

SOFTWARE ENGINEERING for IMAGE PROCESSING SYSTEMS

Phillip A. Laplante



CRC PRESS

**Also available as a printed book
see title verso for ISBN details**

**SOFTWARE
ENGINEERING
for
IMAGE
PROCESSING
SYSTEMS**

IMAGE PROCESSING SERIES

Series Editor: Phillip A. Laplante, Pennsylvania State University

Published Titles

Adaptive Image Processing: A Computational Intelligence Perspective

Stuart William Perry, Hau-San Wong, and Ling Guan

Image Acquisition and Processing with LabVIEW™

Christopher G. Relf

Image and Video Compression for Multimedia Engineering

Yun Q. Shi and Huiyang Sun

Multimedia Image and Video Processing

Ling Guan, S.Y. Kung, and Jan Larsen

Shape Analysis and Classification: Theory and Practice

Luciano da Fontoura Costa and Roberto Marcondes Cesar Jr.

Software Engineering for Image Processing Systems

Phillip A. Laplante

SOFTWARE ENGINEERING for IMAGE PROCESSING SYSTEMS

Phillip A. Laplante



CRC PRESS

Boca Raton London New York Washington, D.C.

This edition published in the Taylor & Francis e-Library, 2005.

“To purchase your own copy of this or any of Taylor & Francis or Routledge’s collection of thousands of eBooks please go to www.eBookstore.tandf.co.uk.”

Library of Congress Cataloging-in-Publication Data

Laplante, Phillip A.

Software engineering for image processing systems / Phillip A. Laplante.

p. cm. — (Image processing series)

Includes bibliographical references and index.

ISBN 0-8493-1376-7

1. Image processing. 2. Software engineering. I. Title. II. Series.

TA1637.L34 2003

621.36’7—dc21

2003046213

This book contains information obtained from authentic and highly regarded sources. Reprinted material is quoted with permission, and sources are indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

Neither this book nor any part may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, microfilming, and recording, or by any information storage or retrieval system, without prior permission in writing from the publisher.

The consent of CRC Press LLC does not extend to copying for general distribution, for promotion, for creating new works, or for resale. Specific permission must be obtained in writing from CRC Press LLC for such copying.

Direct all inquiries to CRC Press LLC, 2000 N.W. Corporate Blvd., Boca Raton, Florida 33431.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation, without intent to infringe.

Visit the CRC Press Web site at www.crcpress.com

© 2004 by CRC Press LLC

No claim to original U.S. Government works
International Standard Book Number 0-8493-1376-7
Library of Congress Card Number 2003046213

ISBN 0-203-49610-8 Master e-book ISBN

ISBN 0-203-58596-8 (Adobe eReader Format)

Dedication

to

Nancy, Christopher, and Charlotte

Preface

This book is not intended to be a traditional software engineering text — there are many good ones. Instead, it is designed specifically for those involved in image processing systems. It provides a modern engineering framework for the specification, design, coding, testing, and maintenance of image processing software and systems. In particular, the focus is on imaging systems as a special case of software, thereby providing a common framework and language of discourse for imaging engineers of all backgrounds. This common framework, in turn, should lead to more reliable and economical software throughout the imaging industry.

This text is about image processing, but it contains none because it is not about the images; it is about the software systems that process them. Image processing systems are found everywhere in such places as digital cameras, photocopying systems, computer scanners, video games, industrial inspection, medical imaging, and defense applications. It seems that a software engineering book devoted to this important application domain is overdue.

This text is intended to help those in industry who, though practicing as software engineers, have had little or no formal training in software engineering. The typical reader will have an undergraduate degree in mathematics or in an engineering or physical science or the equivalent experience. Some experience in working on a software project team, as a requirements writer, designer, developer, tester, or manager, would be helpful but is not essential. No experience in programming in any particular language is assumed on the part of the reader. It would be helpful if the reader had a mathematical background at least embracing calculus, but this is not essential since most of software engineering does not involve deep formulaic principles.

It is hoped that those who read this text will come to reject the many misconceptions about software, software engineering, and software engineers. Then the main contribution of this text will be to enhance the quality of software produced by the imaging industry and the standing of those who build these systems.

Acknowledgments

I am indebted to my Penn State colleague Dr. Colin Neill for numerous ideas, discussions, many of the nicer figures, and his strong influence on me and his instruction on the best practices in object-oriented technologies. In particular, he deserves coauthor credit for Chapters 4 and 5, as much of the material was drawn from a series of papers that we wrote together (Laplante and Neill, 2002a; Laplante et al., 2002c, 2002e, 2003b; Neill and Laplante, 2002b, 2003a) and from his own work (Neill and Holt, 2002a; Neill and Gill, 2003b).

I also thank Dr. Dave Sinha, chief software scientist at Kodak Medical Imaging, for discussions on applications and Dr. Michael Hinchey of the NASA Software Engineering Lab at Goddard Space Flight Center for discussions on requirements, design, and formal methods. My colleague Will Gilreath also provided helpful feedback on the draft manuscript.

Over the years, many of my graduate students have provided insight into “real-world” practices. I would like to acknowledge these students for many informative discussions and, in particular, Mike Rzucidlo for his work on software black boxes and fault tolerance, which contributed significantly to the sections on those topics.

Finally, I would be remiss if I neglected to thank CRC’s engineering editor, Nora Konopka, for her many years of support on this and other projects; Helena Redshaw for her assistance during the editorial and preproduction processes; Sheyanne Armstrong for a thoughtful copy edit; and project editor, Marsha Hecht, for her guidance in completing this endeavor.

Author

Dr. Phillip A. Laplante is associate professor of software engineering and a member of the graduate faculty at the Pennsylvania State University. He is also the chief technology officer of the Eastern Technology Council, a nonprofit business advocacy group serving the Greater Philadelphia Metropolitan Area. Before joining Penn State, Dr. Laplante was a professor and later senior academic administrator at several other colleges and universities.

Prior to his academic career, Dr. Laplante spent almost 8 years as a software engineer and project manager working on avionics (including the space shuttle), CAD, and software test systems. He has authored or edited 15 books and more than 100 papers, articles, and editorials. He co-founded the journal *Real-Time Imaging*, which he edited for 5 years, and he created and edits the CRC Press book series on image processing.

Dr. Laplante received his B.S., M.Eng., and Ph.D. in computer science, electrical engineering, and computer science, respectively, from Stevens Institute of Technology and an M.B.A. from the University of Colorado. He is a senior member of the Institute of Electrical and Electronics Engineers (IEEE) and a member of numerous professional societies, program committees, and boards. He is a licensed professional engineer in Pennsylvania and has provided consulting services to Fortune 500 companies, the U.S. Department of Defense, and NASA on real-time systems, image processing, and software engineering. Dr. Laplante also serves on the boards of several privately held companies.

Contents

Chapter 1	Software Engineering: An Overview	1
1.1	Introduction	1
1.2	A Case for a Software Engineering Approach to Building Imaging Systems	2
1.3	The Role of the Software Engineer	3
1.4	The Nature of Software for Imaging Systems	4
1.5	Case Study: A Visual Inspection System	5
1.6	Misconceptions about Software Engineering	7
1.7	Exercises	8
Chapter 2	Imaging Software and Its Properties	9
2.1	Classification of Software Qualities	9
2.1.1	Reliability	9
2.1.2	Correctness	12
2.1.3	Performance	12
2.1.4	Usability	13
2.1.5	Interoperability	13
2.1.6	Maintainability	13
2.1.7	Portability	14
2.1.8	Verifiability	14
2.1.9	Summary of Software Properties and Associated Metrics	14
2.2	Basic Software Engineering Principles	15
2.2.1	Rigor and Formality	15
2.2.2	Separation of Concerns	15
2.2.3	Modularity	15
2.2.4	Anticipation of Change	18
2.2.5	Generality	20
2.2.6	Incrementality	20
2.2.7	Traceability	20
2.3	Exercises	22
Chapter 3	Software Process and Life Cycle Models	23
3.1	Software Processes and Methodologies	23
3.2	Software Life Cycle Models	23
3.2.1	The Waterfall Model	24
3.2.1.1	Software Conception	25
3.2.1.2	Requirements Specification	25

3.2.1.3	Software Design.....	26
3.2.1.4	Software Development.....	26
3.2.1.5	Testing.....	27
3.2.1.6	Software Maintenance	27
3.2.1.7	Backtracking Transitions in the Waterfall Life Cycle	27
3.2.1.8	Waterfall Model Summary	28
3.2.2	V Model	28
3.2.3	The Spiral Model	28
3.2.4	Evolutionary Model	30
3.2.5	Incremental Model	31
3.2.6	Fountain Model	31
3.2.7	Lightweight Methodologies	32
3.2.8	Unified Process Model.....	34
3.2.9	Capability Maturity Model	34
3.2.9.1	CMM-1: Initial.....	34
3.2.9.2	CMM-2: Repeatable	34
3.2.9.3	CMM-3: Defined	35
3.2.9.4	CMM-4: Managed	35
3.2.9.5	CMM-5: Optimizing.....	35
3.2.9.6	CMM-I	35
3.2.10	Prototyping and Risk	35
3.3	Software Standards	36
3.3.1	DOD-STD-2167A	37
3.3.2	DOD-STD-498	37
3.3.3	ISO 9000-3	38
3.3.4	ISO/IEC	39
3.4	Exercises.....	40
Chapter 4	Software Requirements	41
4.1	Requirements Engineering Process	41
4.2	Types of Requirements	42
4.3	Requirements Users	43
4.4	Formal Methods in Software Specification	44
4.4.1	Limitations of Formal Methods	45
4.4.2	Z	46
4.4.3	Finite State Machines.....	46
4.4.4	Statecharts	49
4.4.5	Petri Nets.....	51
4.5	Specification of Imaging Systems: A Survey of Current Practices	53
4.5.1	Multiresolution Block-Matching System Specification Using a Block Diagram and Flowchart.....	55
4.5.2	Collision Testing of Graphical Objects Using Pseudo-Code.....	56
4.5.3	Functional Representation of Machine Vision System Using a Structured Approach.....	56

4.5.4	Markov Random Fields Image Reconstruction Using Object-Oriented Design	57
4.6	Case Study.....	58
4.6.1	Structured Analysis and Design.....	59
4.6.2	Structured Analysis	59
4.7	Object-Oriented Analysis.....	61
4.8	Object-Oriented vs. Structured Analysis	62
4.8.1	Recommendations on Specification Approach for Imaging Systems	64
4.9	Organizing the Requirements Document	64
4.9.1	Writing Good Requirements	66
4.10	Requirements Validation and Review	67
4.11	Some Surprises about Current Software Specification Practices.....	68
4.11.1	Surprise 1	68
4.11.2	Surprise 2	68
4.11.3	Surprise 3	68
4.11.4	Surprise 4	69
4.12	Exercises.....	69
Chapter 5	Software System Design.....	71
5.1	The Design Activity	71
5.2	Procedural-Oriented Design.....	72
5.2.1	Parnas Partitioning	72
5.2.2	Structured Design.....	74
5.2.2.1	Transitioning from Structured Analysis to Structured Design.....	74
5.2.2.2	Data Dictionaries	76
5.2.2.3	Problems with SASD in Imaging Applications	77
5.2.2.4	Real-Time Extensions of SASD.....	78
5.2.3	Design in Procedural Form Using Finite State Machines.....	78
5.3	Object-Oriented Design	80
5.3.1	Benefits of Object Orientation.....	81
5.3.1.1	Open–Closed Principle	81
5.3.1.2	Once and Only Once	82
5.3.1.3	Dependency Inversion Principle.....	82
5.3.1.4	Liskov Substitution Principle	82
5.3.2	Design Patterns	83
5.3.3	Object-Oriented Design Using Unified Modeling Language.....	84
5.3.4	Modeling Time Explicitly.....	84
5.3.5	Visual Inspection System Case Study	88
5.4	Hardware Considerations in Imaging System Design	93
5.4.1	Processors.....	94
5.4.2	Non-von Neumann Architectures	95
5.4.2.1	Single Instruction Single Data	95
5.4.2.2	Single Instruction Multiple Data	95

5.4.2.3	Multiple Instruction Single Data	96
5.4.2.4	Multiple Instruction Multiple Data	96
5.4.3	Interrupt Handling	96
5.4.4	Memory	97
5.4.5	Input and Output	97
5.5	Fault-Tolerant Design	99
5.5.1	Spatial Fault Tolerance	99
5.5.2	Using a Kalman Filter in the Case Study System	99
5.5.3	Checkpoints	102
5.5.4	Recovery Blocks	102
5.5.5	Software Black Boxes	104
5.5.6	N-Version Programming	105
5.5.6.1	Built-In Test Software	105
5.5.6.2	CPU Testing	106
5.5.6.3	Memory Testing	106
5.5.6.4	Other Devices	107
5.6	Exercises	107
Chapter 6	The Software Production Process	109
6.1	Programming Languages	109
6.1.1	Parameter Passing Techniques	110
6.1.2	Call-by-Value and Call-by-Reference	110
6.1.3	Global Variables	110
6.1.4	Recursion	111
6.1.5	Dynamic Memory Allocation	111
6.1.6	Typing	112
6.1.7	Exception Handling	112
6.1.8	Modularity	113
6.1.9	Brief Survey of Languages	114
6.1.9.1	Ada 95	114
6.1.9.2	Assembly Language	115
6.1.9.3	C	115
6.1.9.4	C++	116
6.1.9.5	Fortran	116
6.1.9.6	Java	117
6.2	Writing and Testing Code	118
6.2.1	Example: The Unix/Linux C Compiler	119
6.2.2	Handling Compiler Errors	120
6.2.3	Some Debugging Tips: Unit-Level Testing	120
6.2.4	Extended Syntax and Semantic Checking	120
6.2.5	Symbolic Debugging	121
6.2.6	Test-First Coding	122
6.2.7	Know the Compiler	122
6.3	Coding Standards	123
6.4	Reviews and Audits	124

6.5 Documentation 126

6.6 Exercises..... 127

Chapter 7 Software Measurement and Testing..... 129

7.1 The Role of Metrics 129

7.1.1 Lines of Code..... 129

7.1.2 McCabe’s Metric..... 130

7.1.2.1 Measuring Software Complexity..... 130

7.1.2.2 Determining the Limit on Number of Test Cases 132

7.1.3 Halstead’s Metrics..... 132

7.1.4 Function Points 133

7.1.5 Feature Points..... 137

7.1.6 Metrics for Object-Oriented Software..... 137

7.1.7 Objections to Metrics..... 138

7.2 Faults, Failures, and Bugs..... 138

7.3 The Role of Testing 139

7.4 Testing Techniques..... 139

7.4.1 Unit-Level Testing..... 139

7.4.1.1 Black Box Testing 139

7.4.1.2 White Box Testing..... 141

7.4.2 Testing Object-Oriented Software 142

7.4.3 System-Level Testing..... 142

7.4.3.1 Cleanroom Testing..... 143

7.4.3.2 Stress Testing 143

7.5 Design of Testing Plans 144

7.6 Exercises..... 144

Chapter 8 Hardware–Software Integration and Maintenance 145

8.1 Goals of System Integration 145

8.2 System Unification 145

8.3 System Verification 146

8.4 System Integration Tools 146

8.4.1 Multimeter..... 147

8.4.2 Oscilloscope 147

8.4.3 Logic Analyzer..... 147

8.4.3.1 Timing Instructions..... 148

8.4.3.2 Timing Code 148

8.4.4 In-Circuit Emulator..... 149

8.4.5 Software Simulators 149

8.4.6 Hardware Prototypes..... 149

8.5 Software Integration..... 150

8.5.1 A Simple Integration Strategy 150

8.5.2 Patching 150

8.5.3 The Probe Effect 152

8.6	Postintegration Software Optimization.....	153
8.6.1	CPU Utilization Estimation	153
8.6.2	Execution Time Estimation.....	154
8.6.3	Scaled Numbers	154
8.6.4	Binary Angular Measure.....	155
8.6.5	Look-Up Tables.....	155
8.6.6	Imprecise Computation	157
8.6.7	Optimizing Memory Usage	157
8.7	A Software Reengineering Process Model.....	157
8.8	A Maintenance Process Model	158
8.9	Software Reuse	159
8.9.1	When Not to Reuse.....	160
8.9.2	Achieving Reuse	160
8.9.2.1	In Procedural Languages	161
8.9.2.2	In Object-Oriented Languages	162
8.9.2.3	Pareto's Principle	162
8.10	The Second System Effect.....	162
8.11	Code and Program Maintenance.....	163
8.12	Exercises.....	163

Chapter 9 Management of Software Projects..... 165

9.1	Why Software Project Management?	165
9.2	Software Project Management Themes	166
9.3	General Project Management Basics.....	166
9.3.1	What Does the Project Manager Control?	166
9.4	Software Project Management.....	167
9.5	Managing and Mitigating Risks	168
9.6	Personnel Management	169
9.6.1	The <i>n</i> -Body Problem.....	170
9.6.2	Some Approaches to Leading Teams	171
9.6.2.1	Theory X.....	171
9.6.2.2	Theory Y	171
9.6.2.3	Theory Z	171
9.6.2.4	Theory W	172
9.6.3	Principle-Centered Leadership.....	173
9.6.3.1	Management by Sight.....	173
9.6.3.2	Management by Objectives	173
9.6.4	Dealing with Difficult People	174
9.7	Assessment of Project Personnel.....	174
9.7.1	Skills Testing	174
9.7.2	Recommended Practices	175
9.8	Tracking and Reporting Progress	177
9.8.1	Gantt Chart.....	177
9.8.2	Critical Path Method	178
9.8.3	Program Evaluation and Review Technique.....	179

9.9 Cost Estimation Using COCOMO 180

9.9.1 Basic COCOMO 180

9.9.2 Intermediate and Detailed COCOMO 182

9.9.3 COCOMO II 183

9.10 Exercises..... 183

Glossary 185

References 203

Index..... 209

1 Software Engineering: An Overview

Nothing is more terrible than activity without insight.

