of the pattern.

### 12.2.5 Common Design Mistakes

Pattern-based design can make you a better software designer, but it is *not* a panacea. Like all design methods, you must begin with first principles, emphasizing software quality fundamentals and ensuring that the design does, in fact, address the needs expressed by the requirements model.



*Don't force a pattern, even if it addresses the problem at hand. If the context and forces are wrong, look for another pattern.*

A number of common mistakes occur when pattern-based design is used. In some cases, not enough time has been spent to understand the underlying problem and its context and forces, and as a consequence, you select a pattern that looks right but is inappropriate for the solution required. Once the wrong pattern is selected, you refuse to see your error and force-fit the pattern. In other cases, the problem has forces that are not considered by the pattern you've chosen, resulting in a poor or

erroneous fit. Sometimes a pattern is applied too literally and the required adaptations for your problem space are not implemented.

Can these mistakes be avoided? In most cases the answer is “yes.” Every good designer looks for a second opinion and welcomes review of her work. The review techniques discussed in Chapter 15 can help to ensure that the pattern-based design you’ve developed will result in a high-quality solution for the software problem to be solved.

### 12.3 ARCHITECTURAL PATTERNS

#### KEY POINT

A software architecture may have a number of architectural patterns that address issues such as concurrency, persistence, and distribution.

**?** What are some typical architectural pattern domains?

If a house builder decides to construct a center-hall colonial, there is a single architectural style that can be applied. The details of the style (e.g., number of fireplaces, façade of the house, placement of doors and windows) can vary considerably, but once the decision on the overall architecture of the house is made, the style is imposed on the design.<sup>7</sup>

Architectural patterns are a bit different. For example, every house (and every architectural style for houses) employs a **Kitchen** pattern. The **Kitchen** pattern and patterns it collaborates with address problems associated with the storage and preparation of food, the tools required to accomplish these tasks, and rules for placement of these tools relative to workflow in the room. In addition, the pattern might address problems associated with countertops, lighting, wall switches, a central island, flooring, and so on. Obviously, there is more than a single design for a kitchen, often dictated by the context and system of forces. But every design can be conceived within the context of the “solution” suggested by the **Kitchen** pattern.

As I have already noted, architectural patterns for software define a specific approach for handling some characteristic of the system. Bosch [Bos00] and Booch [Boo08] define a number of architectural pattern domains. Representative examples are provided in the paragraphs that follow:

**Access control.** There are many situations in which access to data, features, and functionality delivered by an application is limited to specifically defined end users. From an architectural point of view, access to some portion of the software architecture must be controlled rigorously.

**Concurrency.** Many applications must handle multiple tasks in a manner that simulates parallelism (i.e., this occurs whenever multiple “parallel” tasks or components are managed by a single processor). There are a number of different ways in which an application can handle concurrency, and each can be presented by a different architectural pattern. For example, one approach is to use an **OperatingSystem-ProcessManagement** pattern that provides built-in OS features that allow

<sup>7</sup> This implies that there will be a central foyer and hallway, that rooms will be placed to the left and right of the foyer, that the house will have two (or more) stories, that the bedrooms of the house will be upstairs, and so on. These “rules” are imposed once the decision is made to use the center-hall colonial style.

components to execute concurrently. The pattern also incorporates OS functionality that manages communication between processes, scheduling, and other capabilities required to achieve concurrency. Another approach might be to define a task scheduler at the application level. A **TaskScheduler** pattern contains a set of active objects that each contains a *tick()* operation [Bos00]. The scheduler periodically invokes *tick()* for each object, which then performs the functions it must perform before returning control back to the scheduler which then invokes the *tick()* operation for the next concurrent object.

**Distribution.** The distribution problem addresses the manner in which systems or components within systems communicate with one another in a distributed environment. Two subproblems are considered: (1) the way in which entities connect to one another, and (2) the nature of the communication that occurs. The most common architectural pattern established to address the distribution problem is the **Broker** pattern. A broker acts as a “middleman” between the client component and a server component. The client sends a message to the broker (containing all appropriate information for the communication to be effected) and the broker completes the connection.

**Persistence.** Data persists if it survives past the execution of the process that created it. Persistent data are stored in a database or file and may be read or modified by other processes at a later time. In object-oriented environments, the idea of a persistent object extends the persistence concept a bit further. The values of all of the object’s attributes, the general state of the object, and other supplementary information are stored for future retrieval and use. In general, two architectural patterns are used to achieve persistence—a **DatabaseManagementSystem** pattern that applies the storage and retrieval capability of a DBMS to the application architecture or an **Application Level-Persistence** pattern that builds persistence features into the application architecture (e.g., word processing software that manages its own document structure).

Before any one of the representative architectural patterns noted in the preceding paragraphs can be chosen, it must be assessed for its appropriateness for the application and the overall architectural style, as well as the context and system of forces that it specifies.



### Design Pattern Repositories

There are many sources for design patterns available on the Web. Some patterns can be obtained from individually published pattern languages, while others are available as part of a patterns portal or patterns repository. The following Web sources are worth a look:

Hillside.net <http://hillside.net/patterns/>

One of the Web’s most comprehensive collections of patterns and pattern languages.

### INFO

Portland Pattern Repository

<http://c2.com/ppr/index.html>

Contains pointers to a wide variety of patterns resources and collections.

Pattern Index <http://c2.com/cgi/wiki?PatternIndex>

An “eclectic collection of patterns”

Booch’s Architecture Patterns Handbook

[www.booch.com/architecture/index.jsp](http://www.booch.com/architecture/index.jsp)

Bibliographic reference to hundreds of architectural and component design patterns

#### UI Patterns Collections

UI/HCI Patterns

[www.hcipatterns.org/patterns.html](http://www.hcipatterns.org/patterns.html)

Jennifer Tidwell's UI patterns

[www.time-tripper.com/uipatterns/](http://www.time-tripper.com/uipatterns/)

Mobile UI Design Patterns

<http://patterns.littlespringsdesign.com/wikka.php?wakka=Mobile>

#### Patterns

Pattern Language for UI Design

[www.maplefish.com/todd/papers/Experiences.html](http://www.maplefish.com/todd/papers/Experiences.html)

Interaction Design Library for Games

[www.eelke.com/research/usability.html](http://www.eelke.com/research/usability.html)

UI Design Patterns

[www.cs.helsinki.fi/u/salaakso/patterns/](http://www.cs.helsinki.fi/u/salaakso/patterns/)

#### Specialized Design Patterns:

Aircraft Avionics

<http://g.oswego.edu/dl/acs/acs.acs.html>

Business Information Systems

[www.objectarchitects.de/arcus/cookbook/](http://www.objectarchitects.de/arcus/cookbook/)

Distributed Processing

[www.cs.wustl.edu/~schmidt/](http://www.cs.wustl.edu/~schmidt/)

IBM Patterns for e-Business [www128.ibm.com/developerworks/patterns/](http://www128.ibm.com/developerworks/patterns/)

Yahoo! Design Pattern Library

<http://developer.yahoo.com/ypatterns/>

WebPatterns.org <http://webpatterns.org/>

## 12.4 COMPONENT-LEVEL DESIGN PATTERNS

Component-level design patterns provide you with proven solutions that address one or more subproblems extracted from the requirements model. In many cases, design patterns of this type focus on some functional element of a system. For example, the **SafeHomeAssured.com** application must address the following design subproblem: *How can we get product specifications and related information for any SafeHome device?*

Having enunciated the subproblem that must be solved, you should now consider context and the system of forces that affect the solution. Examining the appropriate requirements model use case, you learn that the consumer uses the specification for a *SafeHome* device (e.g., a security sensor or camera) for informational purposes. However, other information that is related to the specification (e.g., pricing) may be used when e-commerce functionality is selected.

The solution to the subproblem involves a search. Since searching is a very common problem, it should come as no surprise that there are many search-related patterns. Looking through a number of patterns repositories, you find the following patterns, along with the problem that each solves:

**AdvancedSearch.** Users must find a specific item in a large collection of items.

**HelpWizard.** Users need help on a certain topic related to the website or when they need to find a specific page within the site.

**SearchArea.** Users must find a page.

**SearchTips.** Users need to know how to control the search engine.

**SearchResults.** Users have to process a list of search results.

**SearchBox.** Users have to find an item or specific information.

For **SafeHomeAssured.com** the number of products is not particularly large, and each has a relatively simple categorization, so **AdvancedSearch** and **HelpWizard** are probably not necessary. Similarly, the search is simple enough not to require **SearchTips**. The description of **SearchBox**, however, is given (in part) as:

---

### Search Box

(Adapted from [www.welie.com/patterns/showPattern.php?patternID=search.](http://www.welie.com/patterns/showPattern.php?patternID=search.))

**Problem:** The users need to find an item or specific information.

**Motivation:** Any situation in which a keyword search is applied across a collection of content objects organized as Web pages.

**Context:** Rather than using navigation to acquire information or content, the user wants to do a direct search through content contained on multiple Web pages. Any website that already has primary navigation. User may want to search for an item in a category. User might want to further specify a query.

**Forces:** The website already has primary navigation. Users may want to search for an item in a category. Users might want to further specify a query using simple Boolean operators.

**Solution:** Offer search functionality consisting of a search label, a keyword field, a filter if applicable, and a “go” button. Pressing the return key has the same function as selecting the go button. Also provide Search Tips and examples in a separate page. A link to that page is placed next to the search functionality. The edit box for the search term is large enough to accommodate three typical user queries (typically around 20 characters). If the number of filters is more than 2, use a combo box for filters selection, otherwise a radio button.

The search results are presented on a new page with a clear label containing at least “Search results” or similar. The search function is repeated in the top part of the page with the entered keywords, so that the users know what the keywords were.

---

*The pattern description continues with other entries as described in Section 12.1.3.*

---

The pattern goes on to describe how the search results are accessed, presented, matched, and so on. Based on this, the **SafeHomeAssured.com** team can design the components required to implement the search or (more likely) acquire existing reusable components.

## SAFEHOME



### Applying Patterns

**The scene:** Informal discussion during the design of a software increment that implements sensor control via the Internet for **SafeHomeAssured.com**.