Thomas Carlyle

1.1 INTRODUCTION

A 2002 study by the National Institute of Standards Technology (NIST) estimated that software errors cost the U.S. economy \$59.5 billion each year. The report noted that software testing could reduce those costs to about \$22.5 billion. Of the \$59.5 billion, users paid for 64% of the costs and developers 36% (NIST, 2002). What proportion of these findings can be attributed to software in embedded imaging systems or support software related to those systems is unknown. Clearly, however, detection and elimination of software errors are of great concern to users and vendors of software for imaging applications.

A formal engineering framework is essential in the development of reliable, maintainable, and cost-effective software systems. It is the goal of this text, therefore, to show how an engineering framework for the specification, design, developing, testing, maintenance, and documentation of software can ultimately reduce the cost of software and improve its quality.

Many practicing software engineers have little or no formal education in software engineering. While software engineering is a discipline in which practice and experience are important, it is rare that someone who has not studied software engineering will have the skills and knowledge needed to efficiently build industrial-strength, reliable software systems. Shortages of trained software engineers in the 1980s and 1990s led to aggressive hiring of many without formal training in software engineering. This situation is commonly found in companies building imaging applications, where typically engineers were trained in image processing, electrical engineering, physics, optical engineering, and so forth, but not software engineering. While these engineers are perfectly capable of building working systems, unless a deliberate software engineering approach is followed, the cost of development will probably be higher than necessary, and the cost of maintaining the system will certainly be higher than it ought to be.

1.2 A CASE FOR A SOFTWARE ENGINEERING APPROACH TO BUILDING IMAGING SYSTEMS

It is important to heed best practices of software engineering when building imaging systems. Failure to do so can lead to significant problems in the software product. Lack of software requirements, poorly written requirements, failure to design for test, poor design of software, and improper or insufficient testing and documentation are all typical symptoms of poor software engineering. Managers, engineers, and even customers often excuse these practices by citing pressures to market, high cost-to-benefit ratio, and (unfulfilled) promises to go back and fix things later.

These oversights can be exacerbated when the prototype system is commercialized, at which time it is generally cost-prohibitive to go back and correct the problems. Poor software engineering practices early in the project can plague the system long after it is deployed, costing time, money, and reputation. Therefore, making an early commitment to good software engineering practice can pay huge dividends throughout the software life cycle.

There are several thematic problems that can occur within the software life cycle of systems:

1. Failure to document code is pervasive in industry. “My code is self-documenting” is a familiar protest, along with “I had to work hard to design the algorithm; therefore, others should work just as hard to understand it.” The reality is that it simply is not possible to make a nontrivial algorithm, such as those found in imaging applications, easily understandable to every reader of the code.
2. Certainly, code reuse is a wonderful and economical practice when followed correctly. No one wants to rewrite a module that is thought to work perfectly, and many persons feel unable to challenge the assumptions or the reputations of others. When software is reused indiscriminately and without proper testing and documentation, however, numerous problems can occur, and these are very hard to detect.
3. It is an engineer’s nature to anticipate needs and to provide for them. When developing software, it is often believed that more is better — there is always a way to use these unneeded features later. Gold plating — that is, adding unnecessary features — can lead to memory and time overloading problems and should be avoided.
4. Perhaps the greatest of these problems occurs because of the eagerness to bring the software to market. Of all the symptoms of poor software engineering, this is the one that management is most likely to condone.
5. Excess code and overengineering of algorithms is a bad engineering habit. Engineers want things not only to work but also to be a monument of ingenuity. Nevertheless, parsimony and elegance, without sacrificing clarity, are essential requirements for the maintenance of a software system through personnel changes over a long period of time.
6. Failure to test the software sufficiently or to test without documentation can lead to unreliable systems. Insufficient testing can also foster latent

problems that will emerge later in the life of the system when sections of the code are stressed by expert users. Failure to document testing procedures or to develop a coherent test plan can make it difficult or even impossible to test a system as new features are added later.

7. Engineering can get personal. Team members stop talking to one another or set about outright sabotage. This is a management problem, but after all, much of software engineering falls more rightly into the realm of management than engineering.

Of course, these problems and their resolutions will be discussed throughout the text.

1.3 THE ROLE OF THE SOFTWARE ENGINEER

The production of software is a problem-solving activity that is accomplished by modeling. As a problem-solving, modeling discipline, software engineering is a human activity that is based upon previous experience and is subject to human error.

Modeling is a translation activity. The software product concept is translated into a requirements specification. The requirements are converted into a design. The design is then converted into code, which is automatically translated by compilers and assemblers, which produces machine executable code. In each of these translation steps, however, errors are likely to be introduced either by the humans involved or by the tools they use. Thus, the software engineer must strive to identify these likely errors and to avoid or fix them.

Software engineers should also strive to develop code that is built to be tested, designed for reuse, and ready for inevitable change. Anticipation of problems can only come from drawing upon a body of software practice experience that is more than 50 years old.

Software engineers probably spend less than 10% of their time writing code. The other 90% of their time consists of involved activities that are generally more important than writing the code. These activities include:

1. Eliciting requirements
2. Analyzing requirements
3. Writing software requirements documents
4. Building and analyzing prototypes
5. Developing software designs
6. Writing software design documents
7. Researching software engineering techniques or obtaining information about the application domain
8. Developing test strategies and test cases
9. Testing the software and recording the results
10. Isolating problems and solving them
11. Learning to use or installing and configuring new software and hardware tools
12. Writing documentation such as user's manuals

13. Attending meetings with colleagues, customers, and supervisors
14. Archiving software or readying it for distribution

This is only a partial list of software engineering activities. These activities are not necessarily sequential, and they are not all-encompassing. Finally, most of these activities can recur throughout the software life cycle and in each new minor or major software version. Many software engineers specialize in a small subset of these activities, for example, software testing.

1.4 THE NATURE OF SOFTWARE FOR IMAGING SYSTEMS

An imaging software system is a multiple-input multiple-output system involving specialized hardware and software. Often, humans are involved in making inputs, and hence control information to the system. If a system does not provide a direct interface to the human user, however, it is said to be embedded. For example, a vision system used in the navigation of an autonomous vehicle is embedded.

The typical configuration for a control system featuring imaging devices, sensors, and actuators is shown in Figure 1.1.

While it is possible to model the camera input as just another sensor or the display device as simply the target for a set of control signals, it is probable that this simplification is unacceptable in many cases. Sensor inputs and control signals are often a single wire, differing significantly from the information that must be exchanged with imaging devices.

But there are other compelling reasons why imaging systems are different from other kinds of systems, and they are related to the existence of the following volatile components:

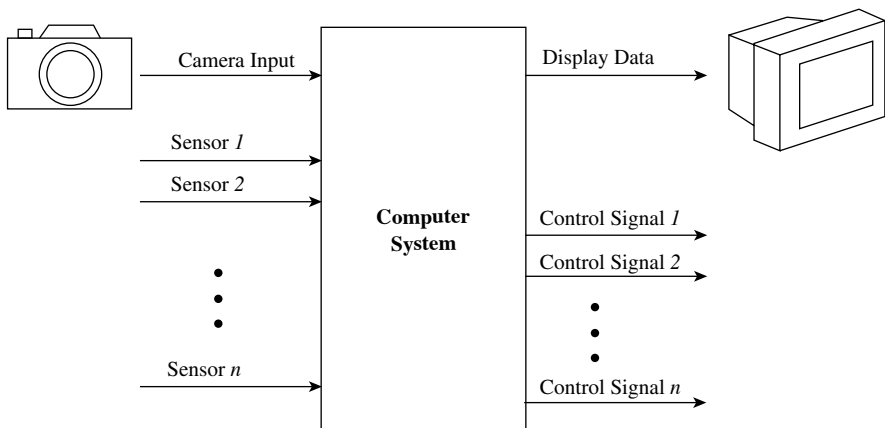


FIGURE 1.1 A typical imaging system. (From Laplante, P. and Neill, C., A Class of Kalman Filters for Real-Time Image Processing, paper presented at Proceedings of the Real-Time Imaging Conference, SPIE, Santa Clara, CA, January 2003, pp. 22–29.)

1. Interchangeable algorithms for critical operations (e.g., compression and decompression, filtration, enhancement, and display)
2. Rapidly changing underlying hardware (e.g., displays, cameras, and storage)
3. Significant hardware and software reuse
4. Legacy and off-the-shelf software that have not been developed using rigorous software engineering approaches

While these characteristics are often found in other kinds of embedded systems, they are pervasive in imaging systems.

1.5 CASE STUDY: A VISUAL INSPECTION SYSTEM

For the purposes of illustration, this text uses a fairly detailed example of an embedded imaging system throughout. The example is adapted from Thomas et al. (1995) and Poling (2002) and is of an industrial automated visual inspection system (VIS). Visual inspection is an interesting case since it represents a simple intuitive example of an embedded system where the temporal performance is dictated by the operating environment rather than by the computer system itself.

A typical setup is shown in Figure 1.2. The system includes an input source, optics, lighting, a part sensor, a frame grabber, a PC platform, inspection software, digital input and output (I/O), and a network connection and some positioning mechanism — either a conveyor system, as shown, or an X-Y positioning table.

The input sources usually consist of one or more cameras and optical systems that take one or more images of the part being inspected. Depending on the application, the cameras can be standard monochrome, RS-170/CCIR, composite color (Y/C), RGB color, nonstandard monochrome (variable scan), progressive scan, line scan, or custom charge-coupled device (CCD) arrays.

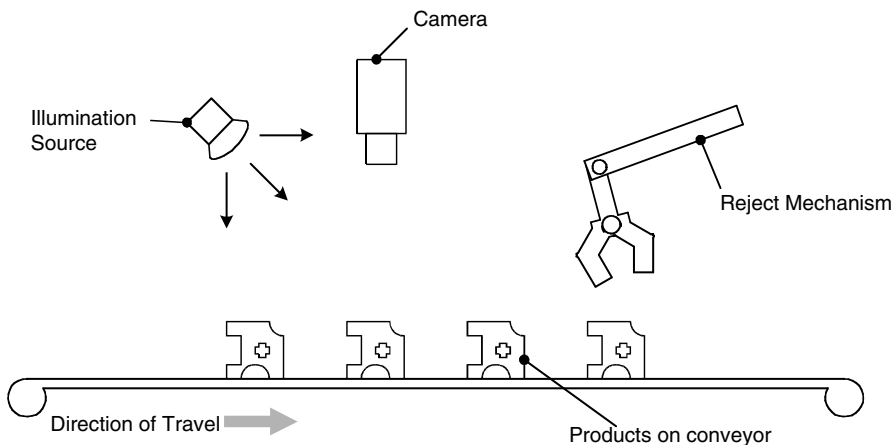


FIGURE 1.2 An automated VIS. (From Laplante, P.A. and Neill, C.J., *J. Electron. Imaging*, 12, 252–262, 2003.)

A frame grabber or video capture card interfaces the imaging units to a host computer. The frame grabber takes the image data provided by the camera(s) in either analog or digital form and converts them to an appropriate format for the host PC. A frame grabber can also provide signals to control camera parameters such as triggering, exposure time, and shutter speed. Frame grabbers come in various configurations to support different camera types as well as different standard computer buses such as Peripheral Component Interconnect (PCI), Industry Standard Architecture (ISA), and Universal Serial Bus (USB).

An X-Y positioning table or conveyor automates the process of acquiring images of multiple samples. The table or conveyor moves a predetermined distance after each image is acquired to properly position the next object or specimen in relation to the camera. The majority of X-Y tables have rapid, smooth movement that minimizes image distortion, although spatial positioning error can be of great concern. Movement along the conveyor, placement of the product on the conveyor, positioning of the camera, and lighting variability can all introduce image capture error, which must be accounted for in any inspection algorithm.

For maximum efficiency, a light barrier or sensor triggers a signal when it senses that a part is in close proximity. The image is then captured. Upon capture, the images are preprocessed and then classified, using an appropriate feature-matching algorithm, as “pass” or “fail” by the system. Defective products are removed from the conveyor by the reject mechanism.

The sampling speed is critical for testing. The amount of time needed to inspect each component or part will present significant real-time image processing concerns such as deadline satisfaction and synchronization. Often PC-based vision systems can inspect 20 to 25 components per second, depending on the measurements and operations required for each part and the PC used.

The inspection system is to be designed to be adaptable to a wide range of applications. Typical applications might include:

- Inspection of freshly picked fruit for defects such as rot or insect damage
- Inspection of light bulbs for soldering defect or glass bulb damage
- Inspection of baked cookies for breakage or over- or underdoneness
- Examination of a continuous skein of textiles for weaving defect
- Examination of bottled pasta sauce for proper portion, sealing, and label placement
- Examination of packaged prepared vegetables for foreign matter
- Inspection of integrated circuit boards for track defects
- X-ray inspection of airline baggage for contraband or dangerous items

This is a very short list that can easily be expanded to include other applications in many other problem domains.

The VIS is to be designed so that to adapt to the different application domains, only a few software units need to be modified. The only potential physical system changes involve the lighting, camera, conveyor speed, and reject mechanism. The software needs to be designed so that users can change the hardware configurations in

a plug-and-play fashion; that is, the software will detect the appropriate hardware and adapt to it. The system also needs to support different pattern recognition algorithms.

Design of such a flexible and robust system requires sound software engineering practices, which will be illustrated in subsequent chapters.

1.6 MISCONCEPTIONS ABOUT SOFTWARE ENGINEERING

There are many misconceptions about what software engineering is and what it is intended to do. Often practitioners of “hard” engineering disciplines such as mechanical or electrical engineering view software engineering dimly or not as an engineering discipline at all. This perception could be partly because there are no fundamental physical laws governing the practice of software engineering, as there are in mechanical, electrical, civil, etc., engineering. More likely, however, the reason that software engineering does not always get the respect it deserves is that it is often not practiced as an engineering discipline and the barriers to “practicing” it are considered to be too low. Anyone can call himself a software engineer if he writes code, but as will be seen, he is not usually practicing software engineering.*

Some misconceptions about software engineering and their brief rebuttals follow:

1. *Software system development is primarily concerned with programming.*

As previously mentioned, 10% or less of the software engineer’s time is spent writing code. Someone who spends the majority of his or her time generating code is more aptly called a programmer. Just as wiring a circuit designed by an electrical engineer is not engineering, writing code designed by a software engineer is not an engineering activity.

2. *Software tools and development methods can solve most or all of the problems pertaining to software engineering.*

This is a dangerous misconception. Tools, software or otherwise, are only as good as the wielder. Bad habits and flawed reasoning can just as easily be amplified by tools as they can be corrected. While software engineering tools are essential and provide significant advantages, to rely on them to remedy process or engineering deficiencies is naïve.

3. *Software productivity is a function of system complexity.*

While it is certainly the case that system complexity can degrade productivity, there are many other factors that affect it, including stability, engineering skill, quality of management, and availability of resources to name a few.

4. *Once software is delivered, the job is finished.*

Of course, this is not true. At the very least, some form of documentation of the end product as well as the process used needs to be written. More likely, the software product will now enter a maintenance mode

* At this writing, one notable exception involves the requirement of professional licensure for those practicing software engineering in New York State.

after delivery, in which it will now experience many recurring life cycles as errors are detected and corrected and features are added.

5. *Errors are an unavoidable side effect of software development.*

While it is unreasonable to expect that all errors can be avoided (as in every discipline involving humans), good software engineering techniques can minimize the number of errors that are delivered to a customer. The attitude that errors are inevitable can be used to excuse sloppiness or complacency, whereas an approach to software engineering that is intended to detect every possible error, no matter how unrealistic this goal may be, will lead to a culture that will encourage engineering rigor and high software quality.