**The players:** Jamie (responsible for design) and Vinod (**SafeHomeAssured.com** chief system architect).

#### The conversation:

**Vinod:** So how is the design of the camera control interface coming along?

**Jamie:** Not too bad. I've designed most of the capability to connect to the actual sensors without too many problems. I've also started thinking about the interface for the users to actually move, pan, and zoom the cameras from a remote Web page, but I'm not sure I've got it right yet.

**Vinod:** What have you come up with?

**Jamie:** Well, the requirements are that the camera control needs to be highly interactive—as the user moves the control, the camera should move as soon as possible. So, I was thinking of having a set of buttons laid out like a normal camera, but when the user clicks them, it controls the camera.

**Vinod:** Hmm. Yeah, that would work, but I'm not sure it's right—for each click of a control you need to wait for the whole client-server communication to occur, and so you won't get a good sense of quick feedback.

**Jamie:** That's what I thought—and why I wasn't very happy with the approach, but I'm not sure how else I might do it.

**Vinod:** Well, why not just use the **InteractiveDeviceControl** pattern!

**Jamie:** Uhmmm—what's that? I haven't heard of it?

**Vinod:** It's basically a pattern for exactly the problem you are describing. The solution it proposes is basically to create a control connection to the server with the device, through which control commands can be sent. That way you don't need to send normal HTTP requests. And the pattern even shows how you can implement this using some simple AJAX techniques. You have some simple client-side JavaScript that communicates directly with the server and sends the commands as soon as the user does anything.

**Jamie:** Cool! That's just what I needed to solve this thing. Where do I find it?

**Vinod:** It's available in an online repository. Here's the URL.

**Jamie:** I'll go check it out.

**Vinod:** Yep—but remember to check the consequences field for the pattern. I seem to remember that there was something in there about needing to be careful about issues of security. I think it might be because you are creating a separate control channel and so bypassing the normal Web security mechanisms.

**Jamie:** Good point. I probably wouldn't have thought of that! Thanks.

## 12.5 USER INTERFACE DESIGN PATTERNS

Hundreds of user interface (UI) patterns have been proposed in recent years. Most fall within one of the following 10 categories of patterns (discussed with a representative example<sup>8</sup>) as described by Tidwell [Tid02] and vanWelie [Wel01]:

**Whole UI.** Provide design guidance for top-level structure and navigation throughout the entire interface.

### Pattern: TopLevelNavigation

**Brief description:** Used when a site or application implements a number of major functions. Provides a top-level menu, often coupled with a logo or

<sup>8</sup> An abbreviated pattern template is used here. Full pattern descriptions (along with dozens of other patterns) can be found at [Tid02] and [Wel01].

identifying graphic, that enables direct navigation to any of the system's major functions.

**Details:** Major functions (generally limited to between four and seven function names) are listed across the top of the display (vertical column formats are also possible) in a horizontal line of text. Each name provides a link to the appropriate function or information source. Often used with the **Breadcrumbs** pattern discussed later.

**Navigation elements:** Each function/content name represents a link to the appropriate function or content.

**Page layout.** Address the general organization of pages (for websites) or distinct screen displays (for interactive applications).

#### **Pattern: CardStack**

**Brief description:** Used when a number of specific subfunctions or content categories related to a feature or function must be selected in random order. Provides the appearance of a stack of tabbed cards, each selectable with a mouse click and each representing specific subfunctions or content categories.

**Details:** Tabbed cards are a well-understood metaphor and are easy for the user to manipulate. Each tabbed card (divider) may have a slightly different format. Some may require input and have buttons or other navigation mechanisms; others may be informational. May be combined with other patterns such as **DropDownList**, **Fill-in-the-Blanks**, and others.

**Navigation elements:** A mouse click on a tab causes the appropriate card to appear. Navigation features within the card may also be present, but in general, these should initiate a function that is related to card data, not cause an actual link to some other display.

**Forms and input.** Consider a variety of design techniques for completing form-level input.

#### **Pattern: Fill-in-the-Blanks**

**Brief description:** Allow alphanumeric data to be entered in a "text box."

**Details:** Data may be entered within a text box. In general, the data are validated and processed after some text or graphic indicator (e.g., a button containing "go," "submit," "next") is picked. In many cases this pattern can be combined with drop-down list or other patterns (e.g., SEARCH *<drop down list>* FOR *<fill-in-the-blanks text box>*).

**Navigation elements:** A text or graphic indicator that initiates validation and processing.

**Tables.** Provide design guidance for creating and manipulating tabular data of all kinds.

**Pattern: SortableTable**

**Brief description:** Display a long list of records that can be sorted by selecting a toggle mechanism for any column label.

**Details:** Each row in the table represents a complete record. Each column represents one field in the record. Each column header is actually a selectable button that can be toggled to initiate an ascending or descending sort on the field associated with the column for all records displayed. The table is generally resizable and may have a scrolling mechanism if the number of records is larger than available window space.

**Navigation elements:** Each column header initiates a sort on all records. No other navigation is provided, although in some cases, each record may itself contain navigation links to other content or functionality.

**Direct data manipulation.** Address data editing, modification, and transformation.

**Pattern: BreadCrumbs**

**Brief description:** Provides a full navigation path when the user is working with a complex hierarchy of pages or display screens.

**Details:** Each page or display screen is given a unique identifier. The navigation path to the current location is specified in a predefined location for every display. The path takes the form: **home>major topic page>subtopic page>specific page>current page**.

**Navigation elements:** Any of the entries within the bread crumbs display can be used as a pointer to link back to a higher level of the hierarchy.

**Navigation.** Assist the user in navigating through hierarchical menus, Web pages, and interactive display screens.

**Pattern: EditInPlace**

**Brief description:** Provide simple text editing capability for certain types of content in the location that it is displayed. No need for the user to enter a text editing function or mode explicitly.

**Details:** The user sees content on the display that must be changed. A mouse double click on the content indicates to the system that editing is desired. The content is highlighted to signify that editing mode is available and the user makes appropriate changes.

**Navigation elements:** None.

**Searching.** Enable content-specific searches through information maintained within a website or contained by persistent data stores that are accessible via an interactive application.

**Pattern: SimpleSearch**

**Brief description:** Provides the ability to search a website or persistent data source for a simple data item described by an alphanumeric string.

**Details:** Provides the ability to search either locally (one page or one file) or globally (entire site or complete database) for the search string. Generates a list of “hits” in order of their probability of meeting the user’s needs. Does not provide multiple item searches or special Boolean operations (see *advanced search pattern*).

**Navigation elements:** Each entry in the list of hits represents a navigation link to the data referenced by the entry.

**Page elements.** Implement specific elements of a Web page or display screen.

### **Pattern: Wizard**

**Brief description:** Takes the user through a complex task one step at a time, providing guidance for the completion of the task through a series of simple window displays.

**Details:** Classic example is a registration process that contains four steps. The wizard pattern generates a window for each step, requesting specific information from the user one step at a time.

**Navigation elements:** Forward and back navigation allows the user to revisit each step in the wizard process.

**E-commerce.** Specific to websites, these patterns implement recurring elements of e-commerce applications.

### **Pattern: ShoppingCart**

**Brief description:** Provides a list of items selected for purchase.

**Details:** Lists item, quantity, product code, availability (in stock, out of stock), price, delivery information, shipping costs, and other relevant purchase information. Also provides ability to edit (e.g., remove, change quantity).

**Navigation elements:** Contains ability to proceed with shopping or go to checkout.

**Miscellaneous.** Patterns that do not easily fit into one of the preceding categories. In some cases, these patterns are domain dependent or occur only for specific classes of users.

### **Pattern: ProgressIndicator**

**Brief description:** Provides an indication of progress when an operation takes longer than  $n$  seconds.

**Details:** Represented as an animated icon or a message box that contains some visual indication (e.g., a rotating “barber pole,” a slider with a percent complete indicator) that processing is under way. May also contain a text content indication of the status of processing.

**Navigation elements:** Often contains a button that allows the user to pause or cancel processing.

Each of the preceding example patterns (and all patterns within each category) would also have a complete component-level design, including design classes, attributes, operations, and interfaces.

A comprehensive discussion of user interface patterns is beyond the scope of this book. If you have further interest, see [Duy02], [Bor01], [Tid02], and [Wel01] for further information.

## 12.6 WEBAPP DESIGN PATTERNS

Throughout this chapter you've learned that there are different types of patterns and many different ways they can be categorized. When you consider the design problems that must be solved when a WebApp is to be built, it's worth considering pattern categories by focusing on two dimensions: the design focus of the pattern and its level of granularity. *Design focus* identifies which aspect of the design model is relevant (e.g., information architecture, navigation, interaction). *Granularity* identifies the level of abstraction that is being considered (e.g., does the pattern apply to the entire WebApp, to a single Web page, to a subsystem, or an individual WebApp component?).

### 12.6.1 Design Focus



Your focus becomes “narrower” the further you move into design.

In earlier chapters I emphasized a design progression that begins by considering architecture, component-level issues, and user interface representations. At each step, the problems you consider and the solutions you propose begin at a high level of abstraction and slowly become more detailed and specific. Stated another way, design focus becomes “narrower” as you move further into design. The problems (and solutions) you will encounter when designing an information architecture for a WebApp are different from the problems (and solutions) that are encountered when performing interface design. Therefore, it should come as no surprise that patterns for WebApp design can be developed for different levels of design focus, so that you can address the unique problems (and related solutions) that are encountered at each level. WebApp patterns can be categorized using the following levels of design focus:

- **Information architecture patterns** relate to the overall structure of the information space, and the ways in which users will interact with the information.
- **Navigation patterns** define navigation link structures, such as hierarchies, rings, tours, and so on.
- **Interaction patterns** contribute to the design of the user interface. Patterns in this category address how the interface informs the user of the consequences of a specific action, how a user expands content based on usage

context and user desires, how to best describe the destination that is implied by a link, how to inform the user about the status of an ongoing interaction, and interface-related issues.

- **Presentation patterns** assist in the presentation of content as it is presented to the user via the interface. Patterns in this category address how to organize user interface control functions for better usability, how to show the relationship between an interface action and the content objects it affects, and how to establish effective content hierarchies.
- **Functional patterns** define the workflows, behaviors, processing, communication, and other algorithmic elements within a WebApp.

In most cases, it would be fruitless to explore the collection of information architecture patterns when a problem in interaction design is encountered. You would examine interaction patterns, because that is the design focus that is relevant to the work being performed.

### 12.6.2 Design Granularity

When a problem involves “big picture” issues, you should attempt to develop solutions (and use relevant patterns) that focus on the big picture. Conversely, when the focus is very narrow (e.g., uniquely selecting one item from a small set of five or fewer items), the solution (and the corresponding pattern) is targeted quite narrowly. In terms of the level of granularity, patterns can be described at the following levels:

- **Architectural patterns.** This level of abstraction will typically relate to patterns that define the overall structure of the WebApp, indicate the relationships among different components or increments, and define the rules for specifying relationships among the elements (pages, packages, components, subsystems) of the architecture.
- **Design patterns.** These address a specific element of the design such as an aggregation of components to solve some design problem, relationships among elements on a page, or the mechanisms for effecting component-to-component communication. An example might be the **Broadsheet** pattern for the layout of a WebApp home page.
- **Component patterns.** This level of abstraction relates to individual small-scale elements of a WebApp. Examples include individual interaction elements (e.g., radio buttons), navigation items (e.g., how might you format links?) or functional elements (e.g., specific algorithms).

It is also possible to define the relevance of different patterns to different classes of applications or domains. For example, a collection of patterns (at different levels of design focus and granularity) might be particularly relevant to e-business.

## INFO



### Hypermedia Design Patterns Repositories

The IAWiki website (<http://iawiki.net/> **WebsitePatterns**), a collaborative discussion space for information architects, contains many useful resources. Among them are links to a number of useful hypermedia patterns catalogs and repositories. Hundreds of design patterns are represented:

*Hypermedia Design Patterns Repository*

[www.designpattern.lu.unisi.ch/](http://www.designpattern.lu.unisi.ch/)

*InteractionPatterns by Tom Erickson*

[www.pliant.org/personal/Tom\\_Erickson/  
InteractionPatterns.html](http://www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html)

*Web Design Patterns by Martijn van Welie*

[www.welie.com/patterns/](http://www.welie.com/patterns/)

*Web Patterns for UI Design*

[http://harbinger.sims.berkeley.edu/  
ui\\_designpatterns/webpatterns2/  
webpatterns/home.php](http://harbinger.sims.berkeley.edu/ui_designpatterns/webpatterns2/webpatterns/home.php)

*Patterns for Personal Websites*

[www.rdrop.com/~half/Creations/Writings/Web.patterns/index.html](http://www.rdrop.com/~half/Creations/Writings/Web.patterns/index.html)

*Improving Web Information Systems with Navigational Patterns* <http://www8.org/w8-papers/5b-hypertext-media/improving/improving.html>

*An HTML 2.0 Pattern Language*

[www.anamorph.com/docs/patterns/  
default.html](http://www.anamorph.com/docs/patterns/default.html)

*Common Ground—A Pattern Language for HCI Design*

[www.mit.edu/~jtidwell/interaction\\_patterns.html](http://www.mit.edu/~jtidwell/interaction_patterns.html)

*Patterns for Personal Websites* [www.rdrop.com/~half/Creations/Writings/Web.patterns/index.html](http://www.rdrop.com/~half/Creations/Writings/Web.patterns/index.html)

*Indexing Pattern Language* [www.cs.brown.edu/~rms/InformationStructures/Indexing/Overview.html](http://www.cs.brown.edu/~rms/InformationStructures/Indexing/Overview.html)

## 12.7 SUMMARY

Design patterns provide a codified mechanism for describing problems and their solution in a way that allows the software engineering community to capture design knowledge for reuse. A pattern describes a problem, indicates the context enabling the user to understand the environment in which the problem resides, and lists a system of forces that indicate how the problem can be interpreted within its context and how the solution can be applied. In software engineering work, we identify and document generative patterns that describe an important and repeatable aspect of a system and then provide us with a way to build that aspect within a system of forces that is unique to a given context.

Architectural patterns describe broad-based design problems that are solved using a structural approach. Data patterns describe recurring data-oriented problems and the data modeling solutions that can be used to solve them. Component patterns (also referred to as design patterns) address problems associated with the development of subsystems and components, the manner in which they communicate with one another, and their placement within a larger architecture. Interface design patterns describe common user interface problems and their solution with a system of forces that includes the specific characteristics of end users. WebApp patterns address a problem set that is encountered when building WebApps and often incorporates many of the other patterns categories just mentioned. A framework

provides an infrastructure in which patterns may reside and idioms describe programming language-specific implementation detail for all or part of a specific algorithm or data structure. A standard form or template is used for pattern descriptions. A pattern language encompasses a collection of patterns, each described using a standardized template and interrelated to show how these patterns collaborate to solve problems across an application domain.

Pattern-based design is used in conjunction with architectural, component-level, and user interface design methods. The design approach begins with an examination of the requirements model to isolate problems, define context, and describe the system of forces. Next, pattern languages for the problem domain are searched to determine if patterns exist for the problems that have been isolated. Once appropriate patterns have been found, they are used as a design guide.

### PROBLEMS AND POINTS TO PONDER

**12.1.** Discuss the three “parts” of a design pattern and provide a concrete example of each from some field other than software.

**12.2.** What is the difference between a nongenerative and a generative pattern?

**12.3.** How do architectural patterns differ from component patterns?

**12.4.** What is a framework and how does it differ from a pattern? What is an idiom and how does it differ from a pattern?

**12.5.** Using the design pattern template presented in Section 12.1.3, develop a complete pattern description for a pattern suggested by your instructor.

**12.6.** Develop a skeletal pattern language for a sport with which you are familiar. You can begin by addressing the context, the system of forces, and the broad problems that a coach and team must solve. You need only specify pattern names and provide a one-sentence description for each pattern.

**12.7.** Find five patterns repositories and present an abbreviated description of the types of patterns contained in each.

**12.8.** When Christopher Alexander says “good design cannot be achieved simply by adding together performing parts,” what do you think he means?

**12.9.** Using the pattern-based design tasks noted in Section 12.2.3, develop a skeletal design for the “interior design system” described in Section 11.3.2.

**12.10.** Build a pattern-organizing table for the patterns you used in Problem 12.9.

**12.11.** Using the design pattern template presented in Section 12.1.3, develop a complete pattern description for the **Kitchen** pattern mentioned in Section 12.3.

**12.12.** The gang of four [Gam95] have proposed a variety of component patterns that are applicable to object-oriented systems. Select one (these are available on the Web) and discuss it.

**12.13.** Find three patterns repositories for user interface patterns. Select one pattern from each and present an abbreviated description of it.

**12.14.** Find three patterns repositories for WebApp patterns. Select one pattern from each and present an abbreviated description of it.

## FURTHER READING AND INFORMATION SOURCES

Over the past decade, many books on pattern-based design have been written for software engineers. Gamma and his colleagues [Gam95] have written the seminal book on the subject. More recent contributions include books by Lasater (*Design Patterns*, Wordware Publishing, Inc., 2007), Holzner (*Design Patterns for Dummies*, For Dummies, 2006), Freeman and her colleagues (*Head First Design Patterns*, O'Reilly Media, Inc., 2005), and Shalloway and Trott (*Design Patterns Explained*, 2d. ed., Addison-Wesley, 2004). A special issues of *IEEE Software* (July/August, 2007) discusses a wide variety of software patterns topics. Kent Beck (*Implementation Patterns*, Addison-Wesley, 2008) addresses patterns for coding and implementation issues that are encountered during the construction activity.

Other books focus on design patterns as they are supplied in specific application development and language environments. Contributions in this area include: Bowers (*Pro CSS and HTML Design Patterns*, Apress, 2007), Tropashko and Burleson (*SQL Design Patterns: Expert Guide to SQL Programming*, Rampant Techpress, 2007), Mahemoff (*Ajax Design Patterns*, O'Reilly Media, Inc., 2006), Metsker and Wake (*Design Patterns in Java*, Addison-Wesley, 2006), Nilsson (*Applying Domain-Driven Design and Patterns: With Examples in C# and .NET*, Addison-Wesley, 2006), Sweat (*PHPArchitect's Guide to PHP Design Patterns*, Marco Tabini & Associates, Inc., 2005), Metsker (*Design Patterns C#*, Addison-Wesley, 2004), Grand and Merrill (*Visual Basic .NET Design Patterns*, Wiley, 2003), Crawford and Kaplan (*J2EE Design Patterns*, O'Reilly Media, Inc., 2003), Juric et al. (*J2EE Design Patterns Applied*, Wrox Press, 2002), and Marinescu and Roman (*EJB Design Patterns*, Wiley, 2002).

Still other books address specific application domains. These include contributions by Kuchana (*Software Architecture Design Patterns in Java*, Auerbach, 2004), Joshi (*C++ Design Patterns and Derivatives Pricing*, Cambridge University Press, 2004), Douglass (*Real-Time Design Patterns*, Addison-Wesley, 2002), and Schmidt and Rising (*Design Patterns in Communication Software*, Cambridge University Press, 2001).

Classic books by the architect Christopher Alexander (*Notes on the Synthesis of Form*, Harvard University Press, 1964, and *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977) are worthwhile reading for a software designer who intends to fully understand design patterns.

A wide variety of information sources on pattern-based design are available on the Internet. An up-to-date list of World Wide Web references that are relevant to pattern-based design can be found at the SEPA website: [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm).

## WEBAPP DESIGN

13

**KEY CONCEPTS**

content	
architecture	...384
objects	.....382
design	
aesthetic	....380
architectural	...383
component-level	.....390
content	.....382
goals	.....377
graphic	.....381
navigation	....388

**QUICK LOOK**

**What is it?** Design for WebApps encompasses technical and nontechnical activities that include: establishing the look and feel of the WebApp, creating the aesthetic layout of the user interface, defining the overall architectural structure, developing the content and functionality that reside within the architecture, and planning the navigation that occurs within the WebApp.

**Who does it?** Web engineers, graphic designers, content developers, and other stakeholders all participate in the creation of a WebApp design model.