1.7 EXERCISES

- 1.1 Is software engineering really an engineering discipline? Why or why not? If it is not, what is it?
- 1.2 How would you describe the software engineering environment in your current situation? Are there documented procedures for developing requirements specifications, design documents, code, and so forth? Is it a free-for-all?
- 1.3 Consider the last software project in which you were involved. What percentage of time do you estimate was spent in eliciting requirements, writing the requirements specification, performing the software design, writing the software design documentation, writing the code, testing the code, writing end-user documentation, and all other activities?
- 1.4 For the activities and relative percentages you gave in response to Exercise 1.3, why were any of these zeros? Should any of these activities have received more attention? Which activities received too much attention?
- 1.5 The amount of time spent in any of the software activities listed in Section 1.4 can differ, depending on the kind of system involved. Estimate the time (in person-months*) for an imaging system that you are working on or have worked on. If you were the project manager, would any of these numbers have been different? Why?

* Person-month, formerly referred to as a man-month, is the equivalent time worked by one person if he or she had worked solely on the task for a month with standard 40-h workweeks.

2 Imaging Software and Its Properties

I often say that when you can measure what you are speaking about and express it in numbers you know something about it; but when you cannot express it in numbers your knowledge is a meager and unsatisfactory kind: it may be the beginning of knowledge but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be.

Lord Kelvin

2.1 CLASSIFICATION OF SOFTWARE QUALITIES

Software can be characterized by any of a number of qualities. External qualities are those that are visible, such as usability and reliability, and are of concern to the end user. Internal qualities are those that may not be necessarily visible to the user, but help the developers to achieve improvement in external qualities. For example, good requirements and design documentation might not be seen by the typical user, but these are necessary to achieve improvement in most of the external qualities. A specific distinction between whether a particular quality is external or internal is not often made because these qualities are so closely tied. Moreover, the distinction is largely a function of the software itself and the kind of user involved.

While it is helpful to describe these qualities, it is equally desirable to quantify them. Quantification of these characteristics of software is essential in enabling users and designers to talk succinctly about the product and for software process control and project management.

Much of the upcoming discussion has been adapted from the excellent section on software properties found in Tucker (1996).

2.1.1 RELIABILITY

Reliability is a measure of whether a user can depend on the software. This notion can be informally defined in a number of ways. For example, one definition might be “a system that a user can depend on.” Other loose characterizations of a reliable software system include:

- The system stands the test of time.
- There is an absence of known catastrophic errors, that is, errors that render the system useless.

- The system recovers “gracefully” from errors.
- The software is robust.

For imaging systems, other informal characterizations of reliability might include:

- Downtime is below a certain threshold.
- The accuracy of the system is within a certain tolerance.
- Real-time performance requirements are met consistently.

While all of these informal characteristics are desirable in any imaging system, they are difficult to measure. Moreover, they are not truly measures of reliability, but of other attributes of the software.

There is specialized literature on software reliability that defines this quality in terms of statistical behavior, that is, the probability that the software will operate as expected over a specified time interval. These characterizations generally take the following approach. Let S be a software system, and let T be the time of system failure. Then the reliability of S at time t , denoted $r(t)$, is the probability that T is greater than t ; that is,

$$r(t) = P(T > t) \quad (2.1)$$

This is the probability that a software system will operate without failure for a specified period of time.

Thus, a system with reliability function $r(t) = 1$ will never fail. However, it is unrealistic to have such expectations. Instead, some reasonable goal should be set, for example, in the visual inspection system, that the failure probability be no more than 10^{-9} per hour. This represents a reliability function of $r(t) = (0.99999999)^t$ with t in hours. Note that as $t \rightarrow \infty$, $r(t) \rightarrow 0$.

Another way to characterize software reliability is in terms of a real-valued failure function. One failure function uses an exponential distribution where the abscissa is time and the ordinate represents the expected failure intensity at that time (Equation 2.2).

$$f(t) = \lambda e^{-\lambda t} \quad t \geq 0 \quad (2.2)$$

Here the failure intensity is initially high, as would be expected in new software as faults are detected during testing. However, the number of failures would be expected to decrease with time, presumably as failures are uncovered and repaired (Figure 2.1). The factor λ is a system-dependent parameter.

A second failure model is given by the “bathtub curve” shown in Figure 2.2. Brooks (1995) notes that while this curve is often used to describe the failure function of hardware components, it might also be used to describe the number of errors found in a certain release of a software product.

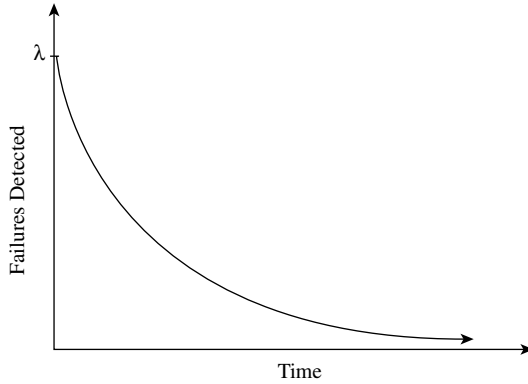


FIGURE 2.1 An exponential model of failure represented by the failure function $f(t) = \lambda e^{-\lambda t}$, $t \geq 0$. λ is a system-dependent parameter.

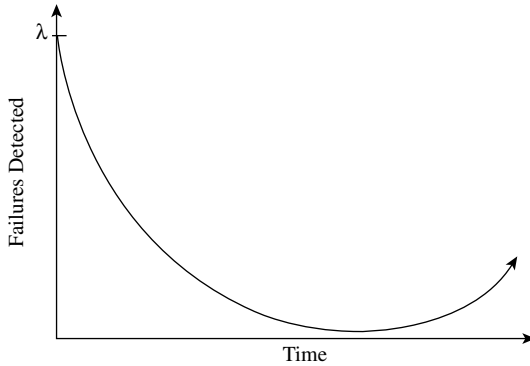


FIGURE 2.2 A software failure function represented by the bathtub curve.

The interpretation of this failure function is clear for hardware: a certain number of product units will fail early due to manufacturing defects. Later, the failure intensity will increase as the hardware ages and wears out. But software does not wear out. If systems seem to fail according to the bathtub curve, then there has to be some plausible explanation.

It is clear that a large number of errors will be found in a particular software product early on, just as in the exponential model of failure. But why would the failure intensity increase much later? There are at least three possible explanations. The first is that the errors are due to the effects of patching the software for various reasons. The second reason is that late software failures are really due to failure of the underlying hardware. Finally, additional failures could appear because of the increased stress on the software by expert users. That is, as users master the software and begin to expose and strain advanced features, it is possible that certain poorly tested functionality of the software is beginning to be used.

Often the traditional quality measures of mean time to first failure (MTFF) or mean time between failures (MTBF) are used to stipulate reliability in the software

requirements specification. This approach to failure definition places great importance on the effective elicitation (gathering) and specification of functional requirements, because the requirements define the software failure.

2.1.2 CORRECTNESS

Software correctness is closely related to reliability, and the terms are often used interchangeably. The main difference is that minor deviation from the requirements is strictly considered a failure, and hence means that the software is incorrect. However, a system may still be deemed reliable if only minor deviations from the requirements are experienced.

Correctness is measured in terms of number of failures detected over time.

2.1.3 PERFORMANCE

Performance is a measure of some required behavior. For example, an imaging system might be required to display a filtered image at a rate of 30 frames per second. A photo reproduction system might be required to digitize, clean, and output color copies at a rate of one every 2 sec.

Most imaging systems are real-time systems; that is, performance satisfaction is based on both the correctness of the outputs and the timeliness of those outputs. Hard real-time systems are those in which missing even a single deadline will lead to total system failure. Firm real-time systems can tolerate a few missed deadlines, while in soft real-time systems, missing deadlines generally leads only to performance degradation.

Imaging systems can fall into any of these categories. For example, if the visual inspection system is involved in scanning airline baggage for bombs, then clearly it must not miss any deadline (e.g., it cannot allow a bag to pass through without analyzing it). On the other hand, a system that is inspecting fresh fruit can probably allow an occasional damaged piece to get through, but not too many, and hence is firm real time. Finally, if the system is inspecting cookies for breakage, then missing too many deadlines might mean that too many broken cookies get shipped to the customer. In this case, missing deadlines leads to performance degradation (though if the problem persists, customers will not buy the cookies anymore, which could lead to failure of the company).

The performance of real-time imaging systems is still largely based on three trade-off problems: performance vs. resolution, performance vs. storage, and jitter (synchronization error). Because each problem is based on a trade-off, these are referred to collectively as the real-time imaging dilemma (Laplante et al., 1996a). Fundamentally, the real-time imaging dilemma is not due to lack of raw computing power, but rather to difficulties of making performance guarantees. The challenge then, in real-time imaging systems, is to be able to specify, calculate, and guarantee deadlines analytically early in the software life cycle — not after the fact, when trial and error is the only technique available.

One method of measuring performance is based on mathematical or algorithmic complexity. Another approach involves directly timing the behavior of the completed

system with logic analyzers and similar tools. Finally, a simulation of the finished system might be built with the specific purpose of estimating performance.

2.1.4 USABILITY

Often referred to as ease of use, or user-friendliness, usability is a measure of how easy the software is for humans to use. This quantity is an elusive one. Properties that make an application user-friendly to novice users are often different from those desired by expert users or the software designers. Use of prototyping can increase the usability of a software system because, for example, interfaces can be built and tested by the user.

Usability is difficult to quantify. However, informal feedback can be used, as well as user feedback from surveys, and problem reports can be used in most cases.

2.1.5 INTEROPERABILITY

This quality refers to the ability of the software system to coexist and cooperate with other systems. For example, in imaging systems the software must be able to communicate with various devices using standard bus structures and protocols.

A concept related to interoperability is that of an open system. An open system is an extensible collection of independently written applications that cooperate to function as an integrated system. Open systems differ from open source code, which is source code that is made available to the user community for moderate improvement and correction.

An open system allows the addition of new functionality by independent organizations through the use of interfaces whose characteristics are published. Any applications developer can then take advantage of these interfaces, and thereby create software that can communicate using the interface. Open systems allow different applications written by different organizations to interoperate.

Interoperability can be measured in terms of compliance with open system standards.

2.1.6 MAINTAINABILITY

Anticipation of change is a general principle that should guide the software engineer. A software system in which changes are relatively easy to make has a high level of maintainability. In the long run, design for change will significantly lower software life cycle costs and lead to an enhanced reputation for the software engineer, the software product, and the company.

Maintainability can be decomposed into two contributing properties: evolvability and repairability. Evolvability is a measure of how easily the system can be changed to accommodate new features or modification of existing features. Software is repairable if it allows for the fixing of defects.

Measuring these qualities of software is not always easy and often is based on anecdotal observation only. This means that changes and the cost of making them are tracked over time. Collecting this data has a twofold purpose. First, the costs of maintenance can be compared to those of similar systems for benchmarking and project management purposes. Second, the information can provide experiential

learning that will help to improve the overall software production process and the skills of the software engineers.

2.1.7 PORTABILITY

Software is portable if it can easily run in different environments. The term *environment* refers to the hardware on which the system runs, operating system, or other software with which the system is expected to interact. Because of the specialized hardware with which imaging systems interact, special care must be taken in making them portable.

Portability is achieved through a deliberate design strategy in which hardware-dependent code is confined to the fewest code units possible. This strategy can be achieved using either object-oriented or procedural programming languages and through object-oriented or structured approaches. Both of these will be discussed throughout the text.

Portability is difficult to measure, other than through anecdotal observation. Person-months required to perform the port are the standard measure of this property.

2.1.8 VERIFIABILITY

A software system is verifiable if its properties, including all of those previously introduced, can be verified easily.

One common technique for increasing verifiability is through the insertion of software code that is intended to monitor various qualities such as performance or correctness. Modular design, rigorous software engineering practices, and the effective use of an appropriate programming language can also contribute to verifiability.

2.1.9 SUMMARY OF SOFTWARE PROPERTIES AND ASSOCIATED METRICS

It has been emphasized so far that measurement of the properties of software is essential throughout the software life cycle. A summary of the software qualities just discussed and possible ways to measure them are shown in Table 2.1.

TABLE 2.1
Software Properties and the Means for Measuring Them

Software Quality	Possible Measurement Approach
Correctness	Probabilistic measures, MTBF, MTFF
Interoperability	Compliance with open standards
Maintainability	Anecdotal observation of resources spent
Performance	Algorithmic complexity analysis, direct measurement, simulation
Portability	Anecdotal observation
Reliability	Probabilistic measures, MTBF, MTFF, heuristic measures
Usability	User feedback from surveys and problem reports
Verifiability	Software monitors

2.2 BASIC SOFTWARE ENGINEERING PRINCIPLES

Software engineering has been criticized for not having the same kind of underlying rigor as other engineering disciplines. And while it may be true that there are few formulaic principles, there are many fundamental rules that form the basis of sound software engineering practice. The following sections describe the most general and prevalent of these.

2.2.1 RIGOR AND FORMALITY

Because software development is a creative activity, there is an inherent tendency toward informal *ad hoc* techniques in software specification, design, and coding. But the informal approach is contrary to good software engineering practice.

Rigor in software engineering requires the use of mathematical techniques. Formality is a higher form of rigor in which precise engineering approaches are used.

For example, imaging systems require the use of rigorous mathematical specification in the description of image acquisition, filtering, enhancement, etc. But the existence of mathematical equations in the requirements or design does not imply an overall formal software engineering approach. In the case of the imaging system, formality further requires that there be an underlying algorithmic approach to the specification, design, coding, and documentation of the software.

2.2.2 SEPARATION OF CONCERNS

Separation of concerns is a kind of divide-and-conquer strategy that software engineers use. There are various ways in which separation of concerns can be achieved. In terms of software design and coding, it is found in modularization of code and in object-oriented design. There may be separation in time, for example, developing a schedule for a collection of periodic computing tasks with different periods.

Yet another way of separating concerns is in dealing with qualities. For example, it may be helpful to address the fault tolerance of a system while ignoring other qualities. However, it must be remembered that many of the qualities of software are interrelated, and it is generally impossible to affect one without affecting the other, perhaps adversely.

2.2.3 MODULARITY

Some separation of concerns can be achieved in software through modular design. Modular design involves the decomposition of software behavior in encapsulated software units and can be achieved in either object-oriented or procedurally oriented programming languages.

Modularity is achieved by grouping together logically related elements, such as statements, procedures, variable declarations, object attributes, and so on, in increasingly fine-grained level of detail (Figure 2.3).

The main objective in seeking modularity is to foster high cohesion and low coupling. With respect to the code units, cohesion represents intramodule connectivity and coupling represents intermodule connectivity. Coupling and cohesion can

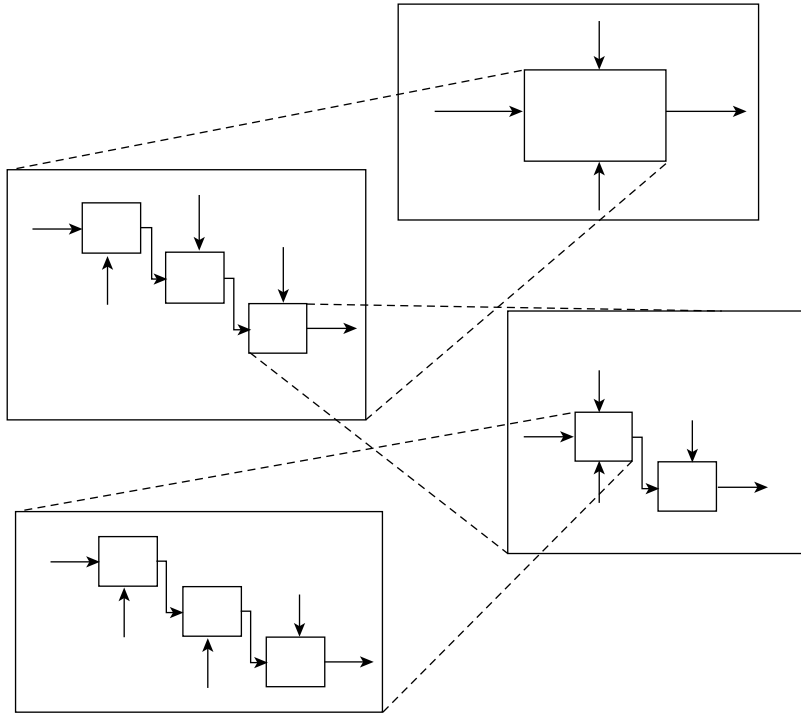


FIGURE 2.3 Modular decomposition of code units. The arrows represent inputs and outputs in the procedural paradigm. In the object-oriented paradigm they represent associations. The boxes represent encapsulated data and procedures in the procedural paradigm. In the object-oriented paradigm they represent classes.

be illustrated informally, as in Figure 2.4, which shows software structures with high cohesion and low coupling and with low cohesion and high coupling. The inside squares represent statements or data; the arcs indicate functional dependency.