**Why is it important?** Design allows you to create a model that can be assessed for quality and improved before content and code are generated, tests are conducted, and end users become involved in large numbers. Design is the place where WebApp quality is established.

**What are the steps?** WebApp design encompasses six major steps that are driven by information obtained during requirements modeling. Content design uses the content model (developed during analysis) as the basis for establishing

the design of content objects. Aesthetic design (also called graphic design) establishes the look and feel that the end user sees. Architectural design focuses on the overall hypermedia structure of all content objects and functions. Interface design establishes the layout and interaction mechanisms that define the user interface. Navigation design defines how the end user navigates through the hypermedia structure, and component design represents the detailed internal structure of functional elements of the WebApp.

**What is the work product?** A design model that encompasses content, aesthetics, architecture, interface, navigation, and component-level design issues is the primary work product that is produced during WebApp design.

**How do I ensure that I've done it right?** Each element of the design model is reviewed in an effort to uncover errors, inconsistencies, or omissions. In addition, alternative solutions are considered, and the degree to which the current design model will lead to an effective implementation is also assessed.

<b>pyramid .....</b>	<b>378</b>
<b>quality .....</b>	<b>374</b>
<b>MVC</b>	
<b>architecture .....</b>	<b>386</b>
<b>OOHDM .....</b>	<b>390</b>
<b>WebApp</b>	
<b>architecture .....</b>	<b>386</b>

function are complex; when the size of the WebApp encompasses hundreds or thousands of content objects, functions, and analysis classes; and when the success of the WebApp will have a direct impact on the success of the business, design cannot and should not be taken lightly.

This reality leads us to Nielsen's second approach—"the engineering ideal of solving a problem for a customer." Web engineering<sup>1</sup> adopts this philosophy, and a more rigorous approach to WebApp design enables developers to achieve it.

### 13.1 WEBAPP DESIGN QUALITY

Design is the engineering activity that leads to a high-quality product. This leads us to a recurring question that is encountered in all engineering disciplines: What is quality? In this section I'll examine the answer within the context of WebApp development.

Every person who has surfed the Web or used a corporate Intranet has an opinion about what makes a "good" WebApp. Individual viewpoints vary widely. Some users enjoy flashy graphics; others want simple text. Some demand copious information; others desire an abbreviated presentation. Some like sophisticated analytical tools or database access; others like to keep it simple. In fact, the user's perception of "goodness" (and the resultant acceptance or rejection of the WebApp as a consequence) might be more important than any technical discussion of WebApp quality.

But how is WebApp quality perceived? What attributes must be exhibited to achieve goodness in the eyes of end users and at the same time exhibit the technical characteristics of quality that will enable you to correct, adapt, enhance, and support the application over the long term?

In reality, all of the technical characteristics of design quality discussed in Chapter 8 and the generic quality attributes presented in Chapter 14 apply to WebApps. However, the most relevant of these generic attributes—usability, functionality, reliability, efficiency, and maintainability—provide a useful basis for assessing the quality of Web-based systems.

Olsina and his colleagues [Ols99] have prepared a "quality requirement tree" that identifies a set of technical attributes—usability, functionality, reliability, efficiency, and maintainability—that lead to high-quality WebApps.<sup>2</sup> Figure 13.1 summarizes their work. The criteria noted in the figure are of particular interest if you design, build, and maintain WebApps over the long term.

 **vote:**

"If products are designed to better fit the natural tendencies of human behavior, then people will be more satisfied, more fulfilled, and more productive."

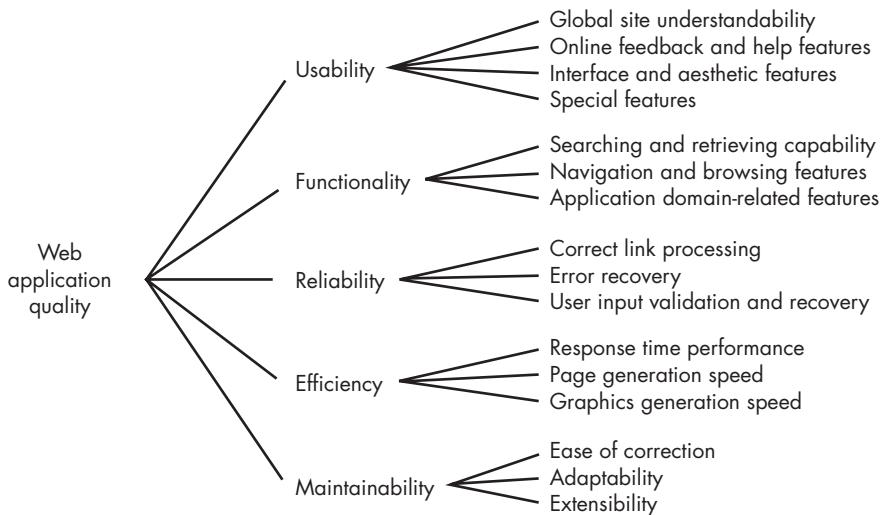
**Susan Weinschenk**

<sup>1</sup> *Web engineering* [Pre08] is an adapted version of the software engineering approach that is presented throughout this book. It proposes an agile, yet disciplined framework for building industry-quality Web-based systems and applications.

<sup>2</sup> These quality attributes are quite similar to those presented in Chapters 8 and 14. The implication: quality characteristics are universal for all software.

**FIGURE 13.1****Quality requirements tree.**

Source: [Ols99].



Offutt [Off02] extends the five major quality attributes noted in Figure 13.1 by adding the following attributes:

**What are the major attributes of quality for WebApps?**

**Security.** WebApps have become heavily integrated with critical corporate and government databases. E-commerce applications extract and then store sensitive customer information. For these and many other reasons, WebApp security is paramount in many situations. The key measure of security is the ability of the WebApp and its server environment to rebuff unauthorized access and/or thwart an outright malevolent attack. A detailed discussion of WebApp security is beyond the scope of this book. If you have further interest, see [Vac06], [Kiz05], or [Kal03].

**Availability.** Even the best WebApp will not meet users' needs if it is unavailable. In a technical sense, availability is the measure of the percentage of time that a WebApp is available for use. The typical end user expects WebApps to be available 24/7/365. Anything less is deemed unacceptable.<sup>3</sup> But "up-time" is not the only indicator of availability. Offutt [Off02] suggests that "using features available on only one browser or one platform" makes the WebApp unavailable to those with a different browser/platform configuration. The user will invariably go elsewhere.

**Scalability.** Can the WebApp and its server environment be scaled to handle 100, 1000, 10,000, or 100,000 users? Will the WebApp and the systems with which it is interfaced handle significant variation in volume or will responsiveness drop dramatically (or cease altogether)? It is not enough to build a WebApp that is successful. It is equally important to build a WebApp that can accommodate the burden of success (significantly more end users) and become even more successful.

<sup>3</sup> This expectation is, of course, unrealistic. Major WebApps must schedule "downtime" for fixes and upgrades.

**Time-to-market.** Although time-to-market is not a true quality attribute in the technical sense, it is a measure of quality from a business point of view. The first WebApp to address a specific market segment often captures a disproportionate number of end users.

### INFO



#### WebApp Design—Quality Checklist

The following checklist, adapted from information presented at [Webreference.com](http://Webreference.com), provides a set of questions that will help both Web designers and end users assess overall WebApp quality:

- Can content and/or function and/or navigation options be tailored to the user's preferences?
- Can content and/or functionality be customized to the bandwidth at which the user communicates?
- Have graphics and other nontext media been used appropriately? Are graphics file sizes optimized for display efficiency?

- Are tables organized and sized in a manner that makes them understandable and displayed efficiently?
- Is HTML optimized to eliminate inefficiencies?
- Is the overall page design easy to read and navigate?
- Do all pointers provide links to information that is of interest to users?
- Is it likely that most links have persistence on the Web?
- Is the WebApp instrumented with site management utilities that include tools for usage tracking, link testing, local searching, and security?

Billions of Web pages are available for those in search of information. Even well-targeted Web searches result in an avalanche of content. With so many sources of information to choose from, how does the user assess the quality (e.g., veracity, accuracy, completeness, timeliness) of the content that is presented within a WebApp? Tillman [Til00] suggests a useful set of criteria for assessing the quality of content:

**What should we consider when assessing content quality?**

- Can the scope and depth of content be easily determined to ensure that it meets the user's needs?
- Can the background and authority of the content's authors be easily identified?
- Is it possible to determine the currency of the content, the last update, and what was updated?
- Are the content and its location stable (i.e., will they remain at the referenced URL)?

In addition to these content-related questions, the following might be added:

- Is content credible?
- Is content unique? That is, does the WebApp provide some unique benefit to those who use it?
- Is content valuable to the targeted user community?
- Is content well organized? Indexed? Easily accessible?

The checklists noted in this section represent only a small sampling of the issues that should be addressed as the design of a WebApp evolves.

## 13.2 DESIGN GOALS

In her regular column on Web design, Jean Kaiser [Kai02] suggests a set of design goals that are applicable to virtually every WebApp regardless of application domain, size, or complexity:

 **vote:**

"Just because you can, doesn't mean you should."

Jean Kaiser

 **vote:**

"To some, Web design focuses on visual look and feel . . . To others, Web design is about structuring information and navigation through the document space. Others might even consider Web design to be about the technology . . . In reality, design includes all of these things and maybe more."

Thomas Powell

**Simplicity.** Although it may seem old-fashioned, the aphorism "all things in moderation" applies to WebApps. There is a tendency among some designers to provide the end user with "too much"—exhaustive content, extreme visuals, intrusive animation, enormous Web pages, the list is long. Better to strive for moderation and simplicity.

Content should be informative but succinct and should use a delivery mode (e.g., text, graphics, video, audio) that is appropriate to the information that is being delivered. Aesthetics should be pleasing, but not overwhelming (e.g., too many colors tend to distract the user rather than enhancing the interaction). Architecture should achieve WebApp objectives in the simplest possible manner. Navigation should be straightforward and navigation mechanisms should be intuitively obvious to the end user. Functions should be easy to use and easier to understand.

**Consistency.** This design goal applies to virtually every element of the design model. Content should be constructed consistently (e.g., text formatting and font styles should be the same across all text documents; graphic art should have a consistent look, color scheme, and style). Graphic design (aesthetics) should present a consistent look across all parts of the WebApp. Architectural design should establish templates that lead to a consistent hypermedia structure. Interface design should define consistent modes of interaction, navigation, and content display. Navigation mechanisms should be used consistently across all WebApp elements. As Kaiser [Kai02] notes: "Remember that to a visitor, a Web site is a physical place. It is confusing if pages within a site are not consistent in design."

**Identity.** The aesthetic, interface, and navigational design of a WebApp must be consistent with the application domain for which it is to be built. A website for a hip-hop group will undoubtedly have a different look and feel than a WebApp designed for a financial services company. The WebApp architecture will be entirely different, interfaces will be constructed to accommodate different categories of users; navigation will be organized to accomplish different objectives. You (and other design contributors) should work to establish an identity for the WebApp through the design.

**Robustness.** Based on the identity that has been established, a WebApp often makes an implicit "promise" to a user. The user expects robust content and functions that are relevant to the user's needs. If these elements are missing or insufficient, it is likely that the WebApp will fail.

**Navigability.** I have already noted that navigation should be simple and consistent. It should also be designed in a manner that is intuitive and predictable. That is,

the user should understand how to move about the WebApp without having to search for navigation links or instructions. For example, if a field of graphic icons or images contains selected icons or images that will be used as navigation mechanisms, these must be identified visually. Nothing is more frustrating than trying to find the appropriate live link among many graphical images.

It is also important to position links to major WebApp content and functions in a predictable location on every Web page. If page scrolling is required (and this is often the case), links at the top and bottom of the page make the user's navigation tasks easier.

**Visual Appeal.** Of all software categories, Web applications are unquestionably the most visual, the most dynamic, and the most unapologetically aesthetic. Beauty (visual appeal) is undoubtedly in the eye of the beholder, but many design characteristics (e.g., the look and feel of content; interface layout; color coordination; the balance of text, graphics, and other media; navigation mechanisms) do contribute to visual appeal.

**Compatibility.** A WebApp will be used in a variety of environments (e.g., different hardware, Internet connection types, operating systems, browsers) and must be designed to be compatible with each.

### 13.3 A DESIGN PYRAMID FOR WEBAPPS

 **note:**

"If a site is perfectly usable but it lacks an elegant and appropriate design style, it will fail."

**Curt Cloninger**

What is WebApp design? This simple question is more difficult to answer than one might believe. In our book [Pre08] on Web engineering, David Lowe and I discuss this when we write:

The creation of an effective design will typically require a diverse set of skills. Sometimes, for small projects, a single developer may need to be multi-skilled. For larger projects, it may be advisable and/or feasible to draw on the expertise of specialists: Web engineers, graphic designers, content developers, programmers, database specialists, information architects, network engineers, security experts, and testers. Drawing on these diverse skills allows the creation of a model that can be assessed for quality and improved *before* content and code are generated, tests are conducted, and end-users become involved in large numbers. If analysis is where *WebApp quality is established*, then design is where the *quality is truly embedded*.

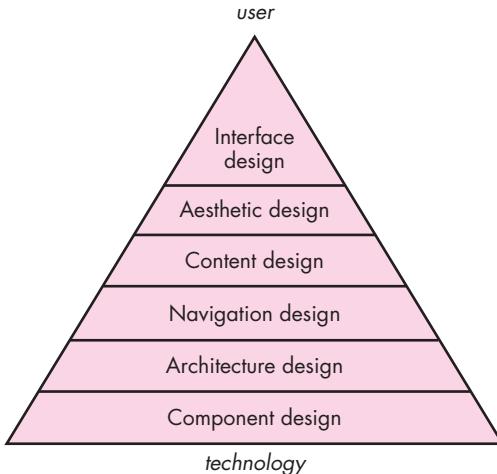
The appropriate mix of design skills will vary depending upon the nature of the WebApp. Figure 13.2 depicts a design pyramid for WebApps. Each level of the pyramid represents a design action that is described in the sections that follow.

### 13.4 WEBAPP INTERFACE DESIGN

When a user interacts with a computer-based system, a set of fundamental principles and overriding design guidelines apply. These have been discussed in

**FIGURE 13.2**

A design pyramid for WebApps



Chapter 11.<sup>4</sup> Although WebApps present a few special user interface design challenges, the basic principles and guidelines are applicable.

One of the challenges of interface design for WebApps is the indeterminate nature of the user's entry point. That is, the user may enter the WebApp at a "home" location (e.g., the home page) or may be linked into some lower level of the WebApp architecture. In some cases, the WebApp can be designed in a way that reroutes the user to a home location, but if this is undesirable, the WebApp design must provide interface navigation features that accompany all content objects and are available regardless of how the user enters the system.

The objectives of a WebApp interface are to: (1) establish a consistent window into the content and functionality provided by the interface, (2) guide the user through a series of interactions with the WebApp, and (3) organize the navigation options and content available to the user. To achieve a consistent interface, you should first use aesthetic design (Section 13.5) to establish a coherent "look." This encompasses many characteristics, but must emphasize the layout and form of navigation mechanisms. To guide user interaction, you may draw on an appropriate metaphor<sup>5</sup> that enables the user to gain an intuitive understanding of the interface. To implement navigation options, you can select from one of a number of interaction mechanisms:

- *Navigation menus*—keyword menus (organized vertically or horizontally) that list key content and/or functionality. These menus may be implemented so

<sup>4</sup> Section 11.5 is dedicated to WebApp interface design. If you have not already done so, read it at this time.

<sup>5</sup> In this context, a *metaphor* is a representation (drawn from the user's real-world experience) that can be modeled within the context of the interface. A simple example might be a slider switch that is used to control the auditory volume of an .mpg file.

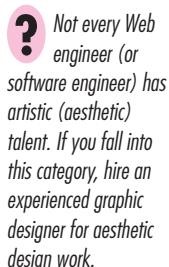


that the user can choose from a hierarchy of subtopics that is displayed when the primary menu option is selected.

- *Graphic icons*—button, switches, and similar graphical images that enable the user to select some property or specify a decision.
- *Graphic images*—some graphical representation that is selectable by the user and implements a link to a content object or WebApp functionality.

It is important to note that one or more of these control mechanisms should be provided at every level of the content hierarchy.

## 13.5 AESTHETIC DESIGN



Aesthetic design, also called *graphic design*, is an artistic endeavor that complements the technical aspects of WebApp design. Without it, a WebApp may be functional, but unappealing. With it, a WebApp draws its users into a world that embraces them on a visceral, as well as an intellectual level.

But what is aesthetic? There is an old saying, “beauty exists in the eye of the beholder.” This is particularly appropriate when aesthetic design for WebApps is considered. To perform effective aesthetic design, return to the user hierarchy developed as part of the requirements model (Chapter 5) and ask, *Who are the WebApp’s users and what “look” do they desire?*

### 13.5.1 Layout Issues

Every Web page has a limited amount of “real estate” that can be used to support non-functional aesthetics, navigation features, informational content, and user-directed functionality. The development of this real estate is planned during aesthetic design.

Like all aesthetic issues, there are no absolute rules when screen layout is designed. However, a number of general layout guidelines are worth considering:

**Don’t be afraid of white space.** It is inadvisable to pack every square inch of a Web page with information. The resulting clutter makes it difficult for the user to identify needed information or features and create visual chaos that is not pleasing to the eye.

**Emphasize content.** After all, that’s the reason the user is there. Nielsen [Nie00] suggests that the typical Web page should be 80 percent content with the remaining real estate dedicated to navigation and other features.

**Organize layout elements from top-left to bottom-right.** The vast majority of users will scan a Web page in much the same way as they scan the page of a book—top-left to bottom-right.<sup>6</sup> If layout elements have specific

### note:

“We find that people quickly evaluate a site by visual design alone.”

**Stanford Guidelines for Web Credibility**

<sup>6</sup> There are exceptions that are cultural and language-based, but this rule does hold for most users.

priorities, high-priority elements should be placed in the upper-left portion of the page real estate.

### Group navigation, content, and function geographically within the page.

Humans look for patterns in virtually all things. If there are no discernable patterns within a Web page, user frustration is likely to increase (due to unnecessary searching for needed information).



*Users tend to tolerate vertical scrolling more readily than horizontal scrolling. Avoid wide page formats.*

**Don't extend your real estate with the scrolling bar.** Although scrolling is often necessary, most studies indicate that users would prefer not to scroll. It is better to reduce page content or to present necessary content on multiple pages.

### Consider resolution and browser window size when designing layout.

Rather than defining fixed sizes within a layout, the design should specify all layout items as a percentage of available space [Nie00].

#### 13.5.2 Graphic Design Issues

Graphic design considers every aspect of the look and feel of a WebApp. The graphic design process begins with layout (Section 13.5.1) and proceeds into a consideration of global color schemes; text types, sizes, and styles; the use of supplementary media (e.g., audio, video, animation); and all other aesthetic elements of an application.

A comprehensive discussion of graphic design issues for WebApps is beyond the scope of this book. You can obtain design tips and guidelines from many websites that are dedicated to the subject (e.g., [www.graphic-design.com](http://www.graphic-design.com), [www.grantasticdesigns.com](http://www.grantasticdesigns.com), [www.wpdfd.com](http://www.wpdfd.com)) or from one or more print resources (e.g., [Roc06] and [Gor02]).

#### INFO



##### Well-Designed Websites

Sometimes, the best way to understand good WebApp design is to look at a few examples. In his article, "The Top Twenty Web Design Tips," Marcelle Toor ([www.graphic-design.com/Web/feature/tips.html](http://www.graphic-design.com/Web/feature/tips.html)) suggests the following websites as examples of good graphic design:

[www.creativepro.com/designresource/home/](http://www.creativepro.com/designresource/home/)

**787.html**—a design firm headed by Primo Angeli

[www.workbook.com](http://www.workbook.com)—this site showcases work by illustrators and designers

**www.pbs.org/riverofsong**—a television series for public TV and radio about American music

**www.RKDINC.com**—a design firm with online portfolio and good design tips

**www.creativehotlist.com/index.html**—a good source for well-designed sites developed by ad agencies, graphics arts firms, and other communications specialists

**www.btdnyc.com**—a design firm headed by Beth Toudreau

## 13.6 CONTENT DESIGN

**quote:**

"Good designers can create normalcy out of chaos; they can clearly communicate ideas through the organizing and manipulating of words and pictures."