Cohesion relates to the relationship of the elements of a module. High cohesion implies that each module represents a single part of the problem solution. Therefore, if the system ever needs modification, then the part that needs to be modified exists in a single place, making it easier to change.

Constantine and Yourdon identified seven levels of cohesion in order of strength (Pressman, 2000):

1. Coincidental — Parts of module are not related, but simply bundled into a single module.
2. Logical — Parts that perform similar tasks are put together in a module.
3. Temporal — Tasks that execute within the same time span are brought together.
4. Procedural — The elements of a module make up a single control sequence.

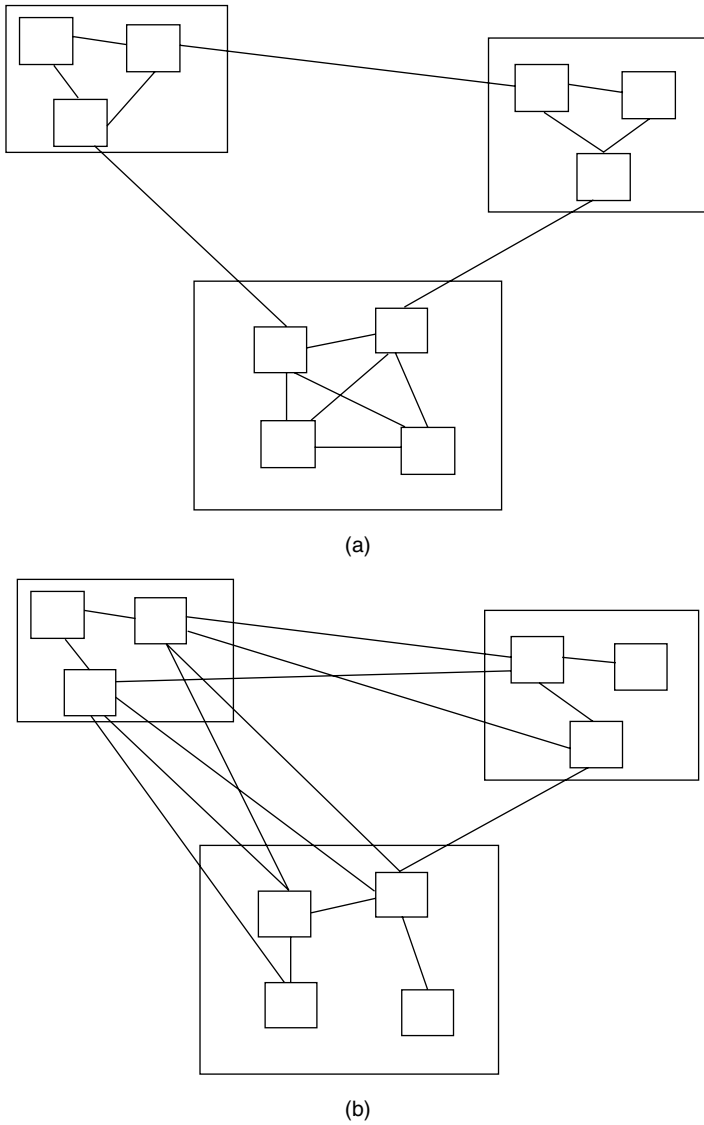


FIGURE 2.4 Software structures with high cohesion and low coupling (a) and low cohesion and high coupling (b). The inside squares represent statements or data; the arcs indicate functional dependency.

5. Communicational — All elements of a module act on the same area of a data structure.
6. Sequential — The output of one part in a module serves as the input for some other part.
7. Functional — Each part of the module is necessary for the execution of a single function.

Coupling relates to the relationships between the modules themselves. There is great benefit in reducing coupling so that changes made to one code unit do not propagate to others; that is, they are hidden. This principle of information hiding, also known as Parnas partitioning, is the cornerstone of all software design (Parnas, 1979). Low coupling limits the effects of errors in a module (lower ripple effect) and reduces the likelihood of data integrity problems. In some cases, however, high coupling due to control structures may be necessary. For example, in most graphical user interfaces, control coupling is unavoidable, and indeed desirable.

Coupling has also been characterized in increasing levels as:

1. No direct coupling — All modules are completely unrelated.
2. Data — When all arguments are homogeneous data items; that is, every argument is either a simple argument or data structure in which all elements are used by the called module.
3. Stamp — When a data structure is passed from one module to another, but that module operates on only some of the data elements of the structure.
4. Control — One module passes an element of control to another; that is, one module explicitly controls the logic of the other.
5. Common — If two modules both have access to the same global data.
6. Content — One module directly references the contents of another.

To further illustrate both coupling and cohesion, consider the class structure for a widely used commercial imaging application program interface (API) package, depicted in Figure 2.5.

The class diagram was obtained through design recovery. The class names are not readable in the figure, but it is not the figure's intention to identify the software. Rather, the point is to illustrate the fact that there is a high level of coupling and low cohesion in the structure.

This design would benefit from refactoring, that is, performing a behavior-preserving code transformation, which would achieve higher cohesion and lower coupling.

2.2.4 ANTICIPATION OF CHANGE

It has been mentioned that software products are subject to frequent change either to support new hardware or software requirements or to repair defects. A high maintainability level of the software product is one of the hallmarks of outstanding commercial software.

Imaging engineers know that their systems are frequently subject to changes in hardware, algorithms, and even application. Therefore, these systems must be designed in such a way so as to facilitate changes without degrading the other desirable properties of the software.

Anticipation of change can be achieved in the software design through appropriate techniques, through the adoption of an appropriate software life-cycle model and associated methodologies, and through appropriate management practices.

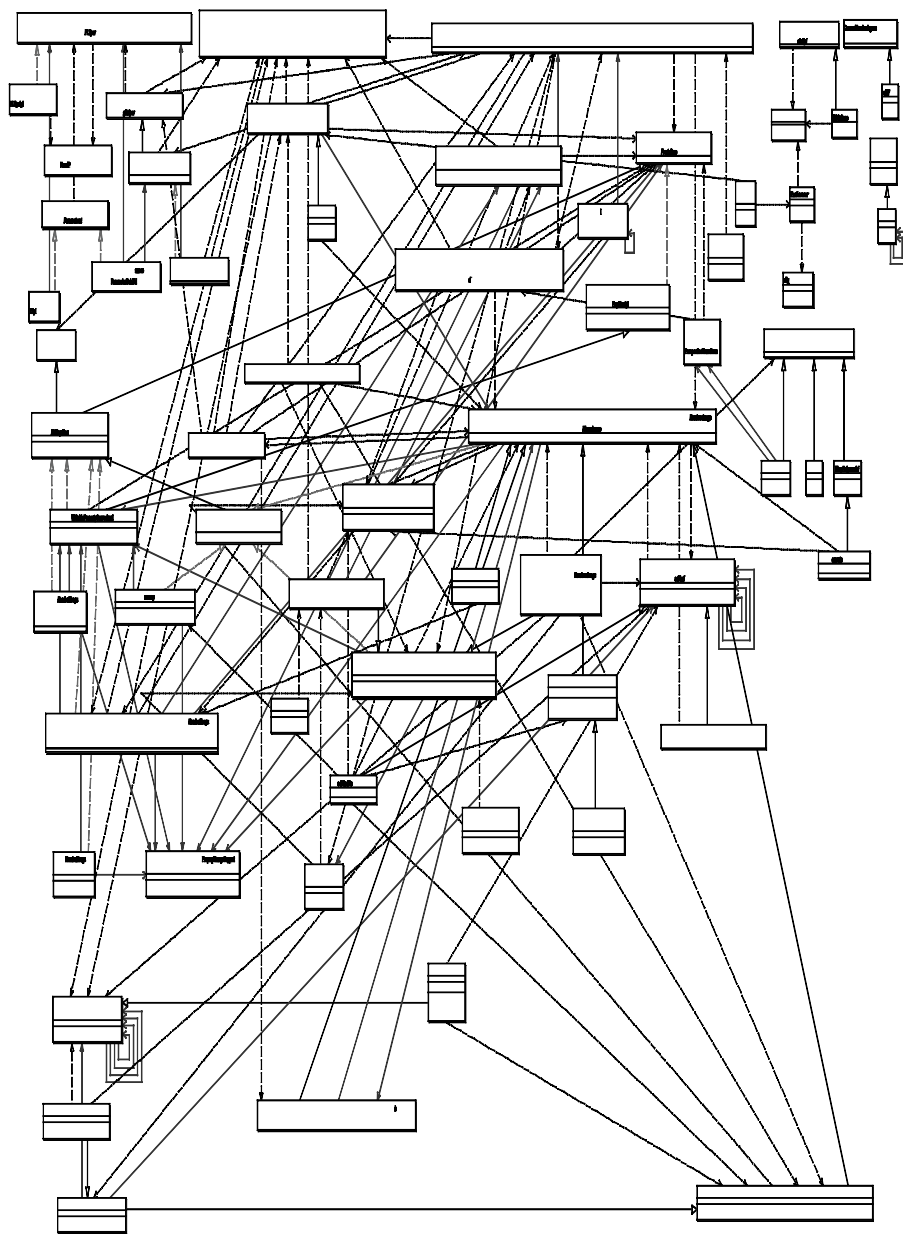


FIGURE 2.5 The class structure for the API of a widely deployed imaging package.

2.2.5 GENERALITY

In solving a problem, the principle of generality can be stated as the intent to look for the more general problem that may be hidden behind it. In an obvious example, designing the visual inspection system for a specific application is less general than designing it to be adaptable to a wide range of applications.

Generality can be achieved through a number of approaches associated with procedural and object-oriented paradigms. For example, in procedural languages, Parnas' information hiding can be used. In the object orientation, the Liskov substitution principle can be used. This approach will be discussed later.

Although generalized solutions may be more costly in terms of the problem at hand, in the long run, the costs of a generalized solution may be worthwhile.

2.2.6 INCREMENTALITY

Incrementality involves a software approach in which progressively larger increments of the desired product are developed. Each increment provides additional functionality, which brings the product closer to the final one. Each increment also offers an opportunity for demonstration of the product to the customer for the purposes of gathering requirements and refining the look and feel of the product.

2.2.7 TRACEABILITY

Traceability is concerned with the relationships between requirements, their sources, and the system design. Regardless of the process model, documentation and code traceability is paramount. A high level of traceability ensures that the software requirements flow down through the design and code, and then can be traced back up at every stage of the process. This would ensure, for example, that a coding decision can be traced back to a design decision to satisfy a corresponding requirement.

Traceability is particularly important in imaging systems because often design and coding decisions are made to satisfy hardware constraints that may not be easily associated with a requirement. Failure to provide a traceable path from such decisions through the requirements can lead to difficulties in extending and maintaining the system.

Generally, traceability can be obtained by providing links between all documentation and the software code. In particular, there should be the following links:

- From requirements to stakeholders who proposed these requirements
- Between dependent requirements
- From the requirements to the design
- From the design to the relevant code segments
- From requirements to the test plan
- From the test plan to test cases

One way to achieve these links is through the use of an appropriate numbering system throughout the documentation. For example, a requirement numbered 3.2.2.1 would be linked to a design element with a similar number (the numbers do not

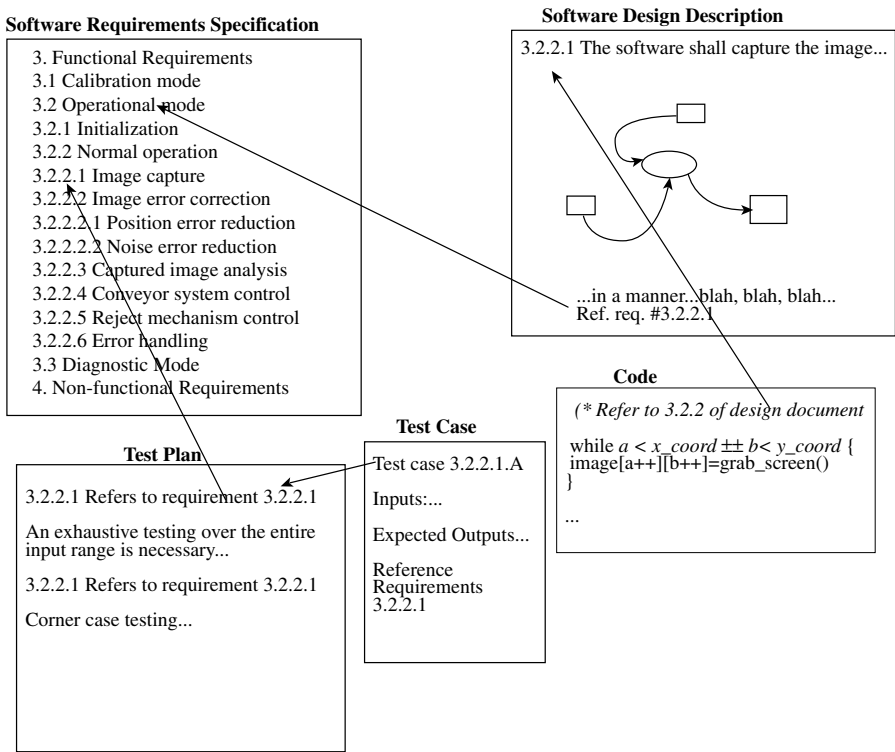


FIGURE 2.6 Linkages between software documentation and code. In this case the links are achieved through both similarity in numbering and specific reference to the related item in the appropriate document.

have to be the same so long as the annotation in the document provides traceability). These linkages are depicted in Figure 2.6. Although the documents shown in the figure have not been introduced yet, the point to be made is that the documents are all connected through appropriate referencing and notation.

Figure 2.6 is simply a graphical representation of the traceable links. In practice, a traceability matrix is constructed to help cross-reference documentation and code elements (Table 2.2).

The matrix is constructed by listing the relevant software documents and the code unit as columns, and then each software requirement in the rows.

Constructing the matrix in a spreadsheet software package allows for providing multiple matrices sorted and cross-referenced by each column as needed. For example, a traceability matrix sorted by test case number would be an appropriate appendix to the test plan.

The traceability matrices are updated at each step in the software life cycle. For example, the column for the code unit names (e.g., procedure names, object class) would not be added until after the code is developed.

Finally, a way to foster traceability between code units is through the use of data dictionaries, which are described later.

TABLE 2.2
A Traceability Matrix Corresponding to Figure 2.6 Sorted by Requirement Number

Requirement Number	Software Design Document Reference Number	Test Plan Reference Number	Code Unit Name	Test Case Number
3.1.1.1	3.1.1.1 3.2.4	3.1.1.1	Simple_fun	3.1.1.A
		3.2.4.1		3.1.1.B
		3.2.4.3		
3.1.1.2	3.1.1.2	3.1.1.2	Kalman_filter	3.1.1.A 3.1.1.B
3.1.1.3	3.1.1.3	3.1.1.3	Under_bar	3.1.1.A
				3.1.1.B
				3.1.1.C

2.3 EXERCISES

- 2.1 For the visual inspection system, select one of the possible specific applications mentioned (for example, inspection of fruit). What might be reasonable expectations for each of the software qualities discussed? Try to answer this question qualitatively, or use one of the metrics mentioned if you are familiar with them.
- 2.2 For a software system with which you are familiar, which of the software qualities were included in the software requirements specification document? Which ones should have been included?
- 2.3 Discuss the cohesiveness and coupling for an imaging system with which you are familiar. Do you think it would be possible to decrease coupling or increase cohesion through refactoring (behavior-preserving code transformation)?
- 2.4 For the software system you are working on (or have worked on), examine the traceability matrices (or equivalent) that might exist. Discuss whether they provide sufficient traceability and how they might be improved. If no traceability matrix exists, design one (no need to complete the entries in the matrix, however).
- 2.5 Using a spreadsheet program, design a set of traceability matrices for the visual inspection system, assuming that the column entries will be those shown in Table 2.2. The result should be five different traceability matrices (one per column), which are appropriately linked.

3 Software Process and Life Cycle Models

We think in generalities, we live in detail.

Alfred North Whitehead

3.1 SOFTWARE PROCESSES AND METHODOLOGIES

A software process is a model that describes an approach to the production and evolution of software. As with any model, a process model is an abstraction. But in this case, the model depicts the process of translation — from system concept, to requirements specification, to a design, to code, and then finally, via compilation and assembly, to the stored program form. Hence, a good process model will help to minimize the problems associated with each translation.

A software process also provides a common software development approach both within a project and across projects. The process allows for productivity improvements and provides for a common culture, a common language, and common skills among organizational members. These benefits foster a high level of traceability and efficient communication throughout the project.