Jeffery Veen

Content design focuses on two different design tasks, each addressed by individuals with different skill sets. First, a design representation for content objects and the mechanisms required to establish their relationship to one another is developed. In addition, the information within a specific content object is created. The latter task may be conducted by copywriters, graphic designers, and others who generate the content to be used within a WebApp.

### 13.6.1 Content Objects

The relationship between content objects defined as part of a requirements model for the WebApp and design objects representing content is analogous to the relationship between analysis classes and design components described in earlier chapters. In the context of WebApp design, a content object is more closely aligned with a data object for traditional software. A content object has attributes that include content-specific information (normally defined during WebApp requirements modeling) and implementation-specific attributes that are specified as part of design.

As an example, consider an analysis class, **ProductComponent**, developed for the *SafeHome* e-commerce system. The analysis class attribute, **description**, is represented as a design class named **CompDescription** composed of five content objects: **MarketingDescription**, **Photograph**, **TechDescription**, **Schematic**, and **Video** shown as shaded objects noted in Figure 13.3. Information contained within the content object is noted as attributes. For example, **Photograph** (a .jpg image) has the attributes **horizontal dimension**, **vertical dimension**, and **border style**.

UML association and an aggregation<sup>7</sup> may be used to represent relationships between content objects. For example, the UML association shown in Figure 13.3 indicates that one **CompDescription** is used for each instance of the **ProductComponent** class. **CompDescription** is composed on the five content objects shown. However, the multiplicity notation shown indicates that **Schematic** and **Video** are optional (0 occurrences are possible), one **MarketingDescription** and one **TechDescription** are required, and one or more instances of **Photograph** are used.

### 13.6.2 Content Design Issues

Once all content objects are modeled, the information that each object is to deliver must be authored and then formatted to best meet the customer's needs. Content authoring is the job of specialists in the relevant area who design the content object by providing an outline of information to be delivered and an indication of the types of generic content objects (e.g., descriptive text, graphic images, photographs) that will be used to deliver the information. Aesthetic design (Section 13.5) may also be applied to represent the proper look and feel for the content.

---

<sup>7</sup> Both of these representations are discussed in Appendix 1.

**FIGURE 13.3**

**Design representation of content objects**



As content objects are designed, they are “chunked” [Pow02] to form WebApp pages. The number of content objects incorporated into a single page is a function of user needs, constraints imposed by download speed of the Internet connection, and restrictions imposed by the amount of scrolling that the user will tolerate.

### 13.7 ARCHITECTURE DESIGN

**note:**

“...the architectural structure of a well designed site is not always apparent to the user—nor should it be.”

Thomas Powell

Architecture design is tied to the goals established for a WebApp, the content to be presented, the users who will visit, and the navigation philosophy that has been established. As an architectural designer, you must identify content architecture and WebApp architecture. *Content architecture*<sup>8</sup> focuses on the manner in which content objects (or composite objects such as Web pages) are structured for presentation and navigation. *WebApp architecture* addresses the manner in which the application is structured to manage user interaction, handle internal processing tasks, effect navigation, and present content.

In most cases, architecture design is conducted in parallel with interface design, aesthetic design, and content design. Because the WebApp architecture may have a

<sup>8</sup> The term *information architecture* is also used to connote structures that lead to better organization, labeling, navigation, and searching of content objects.

strong influence on navigation, the decisions made during this design action will influence work conducted during navigation design.

### 13.7.1 Content Architecture

**What types of content architectures are commonly encountered?**

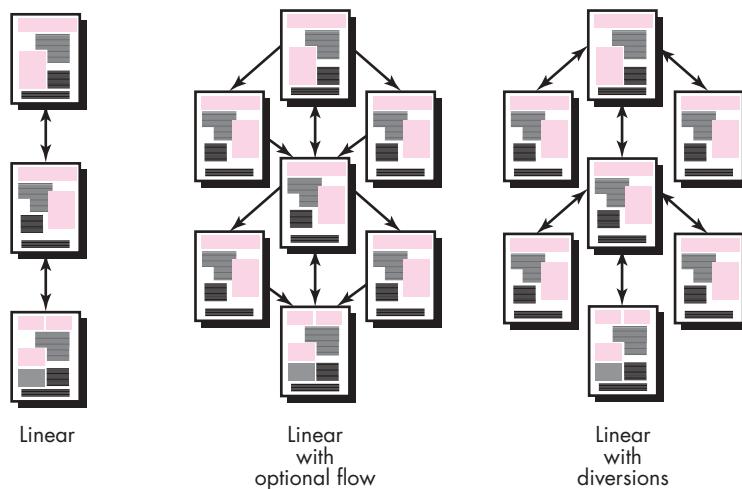
The design of content architecture focuses on the definition of the overall hypermedia structure of the WebApp. Although custom architectures are sometimes created, you always have the option of choosing from four different content structures [Pow00]:

*Linear structures* (Figure 13.4) are encountered when a predictable sequence of interactions (with some variation or diversion) is common. A classic example might be a tutorial presentation in which pages of information along with related graphics, short videos, or audio are presented only after prerequisite information has been presented. The sequence of content presentation is predefined and generally linear. Another example might be a product order entry sequence in which specific information must be specified in a specific order. In such cases, the structures shown in Figure 13.4 are appropriate. As content and processing become more complex, the purely linear flow shown on the left of the figure gives way to more sophisticated linear structures in which alternative content may be invoked or a diversion to acquire complementary content (structure shown on the right side of Figure 13.4) occurs.

*Grid structures* (Figure 13.5) are an architectural option that you can apply when WebApp content can be organized categorically in two (or more) dimensions. For example, consider a situation in which an e-commerce site sells golf clubs. The horizontal dimension of the grid represents the type of club to be sold (e.g., woods, irons, wedges, putters). The vertical dimension represents the offerings provided by various golf club manufacturers. Hence, a user might navigate the grid horizontally to find the putters column and then vertically to examine the offerings provided by

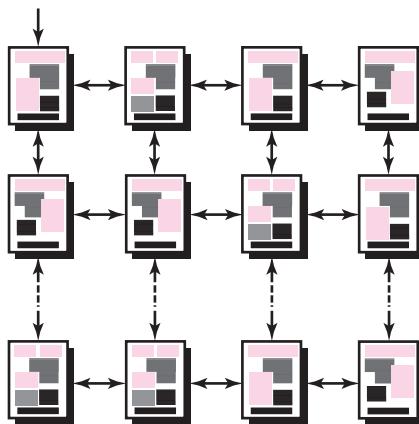
**FIGURE 13.4**

Linear structures

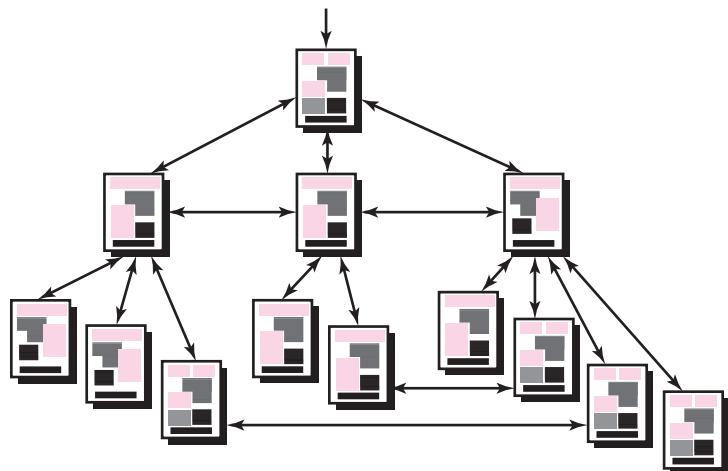


**FIGURE 13.5**

Grid structure

**FIGURE 13.6**

Hierarchical structure



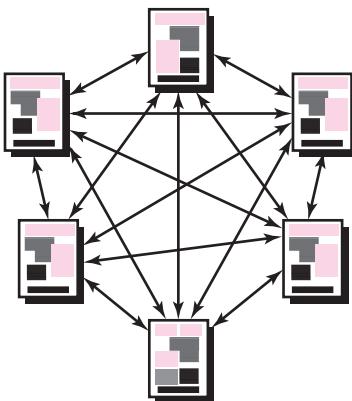
those manufacturers that sell putters. This WebApp architecture is useful only when highly regular content is encountered [Pow00].

*Hierarchical structures* (Figure 13.6) are undoubtedly the most common WebApp architecture. Unlike the partitioned software hierarchies discussed in Chapter 9 that encourage flow of control only along vertical branches of the hierarchy, a WebApp hierarchical structure can be designed in a manner that enables (via hypertext branching) flow of control horizontally across vertical branches of the structure. Hence, content presented on the far left-hand branch of the hierarchy can have hypertext links that lead directly to content that exists in the middle or right-hand branch of the structure. It should be noted, however, that although such branching allows rapid navigation across WebApp content, it can lead to confusion on the part of the user.

A *networked* or “pure web” structure (Figure 13.7) is similar in many ways to the architecture that evolves for object-oriented systems. Architectural components

**FIGURE 13.7**

**Network structure**



(in this case, Web pages) are designed so that they may pass control (via hypertext links) to virtually every other component in the system. This approach allows considerable navigation flexibility, but at the same time, can be confusing to a user.

The architectural structures discussed in the preceding paragraphs can be combined to form *composite structures*. The overall architecture of a WebApp may be hierarchical, but part of the structure may exhibit linear characteristics, while another part of the architecture may be networked. Your goal as an architectural designer is to match the WebApp structure to the content to be presented and the processing to be conducted.

### 13.7.2 WebApp Architecture

WebApp architecture describes an infrastructure that enables a Web-based system or application to achieve its business objectives. Jacyntho and his colleagues [Jac02b] describe the basic characteristics of this infrastructure in the following manner:

Applications should be built using layers in which different concerns are taken into account; in particular, application data should be separated from the page's contents (navigation nodes) and these contents, in turn, should be clearly separated from the interface look-and-feel (pages).

The authors suggest a three-layer design architecture that decouples interface from navigation and from application behavior. They argue that keeping interface, application, and navigation separate simplifies implementation and enhances reuse.

The *Model-View-Controller* (MVC) architecture [Kra88]<sup>9</sup> is one of a number of suggested WebApp infrastructure models that decouple the user interface from the

---

<sup>9</sup> It should be noted that MVC is actually an architectural design pattern developed for the Smalltalk environment (see [www.cetus-links.org/oo\\_smalltalk.html](http://www.cetus-links.org/oo_smalltalk.html)) and can be used for any interactive application.

**FIGURE 13.8****The MVC Architecture**

Source: Adapted from [Jac02].



The MVC architecture decouples the user interface from WebApp functionality and information content.

WebApp functionality and informational content. The *model* (sometimes referred to as the “model object”) contains all application-specific content and processing logic, including all content objects, access to external data/information sources, and all processing functionality that is application specific. The *view* contains all interface-specific functions and enables the presentation of content and processing logic, including all content objects, access to external data/information sources, and all processing functionality required by the end user. The *controller* manages access to the *model* and the *view* and coordinates the flow of data between them. In a WebApp, “the view is updated by the controller with data from the model based on user input” [WMT02]. A schematic representation of the MVC architecture is shown in Figure 13.8.

Referring to the figure, user requests or data are handled by the controller. The controller also selects the view object that is applicable based on the user request. Once the type of request is determined, a behavior request is transmitted to the model, which implements the functionality or retrieves the content required to accommodate the request. The model object can access data stored in a corporate database, as part of a local data store, or as a collection of independent files. The data developed by the model must be formatted and organized by the appropriate view object and then transmitted from the application server back to the client-based browser for display on the customer’s machine.

In many cases, WebApp architecture is defined within the context of the development environment in which the application is to be implemented. If you have further interest, see [Fow03] for a discussion of development environments and their role in the design of Web application architectures.

## 13.8 NAVIGATION DESIGN

**Quote:**

"Just wait, Gretel,  
until the moon  
rises, and then we  
shall see the  
crumbs of bread  
which I have  
strewn about, they  
will show us our  
way home again."

**Hansel and  
Gretel**

**KEY POINT**

An NSU describes the navigation requirements for each use case. In essence, the NSU shows how an actor moves between content objects or WebApp functions.

Once the WebApp architecture has been established and the components (pages, scripts, applets, and other processing functions) of the architecture have been identified, you must define navigation pathways that enable users to access WebApp content and functions. To accomplish this, you should (1) identify the semantics of navigation for different users of the site, and (2) define the mechanics (syntax) of achieving the navigation.

### 13.8.1 Navigation Semantics

Like many WebApp design actions, navigation design begins with a consideration of the user hierarchy and related use cases (Chapter 5) developed for each category of user (actor). Each actor may use the WebApp somewhat differently and therefore have different navigation requirements. In addition, the use cases developed for each actor will define a set of classes that encompass one or more content objects or WebApp functions. As each user interacts with the WebApp, she encounters a series of *navigation semantic units* (NSUs)—“a set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements” [Cac02].

An NSU is composed of a set of navigation elements called *ways of navigating* (WoN) [Gna99]. A WoN represents the best navigation pathway to achieve a navigational goal for a specific type of user. Each WoN is organized as a set of *navigational nodes* (NN) that are connected by navigational links. In some cases, a navigational link may be another NSU. Therefore, the overall navigation structure for a WebApp may be organized as a hierarchy of NSUs.

To illustrate the development of an NSU, consider the use case **Select SafeHome Components**:

#### Use Case: Select SafeHome Components

The WebApp will recommend product components (e.g., control panels, sensors, cameras) and other features (e.g., PC-based functionality implemented in software) for each room and exterior entrance. If I request alternatives, the WebApp will provide them, if they exist. I will be able to get descriptive and pricing information for each product component. The WebApp will create and display a bill-of-materials as I select various components. I'll be able to give the bill-of-materials a name and save it for future reference (see use case **Save Configuration**).

The underlined items in the use-case description represent classes and content objects that will be incorporated into one or more NSUs that will enable a new customer to perform the scenario described in the **Select SafeHome Components** use case.

Figure 13.9 depicts a partial semantic analysis of the navigation implied by the **Select SafeHome Components** use case. Using the terminology introduced earlier, the figure also represents a way of navigating (WoN) for the

**FIGURE 13.9**

**Creating an NSU**



**quote:**

"The problem of Web site navigation is conceptual, technical, spatial, philosophical and logistic. Consequently, solutions tend to call for complex improvisational combinations of art, science and organizational psychology."

Tim Horgan

**SafeHomeAssured.com** WebApp. Important problem domain classes are shown along with selected content objects (in this case the package of content objects named **CompDescription**, an attribute of the **ProductComponent** class). These items are navigation nodes. Each of the arrows represents a navigation link<sup>10</sup> and is labeled with the user-initiated action that causes the link to occur.

You can create an NSU for each use case associated with each user role. For example, a **new customer** for **SafeHomeAssured.com** may have three different use cases, all resulting in access to different information and WebApp functions. An NSU is created for each goal.

During the initial stages of navigation design, the WebApp content architecture is assessed to determine one or more WoN for each use case. As noted earlier, a WoN identifies navigation nodes (e.g., content) and then links that enable navigation between them. The WoN are then organized into NSUs.

### 13.8.2 Navigation Syntax

As design proceeds, your next task is to define the mechanics of navigation. A number of options are available as you develop an approach for implementing each NSU:



In most situations, choose either horizontal or vertical navigation mechanisms, but not both.

- *Individual navigation link*—includes text-based links, icons, buttons and switches, and graphical metaphors. You must choose navigation links that are appropriate for the content and consistent with the heuristics that lead to high-quality interface design.
- *Horizontal navigation bar*—lists major content or functional categories in a bar containing appropriate links. In general, between four and seven categories are listed.

10 These are sometimes referred to as *navigation semantic links* (NSL) [Cac02].

- *Vertical navigation column*—(1) lists major content or functional categories, or (2) lists virtually all major content objects within the WebApp. If you choose the second option, such navigation columns can “expand” to present content objects as part of a hierarchy (i.e., selecting an entry in the original column causes an expansion that lists a second layer of related content objects).
- *Tabs*—a metaphor that is nothing more than a variation of the navigation bar or column, representing content or functional categories as tab sheets that are selected when a link is required.
- *Site maps*—provide an all-inclusive table of contents for navigation to all content objects and functionality contained within the WebApp.



*The site map should be accessible from every page. The map itself should be organized so that the structure of WebApp information is readily apparent.*

In addition to choosing the mechanics of navigation, you should also establish appropriate navigation conventions and aids. For example, icons and graphical links should look “clickable” by beveling the edges to give the image a three-dimensional look. Audio or visual feedback should be designed to provide the user with an indication that a navigation option has been chosen. For text-based navigation, color should be used to indicate navigation links and to provide an indication of links already traveled. These are but a few of dozens of design conventions that make navigation user-friendly.

### 13.9 COMPONENT-LEVEL DESIGN

Modern WebApps deliver increasingly sophisticated processing functions that (1) perform localized processing to generate content and navigation capability in a dynamic fashion, (2) provide computation or data processing capability that are appropriate for the WebApp’s business domain, (3) provide sophisticated database query and access, and (4) establish data interfaces with external corporate systems. To achieve these (and many other) capabilities, you must design and construct program components that are identical in form to software components for traditional software.

The design methods discussed in Chapter 10 apply to WebApp components with little, if any, modification. The implementation environment, programming languages, and design patterns, frameworks, and software may vary somewhat, but the overall design approach remains the same.

### 13.10 OBJECT-ORIENTED HYPERMEDIA DESIGN METHOD (OOHDM)

A number of design methods for Web applications have been proposed over the past decade. To date, no single method has achieved dominance.<sup>11</sup> In this section I present a brief overview of one of the most widely discussed WebApp design methods—OOHDM.

---

<sup>11</sup> In fact, relatively few Web developers use a specific method when designing a WebApp. Hopefully, this ad hoc approach to design will change as time passes.

**FIGURE 13.10** Summary of the OOHDM method.

Source: Adapted from [Sch95].

				
	Conceptual design	Navigational design	Abstract interface design	Implementation
Work products	Classes, subsystems, relationships, attributes	Nodes links, access structures, navigational contexts, navigational transformations	Abstract interface objects, responses to external events, transformations	Executable WebApp
Design mechanisms	Classification, composition, aggregation, generalization specialization	Mapping between conceptual and navigation objects	Mapping between navigation and perceptible objects	Resource provided by target environment
Design concerns	Modeling semantics of the application domain	Takes into account user profile and task. Emphasis on cognitive aspects.	Modeling perceptible objects, implementing chosen metaphors. Describe interface for navigational objects.	Correctness; application performance; completeness

Daniel Schwabe and his colleagues [Sch95, Sch98b] originally proposed the *Object-Oriented Hypermedia Design Method* (OOHDM), which is composed of four different design activities: conceptual design, navigational design, abstract interface design, and implementation. A summary of these design activities is shown in Figure 13.10 and discussed briefly in the sections that follow.

### 13.10.1 Conceptual Design for OOHDM

OOHDM *conceptual design* creates a representation of the subsystems, classes, and relationships that define the application domain for the WebApp. UML may be used<sup>12</sup> to create appropriate class diagrams, aggregations, and composite class representations, collaboration diagrams, and other information that describes the application domain.