A software methodology is not the same as a software process. A software process is, in essence, the “what” of the software product life cycle. The process identifies and determines the order of phases within the life cycle. It establishes phase transition criteria and indicates what is to be done in each phase and when to stop.

On the other hand, the methodology describes the “how.” It identifies how to perform activities for each period, how to represent the activities and products,* and how to generate products.

3.2 SOFTWARE LIFE CYCLE MODELS

Every software process is an abstraction, but in any case, the activities of the process need to be mapped to a life cycle model. The activities within these life cycle models must be time independent, sequential, and nonoverlapping. There are a variety of software life cycle models, and any of these can be applied to imaging systems.

The following sections describe some of the more widely recognized software life cycle models. While significantly more time is focused on the activities of the waterfall model, most of these activities also occur in the other life cycle models.

* The term *artifact* is sometimes used for software or a software-related product such as documentation.

Indeed, it can be argued that most other life cycle models either are refinements of the waterfall model or contain one or more waterfall life cycles within.

3.2.1 THE WATERFALL MODEL

The terms *waterfall*, *conventional*, and *linear sequential* are used to describe a sequential model of nonoverlapping and distinctive activities related to software development. Collectively, the periods in which these activities occur are often referred to as phases or stages. The number of phases differs between variants of the model. While simplistic and dating back at least 30 years, the waterfall model is still popular. One survey, for example, showed that 35% of companies still used a waterfall model (Laplane et al., 2002e).

As an example of a waterfall model, consider a software development effort with activity periods that occur in the following sequence:

1. Concept definition
2. Requirements specification
3. Design specification
4. Code development
5. Testing
6. Maintenance

The waterfall representation of this sequence is shown in Figure 3.1. Table 3.1 summarizes the activities in each period and the main artifacts of these activities.

The next sections discuss each of these activity periods in some detail, while the documentation is discussed in Chapter 6.

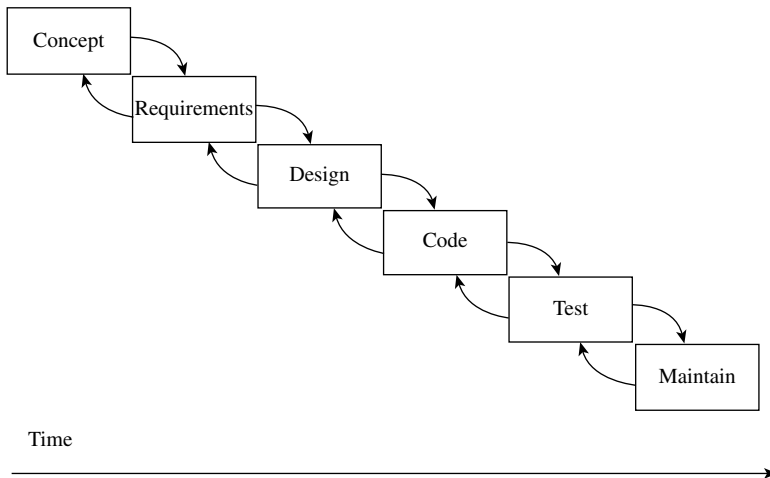


FIGURE 3.1 A waterfall life cycle model. The forward arcs represent time-sequential activities, and the reverse arcs represent backtracking.

TABLE 3.1
Phases of a Waterfall Software Life Cycle with Associated Activities and Artifacts

Phase	Activity	Output
Concept	Define project goals	White paper
Requirements	Decide what the software must do	Software requirements specification
Design	Show how the software will meet the requirements	Software design description
Development	Build the system	Program code
Test	Demonstrate requirement satisfaction	Test reports
Maintenance	Maintain system	Change requests, reports

3.2.1.1 Software Conception

The software conception activities include determination of the software project’s needs and overall goals. These activities are driven by management directives, customer input, technology changes, and marketing decisions.

At the onset of the phase, no formal requirements are written, generally no decisions about hardware or software environments are made, and budgets and schedules cannot be set. In other words, only the features of the software product and possibly the feasibility of testing them are discussed. Usually, no documentation other than internal feasibility studies, white papers, or memos are generated.

Some variants of the waterfall model do not explicitly include a conception period because the activity was either incorporated into the requirements definition or not thought to be part of the software project at all. Nonetheless, the concept activity does occur in every software product, even if it is implicit.

3.2.1.2 Requirements Specification

The requirements specification phase includes the main activity of writing the software requirements specification (SRS). This document is prepared by a customer or in conjunction with a customer through requirements elicitation, that is, the process of determining the customer’s needs. The SRS contains exacting information about what the software product is to do, including all behavioral aspects, expected outputs, and performance goals and deadlines. Ideally, little or no information on how these requirements are to be met is stipulated, but a production schedule and budget are included.

From a testing perspective, it is during the requirements specification activity that test requirements are determined and committed to a formal test plan. The test plan is used as the blueprint for the creation of test cases used in the testing phase, which is discussed later in the text.

The requirements specification phase can and often does occur in parallel with product conception, and as mentioned before, they are often not treated as distinct. It can be argued that the two are separate; however, the requirements generated

during conceptualization are not binding, whereas those determined in the requirements specification phase are (or should be) binding. This rather subtle difference is important from a testing perspective because the SRS represents a binding contract, and hence the criteria for product acceptance. Conversely, ideas generated during system conceptualization may change, and therefore are not yet binding.

3.2.1.3 Software Design

The set of activities associated with the software design phase is characterized by the conversion of the SRS into a software design description (SDD), also known as a detailed design specification (DDS) or similar name. In the waterfall model, preparation of the SDD cannot begin until the SRS is completed. Techniques for software design are discussed in the following chapter.

The main activity of software design is to develop a coherent, well-organized representation of the software system suitable to guide software development. In essence, the design maps the “what” from the SRS to the “how” of the SDD.

The format of the detailed design document can be in accordance with an accepted standard such as MIL-STD-498, Institute of Electrical and Electronics Engineers (IEEE) Standard 830, or a proprietary format. In any case, adherence to a strict design procedure is critical for embedded imaging systems.

Certain testing activities occur concurrently with the preparation of the SDD. These include the development of a set of test cases based on the test plan. Techniques for developing test cases for imaging systems are discussed later.

Often during the software design phase, problems in the SRS are identified. These problems may include conflicts, redundancies, or requirements that cannot be met with current technology. In imaging systems, the most typical problems are related to deadline satisfaction.

Usually, problems such as these require changes to the SRS or the granting of exemptions from the requirements in question. In any case, the problem resolution shows up as a specific directive in the SDD.

3.2.1.4 Software Development

The software development phase involves the production of the software code based on the design using best practices. In theory, this activity should only begin when the design is complete. In practice, however, there is usually an overlap between development and design.

Additionally, in this phase the test team can build the test cases specified in the design phase in some automated form. This approach guarantees the efficacy of the tests and facilitates repeat testing.

The software development phase ends when all software units have been coded, unit tested, and integrated, and have passed the integration testing specified by the software designers.

Management of the software development phase can be greatly improved with version control or configuration management software, which regulates access to the various components of the system from the software library. Version control prevents multiple accesses to the same source code, provides mechanisms for track-

ing changes, and preserves version integrity. In the long run, it increases overall system reliability.

Computer-aided software engineering (CASE) tools can also assist with this and earlier phases of the software life cycle. In imaging systems, commercial tools should be used as appropriate, provided that they incorporate temporal modeling aspects. These problems will be discussed later.

3.2.1.5 Testing

Although ongoing testing is an implicit part of the waterfall model, the model also includes an explicit testing phase. These testing activities (often called acceptance testing to differentiate them from code unit testing) begin when the software development phase ends. During the explicit testing phase, the software is confronted with a set of test cases (module and system level) developed in parallel with the software and documented in a software test requirements specification (STRS). Acceptance or rejection of the software is based on whether it meets the requirements defined in the SRS using tests and criteria set forth in the STRS.

Testing of imaging systems can be more difficult if they involve temporal correctness and deadline satisfaction. In addition, intrusive testing can alter the timing characteristics of the system, making even passing test results uncertain. Finally, testing that requires real-world inputs is not easily done. For example, an autonomous vehicle that navigates based on camera input cannot be tested with live data in the laboratory. Testing of software will be discussed later.

The testing phase ends when either the criteria established in the STRS are satisfied or failure to meet the criteria forces requirements modification, design alteration, or code repair. Regardless of the outcome, one or more test reports are prepared that summarize the conduct and results of the testing.

3.2.1.6 Software Maintenance

The software maintenance phase activities generally consist of a series of reengineering processes to prolong the life of the system. Maintenance activities can be adaptive, which results from external changes to which the system must respond; corrective, which involves maintenance to correct errors; or perfective, which is all other maintenance, including user enhancements, documentation changes, efficiency improvements, and so on.

Maintenance corrections are usually handled by making a software change and then performing regression testing. Another approach is to collect a set of changes and then regression test them against the last set of changes. The maintenance activity only ends when the product is no longer supported.

In some cases, the maintenance phase is not incorporated into the life cycle model, but instead treated as a series of new software products, each with its own waterfall life cycle.

3.2.1.7 Backtracking Transitions in the Waterfall Life Cycle

Although Figure 3.1 implies that the phases in the waterfall phases occur in forward sequence, common sense suggests that this is not always to be expected. In fact,

backtracking transitions are likely to occur. For example, new features, lack of sufficient technology, or other factors force reconsideration of the system purpose, or redesign may result in a return to the requirements phase during design. Similarly, a transition from the programming phase back to the design phase might occur due to a feature that cannot be implemented or caused by an undesirable performance result. This in turn may necessitate redesign, new requirements, or elimination of the feature. Finally, a transition from the testing phase to the programming or design phase may occur due to an error detected during testing. Depending on the severity of the error, solution may require reprogramming, redesign, modification of requirements, or reconsideration of the system goals.

3.2.1.8 Waterfall Model Summary

The waterfall model is quite simplistic. However, Parnas and Clements (1986) have suggested that after the project has been completed, tested, and delivered, users can “cover their tracks” by modifying the documentation so that it appears that a deliberate methodology was used. Hence, when the sequence of the waterfall model cannot be followed strictly, at least the documentation should suggest that it was followed in that sequence.

While this kind of practice might appear disingenuous, the benefit is that a traceable history is established between each program feature and the requirement driving that feature. This promotes a maintainable, robust, and reliable product and, in particular, one where decisions related to timing requirements are well documented. It does indicate, however, that perhaps the process used was a reactive one and not part of a planned strategy.

3.2.2 V MODEL

The V model is a variant of the waterfall model that represents a tacit recognition that there are testing activities that occur throughout the waterfall software life cycle model — and not just during the software testing period. These concurrent activities, depicted in Figure 3.2, are described alongside the activities occurring in the waterfall model.

For example, during requirements specification, the requirements are being evaluated for testability, and a test requirements specification may be written. This document would describe the strategy necessary for testing the requirements. Similarly, during the design phase, a corresponding design of test cases is being performed. While the software is being coded and unit tested, the test cases are developed and automated. The test life cycle converges with the software development life cycle during acceptance testing.

The point, of course, is that testing is a full life cycle activity and that it is important to constantly consider the testability of any software requirement (e.g., deadlines) and to design to allow for such testability.

3.2.3 THE SPIRAL MODEL

The spiral model, suggested by Boehm (1988), recognizes that the waterfall model is not a realistic representation, nor is it necessarily a healthy one. Instead, the spiral

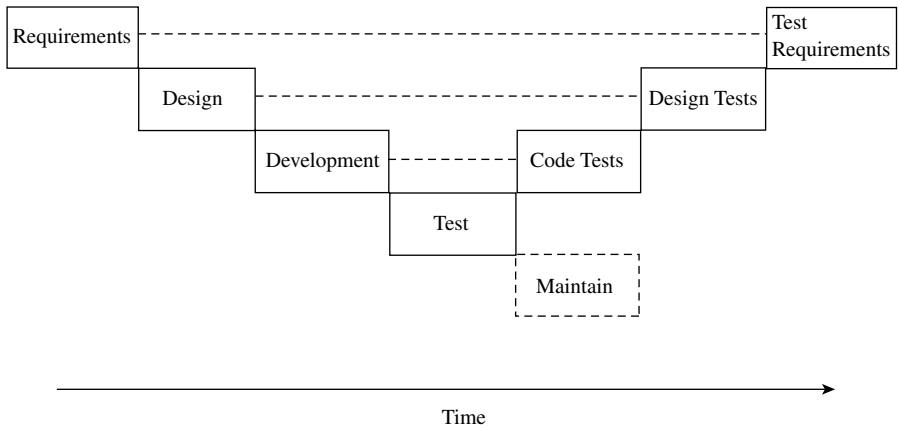


FIGURE 3.2 A V model for the software project life cycle. The concept phase is combined with the requirements phase in this instance.

model augments the waterfall model with a series of strategic prototyping and risk assessment activities throughout the life cycle (Boehm, 1988). The spiral model is depicted in Figure 3.3.

Starting at the center of the figure, the product life cycle continues in a spiral path from the concept and requirements phases. Prototyping and risk analysis are

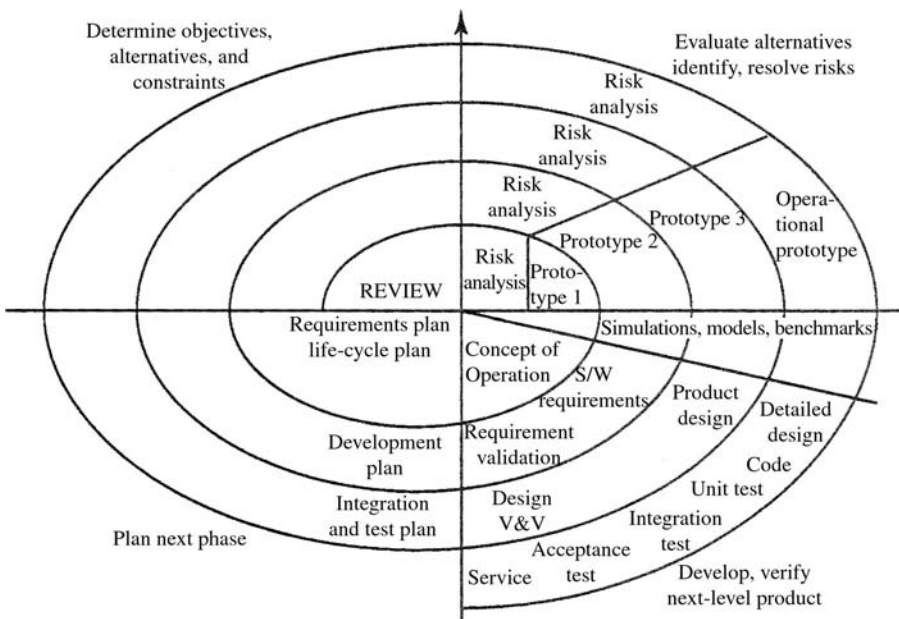


FIGURE 3.3 The spiral software model. (Redrawn from Boehm, B., *IEEE Comput.*, 26, 61-72, 1988.)

used along the way to evaluate the feasibility of potential features. More prototyping is used after a software development plan (SDP) is written, and again after the design and tests have been developed. After that, the model behaves essentially like the waterfall model. The spiral model is apparently not yet widely used; the aforementioned survey indicated that about 9% of organizations used it (Laplante et al., 2002e).

In any event, the added risk protection benefit from the extensive prototyping can be costly, but is well worth it, particularly in embedded systems. More will be mentioned about risk later.

3.2.4 EVOLUTIONARY MODEL

The evolutionary life cycle model promotes software development by iteratively defining requirements for new system increments based on experience from previous iterations. The evolutionary model is also known as evolutionary prototyping, rapid delivery, evolutionary delivery cycle, and rapid application delivery (RAD). This model is gaining in popularity; for example, in the previously mentioned survey, almost 20% of respondents indicated its adoption (Laplante et al., 2002e).

In the evolutionary model, each system iteration follows the waterfall model in that there are requirements, software design, and testing phases. After the final evolutionary step, the system enters the maintenance phase, although it can evolve again through the conventional flow, if necessary.

From the developer's point of view, those requirements that are clear at the beginning of the project drive the initial increment, but the requirements become clearer with each increment.

The evolutionary model can be used in conjunction with imaging systems, particularly in working with prototype or novel hardware and where sensor or camera inputs must first come from simulators and not real images during development. Indeed, there may be significant benefits to this approach. First, early delivery of portions of the system can be generated, even though some of the requirements are not finalized. Then these early releases are used as tools for requirements elicitation, including timing requirements.

However, there are some dangers in using this approach. First, there may be difficulties in estimating costs and schedule when the scope and requirements are ill defined. In addition, with this methodology, the overall project completion time may be greater than if the scope and requirements are established completely before design. Unfortunately, time apparently gained on the front end of a project because of early releases may be lost later because of the need for rework resulting from evolving requirements. Indeed, care must be taken to ensure that the evolving system architecture is both efficient and maintainable so that the completed system does not resemble a patchwork of afterthought add-ons. Finally, additional time must also be planned for integration and regression testing as increments are developed and added to the system. Some of the difficulties in using this approach in imaging systems can be mitigated, however, if the high-level requirements and overall architecture are established before entering an evolutionary cycle.

3.2.5 INCREMENTAL MODEL

The incremental model is characterized by a series of detailed system increments, each increment incorporating new or improved functionality to the system. These increments may be built serially or in parallel, depending on the nature of the dependencies among releases and on the availability of resources.

The difference between the incremental and evolutionary models is, of course, that the incremental model allows for parallel increments. In addition, the serial releases of the incremental model are planned, whereas in the evolutionary model, each sequential release is a function of the experience from the previous iteration.

There are several advantages to the incremental model. These include the ease of understanding each increment because of the decreased functionality, use of successive increments in requirements elicitation, early development of initial functionality (which may aid in developing the real-time scheduling structure and in debugging prototype hardware), and successive building of operational functionality over time. The thinking is that software released in increments over time is more likely to satisfy changing user requirements than if the system were planned as a single overall release at the end of the same period. Finally, because the subprojects are smaller, project management is more manageable for each increment.

However, as with the evolutionary model, there may be increased system development costs and difficulties in developing temporal behavior and meeting timing constraints with a partially implemented system.

There is some evidence that the incremental model is fairly widespread; the aforementioned survey suggests that slightly more than 20% of companies are using it (Laplante et al., 2002e).

3.2.6 FOUNTAIN MODEL

In object-oriented approaches to software development, the waterfall model may be too restrictive. In some cases, an iterative, object-oriented model is used. The fountain model, although iterative, differs from the incremental and evolutionary models in that it emphasizes two key elements of the object-oriented approaches: reuse and domain analysis and design.

In the fountain model, it is tacitly accepted that while some life cycle activities cannot start before others (for example, software development before design), there must be significant overlap and merging of activities during other phases.

The analogy is that just as in a fountain, where water rises up the middle and either falls back to the pool below or is trapped at the intermediate level, the general flow from analysis through design to implementation in object-oriented software development is interlaced with iterative cycles across two (or all three) of the requirements, design, and development phases (see Figure 3.4). The pool represents a pool of classes.

Development of an object-oriented system is much more likely to lead to focus on sections of the whole, known as clusters or subsystems. These subsystems are collections of classes with high cohesion. Some of these classes can be identified fairly early in the systems-level analysis and design, while others emerge later.

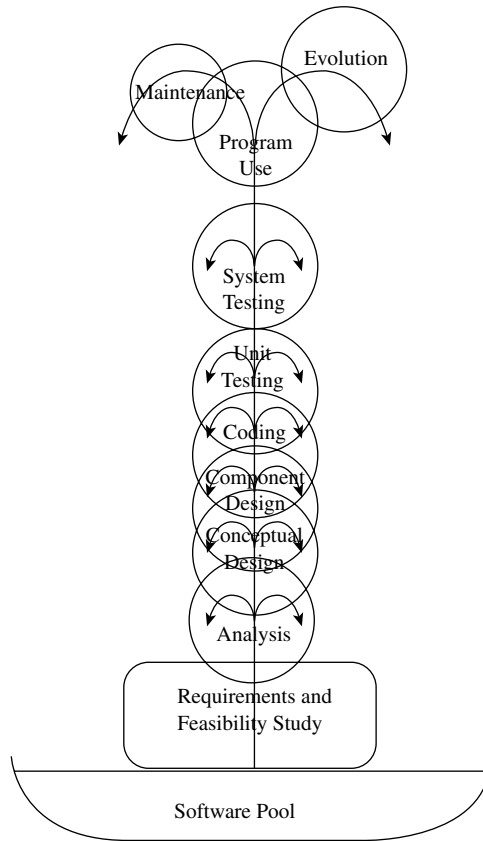


FIGURE 3.4 The fountain software model. (From Henderson-Sellers, B. and Edwards, J.M., *Comm. of the ACM*, 33, 9, 142–159, 1990. With permission.)

The stage of development of each of the classes in one subsystem proceeds at the same rate as the other stages; yet the life cycle stage of different subsystems within the software system being developed or modified could be very different.

More will be said on the suitability of object-oriented approaches to imaging systems.

3.2.7 LIGHTWEIGHT METHODOLOGIES

Recently, a variety of “lightweight” programming methodologies have emerged that emphasize rapid iteration and interaction with users. These lightweight methodologies dispense with much of the software development process documentation found in the other life cycle models, such as large SRSs. In some application domains, they have proven to be somewhat effective, particularly for smaller projects.

Lightweight methods are adaptive rather than predictive. This approach differs significantly from those previously discussed models, which emphasize planning the software in great detail over a long period and for which significant changes in the

SRS can be problematic. These lightweight methods are a response to the common problem of constantly changing requirements that can bog down the more “ceremonial” up-front design approaches, which focus heavily on documentation at the start. They strive to be processes that adapt and thrive on change, even to the point of changing themselves.

Lightweight methods are also people oriented rather than process oriented. This means they explicitly make a point of trying to make development “fun.” Presumably, this is because writing SRSs and SDDs is onerous, and the tedium should be minimized.

There are several lightweight methodologies, which are briefly summarized:

Adaptive programming — Offers a series of frameworks to apply adaptive principles and encourage collaboration.

Agile programming — Divided into four activities: planning, designing, coding, and testing, all performed iteratively.

Crystal — Empowers the development team to define the development process and refine it in subsequent iterations until it is stable.

Dynamic systems development method (DSDM) — Conceived as a methodology for rapid application development; relies on a set of principles that include empowered teams, frequent deliverables, incremental development, and integrated testing.

Extreme programming (XP) — Based on 12 practices, including pair programming (all code developed jointly by two developers), test-first coding (in which the test cases precede the coding), having the customer on-site, and frequent refactoring. XP is perhaps the most prescriptive of the lightweight methodologies.

Feature-driven development — A model-driven, short-iteration methodology built around the feature; a unit of work that has meaning for the client and developer and is small enough to be completed quickly.

Scrum — Based on the empirical process control model, the name is a reference to the point in a rugby match where the opposing teams line up in a tight and contentious formation. Scrum programming relies on self-directed teams and dispenses with much advanced planning, task definition, and management reporting.

It is easy to see that these methodologies are similar to evolutionary or incremental software development, but with much less supporting documentation.

While there are organizations developing significant systems using one or more of these techniques, there is not enough research yet to indicate that lightweight methodologies are the best way to develop robust and maintainable software. It must be emphasized that experience shows that poor documentation can lead to maintenance difficulties in embedded systems over a long period. Therefore, unless lightweight approaches are used as a subprocess within one of the more traditional process methodologies, they should be used in imaging systems development with care.

3.2.8 UNIFIED PROCESS MODEL

The unified process model (UPM) uses an object-oriented approach by modeling a family of related software processes using the unified modeling language (UML) as a notation. Like UML, UPM is a metamodel for defining processes and their components. Hence, any tool based on UPM would be a tool for process authoring and customizing. The actual enactment of processes — that is, planning and executing a project using a process described with UPM — is not within the scope of this model.

UPM was developed to support the definition of software development processes, specifically including those processes that involve or mandate the use of UML, such as the rational unified process, and is closely associated with the development of systems using object-oriented techniques.

3.2.9 CAPABILITY MATURITY MODEL

The capability maturity model (CMM) for software is not a life cycle model, but rather a system for describing the principles and practices underlying software process maturity. CMM is intended to help software organizations improve the maturity of their software processes in terms of an evolutionary path from *ad hoc*, chaotic processes to mature, disciplined software processes. Developed by the Software Engineering Institute at Carnegie Mellon University, the CMM is organized into five maturity levels.

Except for level 1, each maturity level is decomposed into several key process areas that indicate the areas upon which an organization should focus to improve its software process. Each key process area is described in terms of the key practices that contribute to satisfying its goals. The key practices describe the infrastructure and activities that contribute most to the effective implementation and institutionalization of the key process area.

Predictability, effectiveness, and control of an organization's software processes are believed to improve as the organization moves up these five levels. While not truly rigorous, there is some empirical evidence that supports this position.

3.2.9.1 CMM-1: Initial

In this level, the software process is characterized as *ad hoc* and chaotic. This means that few processes are defined, and success depends on individual effort and heroics.

3.2.9.2 CMM-2: Repeatable

Here, basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

The key process areas focus on the software project's concerns related to establishing basic project management controls. They are requirements management, software project planning, software project tracking and oversight, software subcontract management, software quality assurance, and software configuration management.

3.2.9.3 CMM-3: Defined

At this level, the software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, customized version of the organization's standard software process for developing and maintaining software.

The key process areas at this level address both project and organizational issues, as the organization establishes an infrastructure that institutionalizes effective software engineering and management processes across all projects. They are organization process focus, organization process definition, training program, integrated software management, software product engineering, intergroup coordination, and peer reviews.

3.2.9.4 CMM-4: Managed

Here, detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.

The key process areas focus on establishing a quantitative understanding of both the software process and the software work products being built. They are quantitative process management and software quality management.

3.2.9.5 CMM-5: Optimizing

In the last level, continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

The key process areas at this level cover the issues that both the organization and the projects must address to implement continual, measurable software process improvement. They are defect prevention, technology change management, and process change management.

3.2.9.6 CMM-I

The capability maturity model integration (CMM-I) is intended to provide guidance for improving an organization's processes and its ability to manage the development, acquisition, and maintenance of products and services. The CMM-I product suite is composed of multiple integrated models, courses, and an assessment method.

CMM-I is a more generic version of the CMM that is suitable for other endeavors besides software. CMM-I has three varieties: one for software and systems engineering, one that includes integrated product and process development, and a variety that includes some acquisition aspects.

3.2.10 PROTOTYPING AND RISK

Risk factors that threaten imaging systems, in fact, any software system, come from internal and external sources. Internal risks are related to software product, application domain, personnel characteristics, resources available, and schedule constraints. External threats are related to the business environment.

The major internal risk factors that concern embedded system designers include:

- Ongoing requirements changes
- Unrealistic requirements
- Incorrect requirements
- Shortfalls in externally furnished components
- Legacy code problems (e.g., lack of documentation)
- Lack of appropriate tools
- Real-time performance shortfalls
- Hardware and software inadequacies

One way of identifying at least some of these risks early, and mitigating them, is through prototyping. The spiral model makes explicit use of prototyping to mitigate risk, while the evolutionary and incremental models are, in a sense, based on a series of functioning prototypes. The lightweight methodologies could be interpreted as nothing more than intense prototyping.

In imaging systems, use of a mock-up or prototype achieves two goals: it gives the users a feel for how well the design approach works, and it provides an opportunity to exercise prototype hardware that may accompany the embedded system. This approach can detect problems and identify deficiencies early in the life cycle, where changes are more easily and cheaply made. Prototyping also increases communication between those who write requirements and the developers throughout the design process.

There are at least two drawbacks to using prototypes. First, a prototype may not provide good information about timing characteristics and real-time performance, and can lull the designers into a false sense of security. Clearly, the timing results from prototypes cannot always be trusted, though this is not to say that prototypes cannot be used to elicit some form of preliminary timing information.

Second, the pressures of bringing a product to market may lead to a temptation to carry over portions of the prototype into the final system. To avoid this problem, when using prototyping and prototyping environments in particular, be sure that the intent of the prototype is not confused with a first attempt at the real system. In fact, using “throwaway” prototypes is strongly recommended, as opposed to evolutionary prototypes.

3.3 SOFTWARE STANDARDS

Standardizing organizations such as the International Standards Organization (ISO), Association for Computing Machinery (ACM), IEEE, the U.S. Department of Defense (DOD), and others actively promote the development, use, and improvement of standards for software processes and inherent life cycle models. Even though many are interrelated and mutually influenced, the array of standards available can be confusing and even contradictory to the point of frustration.

The intent here is not to endorse any of these standards or to provide a comprehensive survey. Rather, a review of the most widely accepted ones is of value because any, if used correctly, can improve the reliability and compatibility of imaging

systems and provide benchmarks for certain software life cycle practices. Much of the following sections are adapted from the excellent text on software standards by Wang and King (2000).

3.3.1 DOD-STD-2167A

DOD-STD-2167A was designed to produce documentation that achieves a high-integrity description of the evolving software design for baseline control and that serves as the foundation for life cycle management. Formal reviews are prescribed throughout, but are sometimes just staged presentations. These audits often prove to be of questionable value and ultimately increase the cost of the system.

However, DOD-STD-2167A provides structure and discipline into the chaotic and complex development environment of large and mission-critical embedded applications. In fact, because the U.S. DOD is the single largest procurer of software, including imaging systems, the 2167A and waterfall “culture” pervade suppliers of military systems software today, even though the standard has been superseded.

3.3.2 DOD-STD-498

DOD-STD-498 (or MIL-STD-498) is a merger of DOD-STD-2167A (or MIL-STD-2167A), used for weapon systems, with DOD-STD-7935A, used for automated information systems. Together, they form a single software development standard for all of the organizations in the purview of the U.S. DOD. The purpose of developing this standard, which was approved in 1994, was to resolve issues raised in the use of the old standards, particularly with their incompatibility with modern software engineering practice.

The process model adopted in MIL-STD-498 is significantly different from that in 2167A. The former standard explicitly imposed a waterfall model, whereas 498 provides for a development model that is compatible with all of the software life cycle models that have been discussed, except the lightweight methodologies, because of their limited documentation.

With MIL-STD-498, the emphasis has changed from the use of SDDs and the configuration baseline to identification and control of the software product itself. Flexible arrangement of this standard was designed to be compatible with various life cycle models and to be more adaptable, interactive, and amenable to modern software architectures than 2167A's highly structured waterfall approach to development.

MIL-STD-498 does not provide a default process, such as the waterfall, for software development. It assumes that the developer has one selected. The standard provides flexibility for developers to use the process that works for them, so long as it is documented in a software development plan.

The standard activities also encourage the use of CASE tools, which can include automated version and change control, and build support. Formal reviews have been replaced by more frequent and informal reviews comprised of one-on-one low-keyed, joint technical information exchanges.

Standard 498 can be related to the CMM to measure process improvement. MIL-STD-498 made two principal advancements constant with the developer's potential for achieving process improvement. It elevated the level of process control and

removed stipulations on how developers should organize their configuration management functions.

Like its predecessor, MIL-STD-2167A, MIL-STD-498 is highly compatible with embedded systems and is the most widely used standard in this regard.

3.3.3 ISO 9000-3

ISO Standard 9000 is a generic, worldwide standard for quality improvement. The standard, which collectively is described in five standards, ISO 9000 through ISO 9004, was designed to be applied in a wide variety of manufacturing environments. ISO 9001 through ISO 9004 apply to enterprise according to the scope of their activities. ISO 9004 and ISO 9000-X are documents that provide guidelines for specific applications domains. For software development, ISO 9000-3 is the document of interest.

While ISO 9000-3 is widely adopted in Europe, an increasing number of U.S. and Asian companies have also adopted it. Even some defense software engineering units are switching to ISO 9000-3 from MIL-STD-498.

ISO released the 9000-3 quality guidelines in 1997 to help organizations apply the ISO 9001 (1994) requirements to computer software. ISO 9000-3 is essentially an expanded version of ISO 9001 with added narrative to encompass software.

The ISO standards are process-oriented, commonsense practices that help companies create a quality environment. The principal areas of quality focus are:

1. Management responsibility
2. Quality system requirements
3. Contract review requirements
4. Product design requirements
5. Document and data control
6. Purchasing requirements
7. Customer supplied products
8. Product identification and traceability
9. Process control requirements
10. Inspection and testing
11. Control of inspection, measuring, and test equipment
12. Inspection and test status
13. Control of nonconforming products
14. Corrective and preventive actions
15. Handling, storage, and delivery
16. Control of quality records
17. Internal quality audit requirements
18. Training requirements
19. Servicing requirements
20. Statistical techniques

Focusing on many of these areas, such as inspection and testing, design control, and product traceability (through a rational design process), increases the quality of a software product.

**4.4.1
General
Software
development**

The purpose of this standard is to promote procedures to control the product design and development process. These procedures must ensure that all requirements are being met. This standard encourages you to control your software development project and make sure that it is executed in a disciplined manner by incorporating one or more life cycle models and well-documented procedures that insure that:

- A. Software products meet all requirements.
- B. Software development follows your:
 - Quality plan
 - Development plan

FIGURE 3.5 Summary of Section 4.4.1 of ISO 9000-3, Software Development and Design.

Unfortunately, the standard is very general and provides little specific process guidance. For example, ISO 9000-3, 4.4 Software Development and Design, is shown in Figure 3.5. While these recommendations are helpful as a checklist, they provide very little in terms of a process that can be used.

While a number of metrics have been available to add some rigor to this somewhat generic standard, in order to achieve certification under the ISO standard, significant paper trails and overhead are required.

3.3.4 ISO/IEC

ISO/IEC 12207 describes five primary processes: acquisition, supply, development, maintenance, and operation. It divides the five processes into activities, and the activities into tasks, while placing requirements upon their execution. It also specifies eight supporting processes (documentation, configuration management, quality assurance, verification, validation, joint review, audit, and problem resolution) as well as four organizational processes (management, infrastructure, improvement, and training).

The ISO/IEC standard intends for organizations to tailor these 17 processes to fit the scope of their particular projects by deleting all inapplicable activities; it defines 12207 compliance as the performance of those processes, activities, and tasks selected by tailoring.

ISO/IEC 12207 provides a structure of processes using mutually accepted terminology, rather than dictating a particular life cycle model or software development method. Since it is a relatively high-level document, 12207 does not specify the details of how to perform the activities and tasks comprising the processes. It also does not prescribe the name, format, or content of documentation. Therefore, organizations seeking to apply 12207 need to use additional standards or procedures that specify those details.

3.4 EXERCISES

- 3.1 What is the relationship between the waterfall model and the others discussed? Can it be argued that all other life cycle models are simple variants of the waterfall model, or are they more than that?
- 3.2 Describe the possible phases in an 8-, 9-, and 10-phase waterfall model.
- 3.3 In what way can the waterfall model be analogized to the construction of a bridge?
- 3.4 Is any formal model ever applied to a real-world engineering project, such as a bridge, skyscraper, or electric circuit, faithfully and without disruption? Why should software be any different?
- 3.5 In what way can Boehm's spiral model be shown to be analogous to the process used in bridge building?
- 3.6 Is "faking a rational design process" (postdocumenting requirements and design), as Parnas and Clements suggest, sufficient, practical, and advisable for the real world?
- 3.7 How could lightweight methodologies be made more suitable for industrial-strength, imaging systems?
- 3.8 What are the benefits of adhering to a life cycle model to the following stakeholders?
 - a. Requirements writers
 - b. Designers
 - c. Software developers
 - d. Project managers
 - e. Customers
- 3.9 Describe the software life cycle model that best describes imaging projects that you are working on or have worked on. Is it one of the standard ones or *ad hoc*?
- 3.10 Review the literature for the genesis of the XP development methodology in the context of the Chrysler payroll system that fostered its creation. Is this scenario unique? In what other ways could it have been resolved, simply through a new methodology?

4 Software Requirements

A good workman is known by his tools.

Proverb

4.1 REQUIREMENTS ENGINEERING PROCESS

Requirements engineering is a subdiscipline of software engineering that is concerned with determining the goals, functions, and constraints of software systems. Requirements engineering also involves the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families. Figure 4.1 depicts the various activities (smooth rectangles) and documentation (rectangles) of the requirements engineering process in its idealized form.

Ideally, the requirements engineering process begins with a feasibility study activity, which produces a feasibility report. It is possible that the feasibility study may lead to a decision not to continue with the development of the software product. If the feasibility study suggests that the product should be developed, then requirements analysis can begin.

Requirements analysis involves defining requirements through a variety of elicitation techniques, including prototyping through the construction of system models. The requirements that are defined in the definition process need to be captured in a form that is appropriate for the requirements specification. Requirements definition and requirements specification are activities, and their associated documents are the by-products of the activities.

The process of software requirements specification (SRS) is the set of activities designed to capture behavioral and nonbehavioral aspects of the system in the SRS document. The goal of the SRS activity, and the resultant documentation, is to provide a complete description of the system's behavior without describing the internal structure. This is easily said but difficult to achieve, particularly in those systems where temporal behavior must be described.

Precise software specifications provide the basis for analyzing the requirements, validating that they are the stakeholders' intentions, defining what the designers have to build, and verifying that they have done so correctly. Taken another way, the portion of Figure 4.1 that does not deal with feasibility defines a set of activities that involves eliciting, modeling, analyzing, communicating, agreeing on, and evolving requirements.

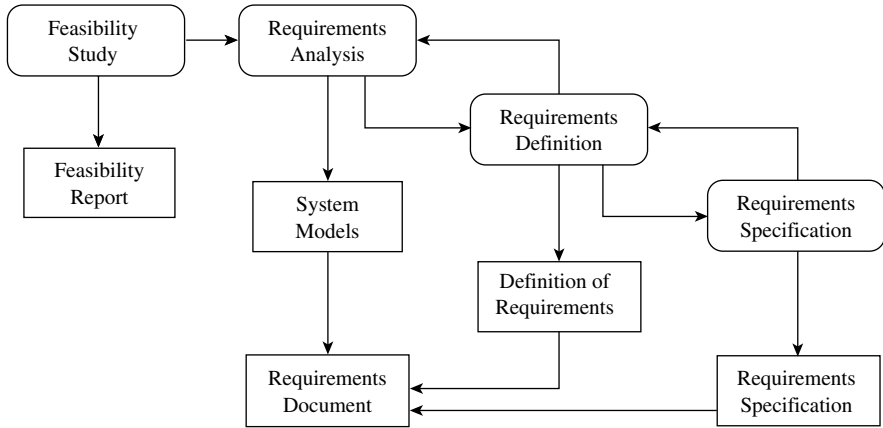


FIGURE 4.1 The requirements engineering process depicting documentation in rectangles and activities in smooth rectangles. (Adapted from Sommerville, I., *Software Engineering*, Addison-Wesley, New York, 2000.)

4.2 TYPES OF REQUIREMENTS

The Institute of Electrical and Electronics Engineers (IEEE) Standard 830 is a widely used standard for performing SRS and writing the corresponding document (IEEE, 1998). Standard 830 defines the following kinds of requirements:

1. Functional
2. External interfaces
3. Performance
4. Logical database
5. Design constraints
 - Standards compliance
 - Software systems attributes
6. Software system attributes
 - Reliability
 - Availability
 - Security
 - Maintainability
 - Portability

Requirements 2 through 6 are considered to be nonfunctional.

Functional requirements include a description of all system inputs, and the sequence of operations associated with each input set. Either through case-by-case description or through some other general form of description (e.g., using universal quantification), the exact sequence of operations and responses (outputs) to normal and abnormal situations must be provided for every input possibility. Abnormal situations might include error handling and recovery. In essence, functional requirements describe the complete deterministic behavior of the system. Generally, the

functional requirements are allocated to software and hardware before requirements analysis begins, though constant trade-off analysis may cause these to shift further into the project life cycle.

External interface requirements are a description of all inputs and outputs to the system, including:

- Name of item
- Description of purpose
- Source of input or destination of output
- Valid range, accuracy, and tolerance
- Units of measure
- Timing
- Relationships to other inputs and outputs
- Screen formats and organization
- Window formats and organization
- Data formats
- Command formats

Performance requirements include the static and dynamic numerical requirements placed on the software or on human interaction with the software as a whole. For an imaging system, static requirements might include the number of simultaneous users to be supported. The dynamic requirements might include the numbers of transactions and tasks, and the amount of data to be processed within certain time periods for both normal and peak workload conditions.

Logical database requirements include the types of information used by various functions such as frequency of use, accessing capabilities, data entities and their relationships, integrity constraints, and data retention requirements.

Design constraint requirements are related to standards compliance and hardware limitations.

Lastly, software system attribute requirements include reliability, availability, security, maintainability, and portability.

It should be noted that the conventional nomenclature for functional vs. non-functional requirements is unfortunate because the terms *functional* and *nonfunctional* seem inappropriate for embedded software systems. A more logical taxonomy would include a classification of behavior observable via execution and that not observable via execution (e.g., maintainability, portability).

4.3 REQUIREMENTS USERS

A variety of stakeholders use the software requirements throughout the software life cycle. Stakeholders include customers (these might be external customers or internal customers such as the marketing department), managers, developers, testers, and those who maintain the system. Each stakeholder has a different perspective on and use for the SRS. Various stakeholders and their uses for the SRS are summarized in Table 4.1.

TABLE 4.1
Software Stakeholders and Their Uses of the SRS

Stakeholder	Use
Customers	Specify the requirements and verify that they meet their needs; they specify or approve changes to the requirements
Managers	Use the requirements document to plan a bid for the system and to plan the system development process
Developers	Use the requirements to understand what system is to be developed
Test engineers	Use the requirements to develop validation tests for the system
Maintenance engineers	Use the requirements to help understand the system and the relationship between its parts

An important consideration in eliciting requirements from stakeholders is that they often do not know what they really want. The software engineer has to be sensitive to the needs of the stakeholders and aware of the problems that stakeholders can create, including:

- Expressing requirements in their own terms
- Providing conflicting requirements
- Introducing organizational and political factors, which may influence the system requirements
- Changing requirements during the analysis process due to new stakeholders who may emerge and changes to the business environment

The software engineer must monitor and control these factors throughout the requirements engineering process.

4.4 FORMAL METHODS IN SOFTWARE SPECIFICATION

Formal methods attempt to improve requirements formulation and expression by the use and extension of existing mathematical approaches such as propositional logic, predicate calculus, and set theory. This approach is attractive because it offers a more scientific way to requirements specification.

Writing formal requirements can often lead to error discovery in the earliest phases of the software life cycle, where they can be corrected quickly and at a low cost. Informal specifications might not achieve this goal because they are not precise enough to be refuted by counterexample.

By their nature, specifications for most imaging systems usually contain some formality in the mathematical expression of the underlying imaging operations. While this fact does not justify the claim that every imaging system specification is fully formalized, it does lead to some optimism that imaging systems can be made suitable for, at least, partial formalization.

Approaches to requirements specification that are not formal are either informal (such as flowcharting) or semiformal. The unified modeling language (UML) is a semiformal specification approach, meaning that while it does not appear to be mathematically based, in fact it is nearly formal because every one of its modeling tools can be converted either completely or partially to an underlying mathematical representation (a work group is focused now on remedying these deficiencies). In any case, UML largely enjoys the benefits of both informal and formal techniques.

Formal methods are typically not intended to take on an all-encompassing role in system or software development. Instead, individual techniques are designed to optimize one or two parts of the development life cycle.

There are three general uses for formal methods:

1. Consistency checking — Where system behavioral requirements are described using a mathematically based notation
2. Model checking — Uses state machines to verify whether a given property is satisfied under all conditions
3. Theorem proving — In which axioms of system behavior are used to derive a proof that a system will behave in a given way

Formal methods offer important opportunities for reusing requirements. Embedded systems are often developed as families of similar products or as incremental redesigns of existing products. For the first situation, formal methods can help identify a consistent set of core requirements and abstractions to reduce duplicate engineering effort. For redesigns, having formal specifications for the existing system provides a precise reference for baseline behavior and a way to analyze proposed changes.

Formal methods, however, are difficult to use by even the most expertly trained persons and are sometimes error prone. For these reasons and because they are sometimes perceived to increase early life cycle costs and delay projects, formal methods are frequently avoided.

Possibly the biggest challenge in applying formal methods to image processing systems is choosing an appropriate technique to match the problem. Still, to make formal models usable by a wide spectrum of people, requirement documents should use one or more nonmathematical notations such as natural language, structured text, or diagrams.

4.4.1 LIMITATIONS OF FORMAL METHODS

Formal methods have two limitations that are of special interest to embedded system developers. First, although formalism is often used in pursuit of absolute correctness and safety, it can guarantee neither. Second, formal techniques do not yet offer good ways to reason about alternative designs or architectures.

Correctness and safety are two of the original motivating factors driving adoption of formal methods. Nuclear, defense, and aerospace regulators in several countries now mandate or strongly suggest use of formal methods for safety-critical systems. This environment has driven an emphasis on safety-oriented applications of formal

methods in the literature. Some researchers emphasize the correctness properties of particular mathematical approaches, without clarifying that mathematical correctness in the development process might not translate into real-world correctness in the finished system. After all, it is only the specification that must be produced and proven at this point, not the software product itself.

Formal software specifications must be converted to a design and, later, to a conventional implementation language. This translation process is subject to all the potential pitfalls of any programming effort. For this reason, testing is just as important when using formal requirement methods as when using traditional ones. Formal verification is also subject to many of the same limitations as traditional testing, namely, that testing cannot prove the absence of bugs — only their presence.

Notation evolution is a slow, but ongoing process in the formal methods community. It can take many years from when a notation is created until it is adopted in industry. Therefore, an important trend influencing this evolution is the rise of semiformal, object-oriented analysis notations, such as the UML. There is some disagreement, however, on whether UML should have a significant place in formal requirements analysis.

4.4.2 Z

Z (pronounced *zed*), introduced in 1982, is a formal specification language that is based on set theory and predicate calculus. As in other algebraic approaches, the final specification in Z is reached by a refinement process starting from the most abstract aspects of the system. There is a mechanism for system decomposition known as schema calculus. Using this calculus, the system specification is decomposed into smaller pieces called schemes, where both static and dynamic aspects of system behavior are described.

The Z language does not have any support for defining timing constraints. Therefore, in recent years, several extensions for time management have been proposed. For example, Z has been integrated with real-time interval logic (RTIL), which provides for an algebraic representation of temporal behavior.

There are other extensions of Z to accommodate the object-oriented approach, which adds formalism for modularity and specification reuse. These extensions define the system state space as a composition of the state spaces of the individual system objects.

Most of these extensions also provide for information hiding, inheritance, polymorphism, and instantiation into the Z schema calculus. For example, one extension, Object-Z, includes all the aforementioned extensions and further integrates the concepts of temporal logic, making it suitable for real-time specification. In this language, the object status is a sort of event history of object behavior, making the language more operational than the early version of Z.

4.4.3 FINITE STATE MACHINES

The finite state machine (FSM), also known as the finite state automaton (FSA) or state transition diagram (STD), is a type of mathematical model used in the speci-

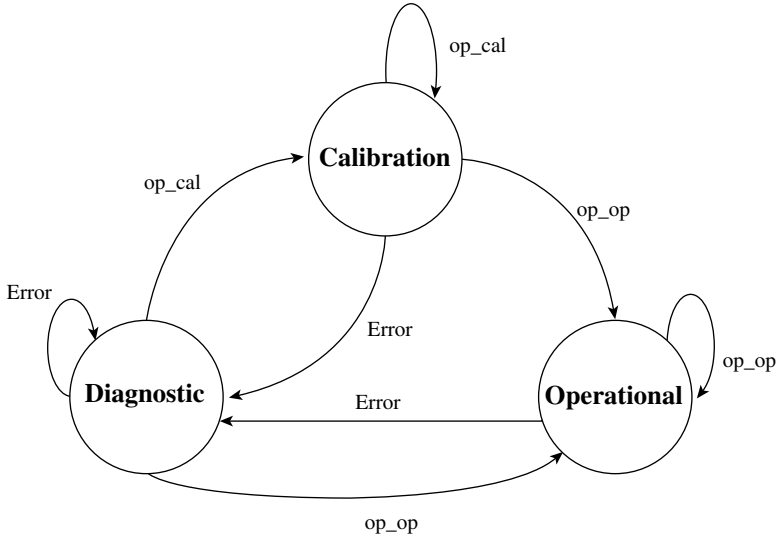


FIGURE 4.2 Partial FSM showing behavior of the visual inspection system.

fication and design of a wide range of systems. Intuitively, FSMs rely on the fact that many systems can be represented by a fixed number of unique states. The system may change state depending on time or the occurrence of specific events — a fact that is reflected in the automaton.

An FSM can be specified in diagrammatic, set-theoretic, and matrix representations. To illustrate them, consider the visual inspection system. Suppose it can be in one of three modes of operation: calibration, diagnostic, or operational. The calibration mode is entered when the operator sets a signal (op_cal). Similarly, the system returns to operational mode upon issuance of the op_op signal. The diagnostic mode is entered if an exceptional condition or error occurs in either of the other modes. The diagnostic mode can only be exited by operator intervention by setting the appropriate signal. This behavior can be described by the FSM shown in Figure 4.2.

The behavior shown in Figure 4.2 can also be represented mathematically by the 5-tuple

$$M = \{S, i, T, \Sigma, \delta\} \quad (4.1)$$

where S is a finite, nonempty set of states; i is the initial state (i is a member of S); T is the set of terminal states (T is a subset of S); Σ is an alphabet of symbols or events used to mark transitions; and δ is a transition function that describes the next state of the machine given the current state, and a symbol from the alphabet (an event). That is, $\delta: S \times \Lambda \rightarrow S$.

In the visual inspection system example, $S = \{\text{calibration, diagnostic, operational}\}$, $i = \text{calibration}$, $T = S$, and $\Sigma = \{op_op, op_cal, error\}$. The transition function can be described by a set of triples of the form (state, signal, next_state).

TABLE 4.2
Transition Matrix for the FSM of the Visual
Inspection System Shown in Figure 4.7

Current State	Event		
	op_op	op_cal	error
Calibration	Operational	Calibration	Diagnostic
Diagnostic	Operational	Calibration	Diagnostic
Operational	Operational	Calibration	Diagnostic

Note: The internal entries are values of the next state function.

It is usually more convenient to represent the transition function with a transition matrix, as shown in Table 4.2.

An FSM that does not depict outputs during transition is called a Moore machine. Outputs during transition can be depicted, however, by a variation of the Moore machine, called the Mealy machine. The Mealy machine can be described mathematically by a 6-tuple,

$$M = \{S, i, T, \Sigma, \Gamma, \delta\} \quad (4.2)$$

where the first five elements of the 6-tuple are the same as those for the Moore machine and a sixth parameter, Γ , represents the set of outputs. The transition function is slightly different from before in that it describes the next state of the machine given the current state, and a symbol from the alphabet. The transition function is then $\delta: S \times \Lambda \rightarrow S \times \Gamma$.

A general Mealy machine for a system with three states, three inputs, and three outputs such as the visual inspection system (VIS) is shown in Figure 4.3.

The transition matrix for the FSM in Figure 4.3 is shown in Table 4.3.

FSMs are easy to develop, and code can be easily generated using tables to represent the transitions between states. They are also unambiguous, as they can be represented with a formal mathematical description. In addition, concurrency can be depicted by using multiple machines.

Finally, because mathematical techniques for reducing the number of states exist, programs based on FSMs can be formally optimized. A rich theory surrounding FSMs can be exploited in the development of system specifications.

On the other hand, the major disadvantage of FSMs is that the internal aspects, or “insideness,” of modules cannot be depicted. That is, there is no way to indicate how functions can be broken down into subfunctions. In addition, intertask communication for multiple FSMs is difficult to depict. Finally, depending on the system and alphabet used, the number of states can grow very large. Both of these problems, however, can be overcome through the use of statecharts.

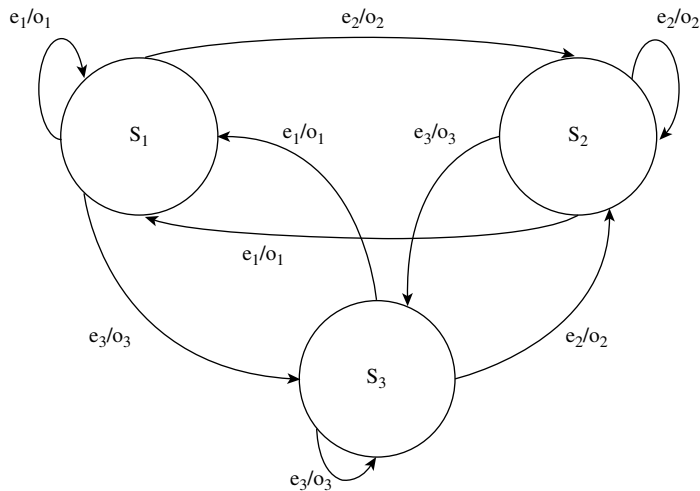


FIGURE 4.3 A generic Mealy machine for a three-state system with events e_1 , e_2 , and e_3 and outputs o_1 , o_2 , and o_3 .

TABLE 4.3
Transition Matrix for FSM in Figure 4.3

	S ₁	S ₂	S ₃
e ₁	S ₁ /S ₁	S ₁ /S ₁	S ₁ /S ₁
e ₂	S ₂ /O ₂	S ₂ /O ₂	S ₂ /O ₂
e ₃	S ₃ /O ₃	S ₃ /O ₃	S ₃ /O ₃

4.4.4 STATECHARTS

Statecharts combine FSMs with data flow diagrams (DFDs) and a feature called broadcast communication in a way that can depict synchronous and asynchronous operations. Statecharts can be described succinctly as statecharts = FSM + depth + orthogonality + broadcast communication (Figure 4.4). Depth represents levels of detail, orthogonality represents separate tasks, and broadcast communication is a method for allowing different orthogonal processes to react to the same event. The statechart resembles an FSM where each state may contain its own FSM describing its behavior. The various components of the statechart are depicted as follows:

1. The FSM is represented in the usual way, with capital letters or descriptive phrases used to label the states.
2. Depth is represented by the insideness of states.
3. Broadcast communications are represented by labeled arrows, in the same way as FSMs.
4. Orthogonality is represented by dashed lines separating states.

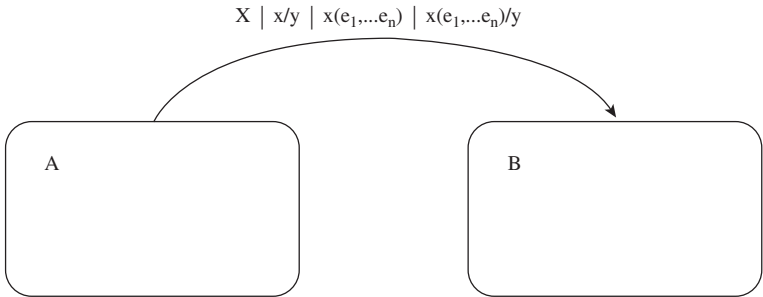


FIGURE 4.4 Statechart format where A and B are states, x is an event that causes the transition marked by the arrow, y is an optional event triggered by x, and e_1, \dots, e_n are conditions qualifying the event.

- 5. Symbols a, b, \dots, z represent events that trigger transitions, in the same way that transitions are represented in FSMs.
- 6. Small letters within parentheses represent conditions that must be true for the transitions to occur.

A significant feature of statecharts is the encouragement of top-down design of a module. For example, for any module (represented like a state in an FSM), increasing detail is depicted as states internal to it. In Figure 4.5, the system is composed of states A and B. Each of these in turn can be decomposed into states A1 and A2 and B1 and B2, respectively, which might represent program modules. Those states can also be decomposed, and so forth. To the software designer, each nested substate within a state represents a procedure within a procedure.

Orthogonality depicts concurrency in the system for processes that run in isolation, called AND states. Orthogonality is represented by dividing the orthogonal components by dashed lines. For example, if state Y consists of AND components

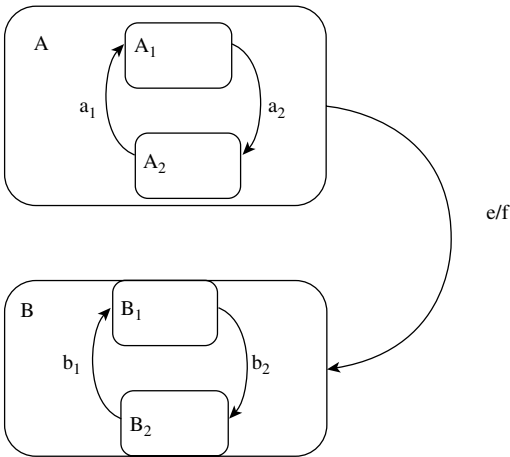


FIGURE 4.5 A statechart depicting insideness.

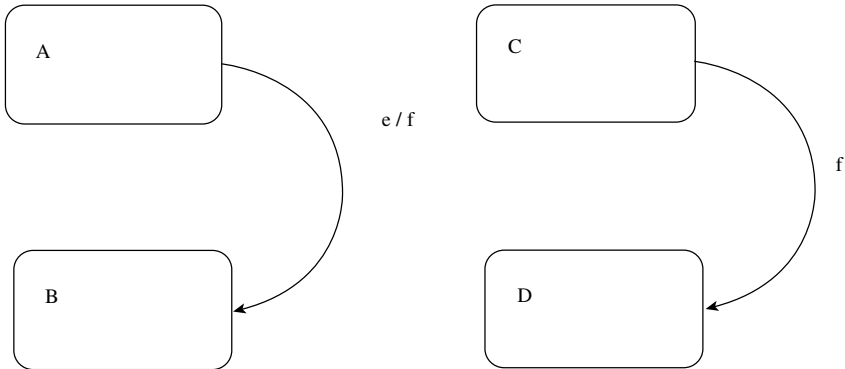


FIGURE 4.6 Statechart depicting a chain reaction.

A and D, Y is called the orthogonal product of A and D. If Y is entered from the outside (without any additional information), then states A and D are entered simultaneously. A commonwealth between the AND states can be achieved through global memory, whereas synchronization can be achieved through a unique feature of statecharts called broadcast communication.

Broadcast communication is depicted by the transition of orthogonal states based on the same event. For example, if an imaging system switches from standby to ready mode, an event indicated by an interrupt can cause a state change in several processes.

Another unique aspect of broadcast communication is the concept of the chain reaction; that is, events can trigger other events. The implementation follows from the fact that statecharts can be viewed as an extension of Mealy machines, and output events can be attached to the triggering event. In contrast with the Mealy machine, however, the output is not seen by the outside world; instead, it affects the behavior of an orthogonal component.

For example, in Figure 4.6 suppose there exists a transition labeled e/f , and if event e occurs, then event f is immediately activated. Event f could, in turn, trigger a transaction such as f/g . The length of a chain reaction is the number of transitions triggered by the first event. Chain reactions are assumed to occur instantaneously. In this system, a chain reaction of length 2 will occur when the e/f transition occurs.

Statecharts are well-suited for representing embedded systems because they can easily depict concurrency while preserving modularity. In addition, the concept of broadcast communication allows for easy intertask synchronization and communication.

In short, the statechart improves upon embedded FSMs. Finally, commercial products allow an engineer to graphically design embedded systems using statecharts, perform detailed simulation analysis, and generate Ada, C or C++ code. Furthermore, statecharts can be used in conjunction with both structured and object-oriented analysis.

4.4.5 PETRI NETS

Petri nets are another formal method used to specify the operations to be performed in a multiprocessing or multitasking environment. While they have a rigorous foun-

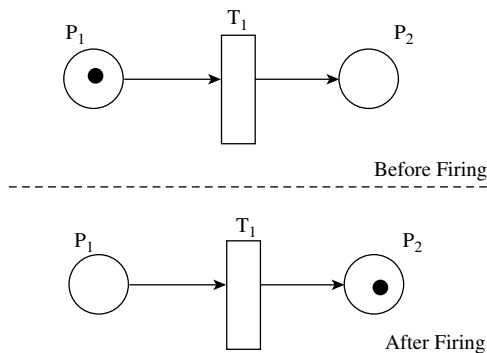


FIGURE 4.7 Petri nets firing rule.

TABLE 4.4		
Firing Table for Petri Net Shown in Figure 4.7		
	P_1	P_2
Before firing	1	0
After firing	0	1

dition, they can also be described graphically. A series of circular bubbles called places are used to represent data stores or processes. Rectangular boxes are used to represent transitions or operations. The processes and transitions are labeled with a data count and transition function, respectively, and are connected by unidirectional arcs.

The initial graph is labeled with markings given by m_0 , which represent the initial data count in the process. Net markings are the result of the firing of transitions. A transition, t , fires if has as many inputs as required for output.

In Petri nets, the graph topology does not change over time; only the markings or contents of the places do. The system advances as transitions fire.

To illustrate the notion of firing, consider the Petri nets given in Figure 4.7, with the associated firing table given in Table 4.4.

For a somewhat more significant example, consider the Petri net in Figure 4.8. Reading from left to right and top to bottom indicates the stages of firings in the net. Table 4.5 depicts the firing table for the Petri net in Figure 4.8.

Petri nets can be used to model systems and to analyze timing constraints and race conditions. Certain Petri net subnetworks can model familiar flowchart constructs. Figure 4.9 illustrates these analogies.

Petri nets are excellent for representing multiprocessing and multiprogramming systems, especially where the functions are simple. Because they are mathematical in nature, techniques for optimization and formal program proving can be employed. But Petri nets can be overkill if the system is too simple. Similarly, if the system is highly complex, timing can be become obscured.

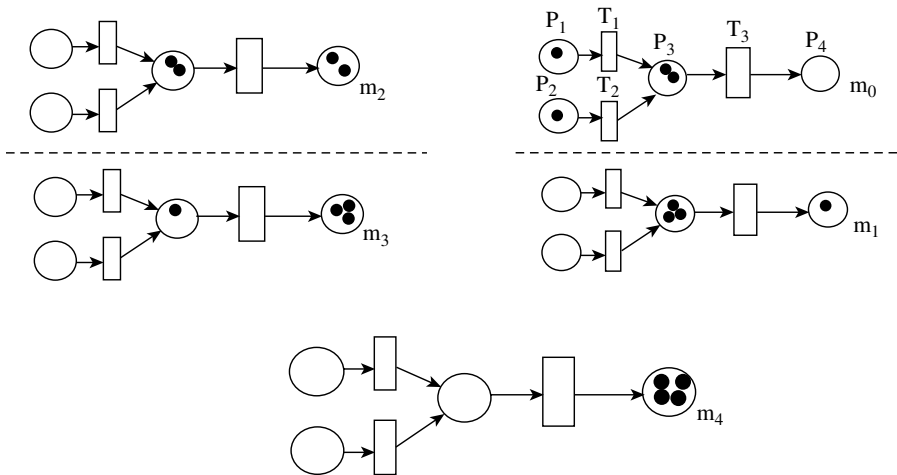


FIGURE 4.8 A slightly more complex Petri net.

TABLE 4.5
Firing Table for Petri Net in
Figure 4.8

	P ₁	P ₂	P ₃	P ₄
m ₀	1	1	2	0
m ₁	0	0	3	1
m ₂	0	0	2	2
m ₃	0	0	1	3
m ₄	0	0	0	4

The Petri net is a powerful analysis and modeling tool that can be used for deadlock and race condition identification.

The model described herein is just one of a variety of available models. For example, there are timed Petri nets, which enable synchronization of firings; colored Petri nets, which allow for labeled data to propagate through the net; and even timed-colored Petri nets, which embody both features.

4.5 SPECIFICATION OF IMAGING SYSTEMS: A SURVEY OF CURRENT PRACTICES

There appears to be no dominant approach for specification of imaging applications. Existing surveys of requirements specification do not address the use of these approaches in imaging applications. In general, it seems that imaging engineers tend to use one or a combination of the following approaches in writing software specifications for imaging applications:

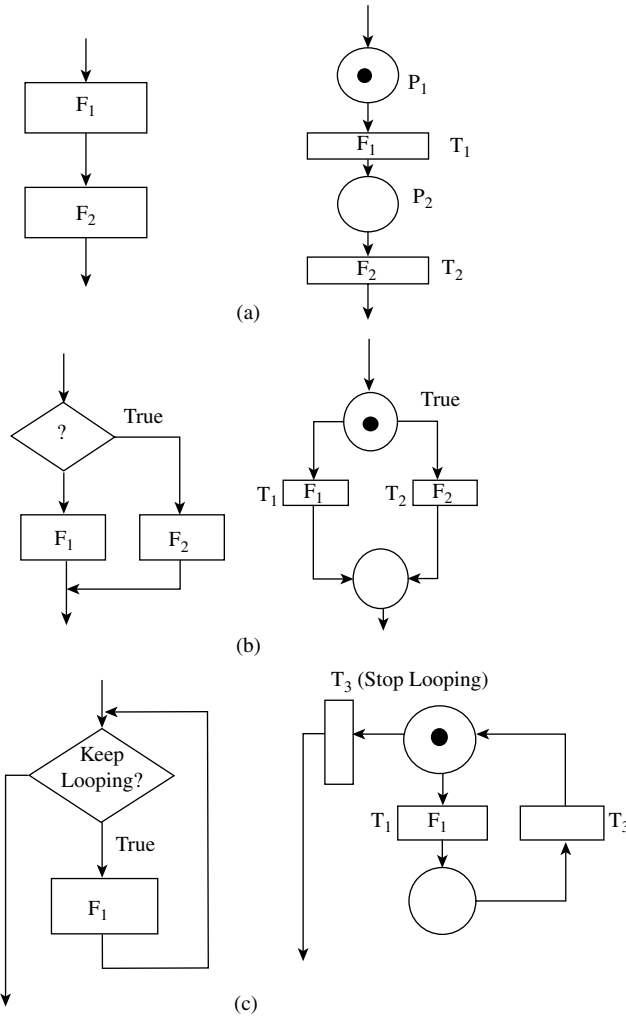


FIGURE 4.9 Flowchart equivalence: (a) sequence, (b) conditional branch, (c) while loop.

- Top-down process decomposition or structured analysis
- Object-based or object-oriented approaches
- Program description languages (PDLs) or pseudo-code
- High-level functional specifications that are not further decomposed
- *Ad hoc* techniques, including simple natural language and mathematical descriptions, which are always included in virtually every system specification

In order to illustrate this, examples excerpted from the literature are given. Much of this discussion has been adapted from Laplante and Neill (2003b).