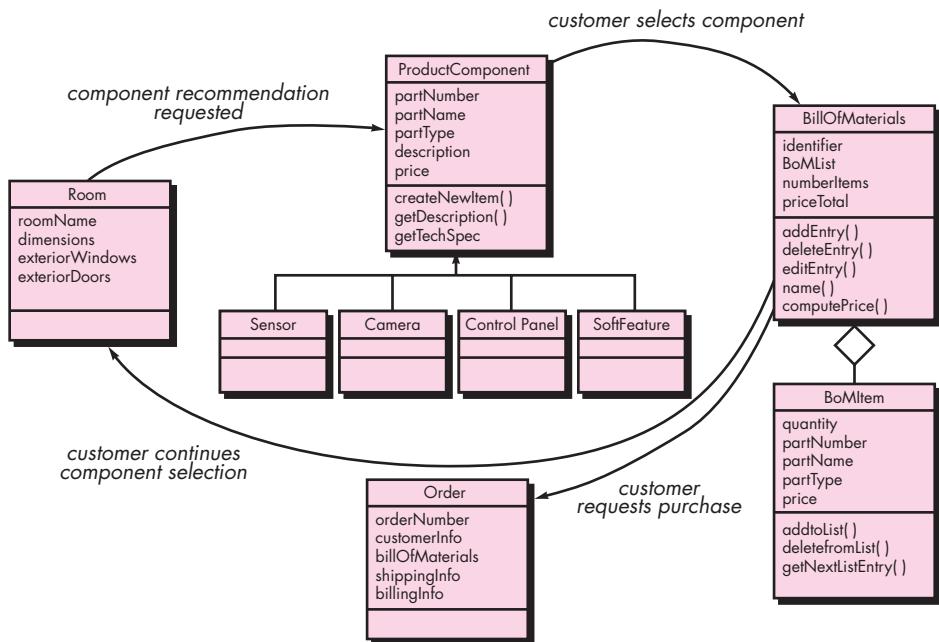
As a simple example of OOHDM conceptual design, consider the **SafeHomeAssured.com** e-commerce application. A partial “conceptual schema” is shown in Figure 13.11. The class diagrams, aggregations, and related information developed as part of WebApp analysis are reused during conceptual design to represent relationships between classes.

### 13.10.2 Navigational Design for OOHDM

*Navigational design* identifies a set of “objects” that are derived from the classes defined in conceptual design. A series of “navigational classes” or “nodes” are

<sup>12</sup> OOHDM does not prescribe a specific notation; however, the use of UML is common when this method is applied.

**FIGURE 13.11** Partial conceptual schema for **SafeHomeAssured.com**



defined to encapsulate these objects. UML may be used to create appropriate use cases, state charts, and sequence diagrams—all representations that assist you in better understanding navigational requirements. In addition, design patterns for navigational design may be used as the design is developed. OOHDM uses a predefined set of navigation classes—nodes, links, anchors, and access structures [Sch98b]. Access structures are more elaborate and include mechanisms such as a WebApp index, a site map, or a guided tour.

Once navigation classes are defined, OOHDM “structures the navigation space by grouping navigation objects into sets called contexts” [Sch98b]. A *context* includes a description of the local navigation structure, restriction imposed on the access of content objects, and methods (operations) required to effect access of content objects. A context template (analogous to CRC cards discussed in Chapter 6) is developed and may be used to track the navigation requirements of each category of user through the various contexts defined in OOHDM. Doing this, specific navigation paths (what we called WoN in Section 13.8.1) emerge.

### 13.10.3 Abstract Interface Design and Implementation

The *abstract interface design* action specifies the interface objects that the user sees as WebApp interaction occurs. A formal model of interface objects, called an *abstract data view* (ADV), is used to represent the relationship between interface objects and navigation objects, and the behavioral characteristics of interface objects.

The ADV model defines a “static layout” [Sch98b] that represents the interface metaphor and includes a representation of navigation objects within the interface and the specification of the interface objects (e.g., menus, buttons, icons) that assist in navigation and interaction. In addition, the ADV model contains a behavioral component (similar to the UML state diagram) that indicates how external events “trigger navigation and which interface transformations occur when the user interacts with the application” [Sch01a].

The OOHDM *implementation* activity represents a design iteration that is specific to the environment in which the WebApp will operate. Classes, navigation, and the interface are each characterized in a manner that can be constructed for the client-server environment, operating systems, support software, programming languages, and other environmental characteristics that are relevant to the problem.

### 13.11 SUMMARY

The quality of a WebApp—defined in terms of usability, functionality, reliability, efficiency, maintainability, security, scalability, and time-to-market—is introduced during design. To achieve these quality attributes, a good WebApp design should exhibit the following characteristics: simplicity, consistency, identity, robustness, navigability, and visual appeal. To achieve these characteristics, the WebApp design activity focuses on six different elements of the design.

Interface design describes the structure and organization of the user interface and includes a representation of screen layout, a definition of the modes of interaction, and a description of navigation mechanisms. A set of interface design principles and an interface design workflow guide you when layout and interface control mechanisms are designed.

Aesthetic design, also called graphic design, describes the “look and feel” of the WebApp and includes color schemes; geometric layout; text size, font, and placement; the use of graphics; and related aesthetic decisions. A set of graphic design guidelines provides the basis for a design approach.

Content design defines the layout, structure, and outline for all content that is presented as part of the WebApp and establishes the relationships between content objects. Content design begins with the representation of content objects, their associations, and relationships. A set of browsing primitives establishes the basis for navigation design.

Architecture design identifies the overall hypermedia structure for the WebApp and encompasses both content architecture and WebApp architecture. Architectural styles for content include linear, grid, hierarchical, and network structures. WebApp architecture describes an infrastructure that enables a Web-based system or application to achieve its business objectives.

Navigation design represents the navigational flow between content objects and for all WebApp functions. Navigation semantics are defined by describing a set of navigation semantic units. Each unit is composed of ways of navigating and navigational links and nodes. Navigation syntax depicts the mechanisms used for effecting the navigation described as part of the semantics.

Component design develops the detailed processing logic required to implement functional components that implement a complete WebApp function. Design techniques described in Chapter 10 are applicable for the engineering of WebApp components.

The Object-Oriented Hypermedia Design Method (OOHDM) is one of a number of methods proposed for WebApp design. OOHDM suggests a design process that includes conceptual design, navigational design, abstract interface design, and implementation.

## PROBLEMS AND POINTS TO PONDER

**13.1.** Why is the “artistic ideal” an insufficient design philosophy when modern WebApps are built? Is there ever a case in which the artistic ideal is the philosophy to follow?

**13.2.** In this chapter we select a broad array of quality attributes for WebApps. Select the three that you believe are most important, and make an argument that explains why each should be emphasized in WebApp design work.

**13.3.** Add at least five additional questions to the WebApp Design—Quality Checklist presented in Section 13.1.

**13.4.** You are a WebApp designer for *FutureLearning Corporation*, a distance learning company. You intend to implement an Internet-based “learning engine” that will enable you to deliver course content to a student. The learning engine provides the basic infrastructure for delivering learning content on any subject (content designers will prepare appropriate content). Develop a prototype interface design for the learning engine.

**13.5.** What is the most aesthetically pleasing website you have ever visited and why?

**13.6.** Consider the content object **Order**, generated once a user of **SafeHomeAssured.com** has completed the selection of all components and is ready to finalize his purchase. Develop a UML description for **Order** along with all appropriate design representations.

**13.7.** What is the difference between content architecture and WebApp architecture?

**13.8.** Reconsidering the *FutureLearning* “learning engine” described in Problem 13.4, select a content architecture that would be appropriate for the WebApp. Discuss why you made the choice.

**13.9.** Use UML to develop three or four design representations for content objects that would be encountered as the “learning engine” described in Problem 13.4 is designed.

**13.10.** Do a bit of additional research on the MVC architecture and decide whether it would be an appropriate WebApp architecture for the “learning engine” discussed in Problem 13.4.

**13.11.** What is the difference between navigation syntax and navigation semantics?

**13.12.** Define two or three NSUs for the **SafeHomeAssured.com** WebApp. Describe each in some detail.

**13.13.** Write a brief paper on a hypermedia design method other than OOHDM.

## FURTHER READINGS AND INFORMATION SOURCES

Van Duyne and his colleagues (*The Design of Sites*, 2d ed., Prentice Hall, 2007) have written a comprehensive book that covers most important aspects of the WebApp design process. Design process models and design patterns are covered in detail. Wodtke (*Information Architecture*, New Riders Publishing, 2003), Rosenfeld and Morville (*Information Architecture for the World Wide Web*, O'Reilly & Associates, 2002), and Reiss (*Practical Information Architecture*, Addison-Wesley, 2000) address content architecture and other topics.

Although hundreds of books have been written on "Web design," very few of these discuss any meaningful technical methods for doing design work. At best, a variety of useful guidelines for WebApp design is presented, worthwhile examples of Web pages and Java programming are shown, and the technical details important for implementing modern WebApps are discussed. Among the many offerings in this category are books by Sklar (*Principles of Web Design*, 4th ed., Course Technology, 2008), McIntire (*Visual Design for the Modern Web*, New Riders Press, 2007), Niederst (*Web Design in a Nutshell*, 3d ed., O'Reilly, 2006), Eccher (*Advanced Professional Web Design*, Charles River Media, 2006), Cederholm (*Bulletproof Web Design*, New Riders Press, 2005), and Shelly and his colleagues (*Web Design*, 2d ed., Course Technology, 2005). Powell's [Pow02] encyclopedic discussion and Nielsen's [Nie00] in-depth discussion of design are also worthwhile additions to any library.

Books by Beard (*The Principles of Beautiful Web Design*, SitePoint, 2007), Clarke and Holzschlag (*Transcending CSS: The Fine Art of Web Design*, New Riders Press, 2006), and Golbeck (*Art Theory for Web Design*, Addison Wesley, 2005) emphasize aesthetic design and are worthwhile reading for practitioners who have little background in the subject.

The agile view of design (and other topics) for WebApps is presented by Wallace and his colleagues (*Extreme Programming for Web Projects*, Addison-Wesley, 2003). Conallen (*Building Web Applications with UML*, 2d ed., Addison-Wesley, 2002) and Rosenberg and Scott (*Applying Use-Case Driven Object Modeling with UML*, Addison-Wesley, 2001) present detailed examples of WebApps modeled using UML.

Design techniques are also mentioned within the context of books written about specific development environments. Interested readers should examine books on HTML, CSS, J2EE, Java, .NET, XML, Perl, Ruby on Rails, Ajax, and a variety of WebApp creation applications (*Dreamweaver*, *HomePage*, *Frontpage*, *GoLive*, *MacroMedia Flash*, etc.) for useful design tidbits.

A wide variety of information sources on design for WebApps is available on the Internet. An up-to-date list of World Wide Web references that are relevant to WebApp design can be found at the SEPA website: [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm).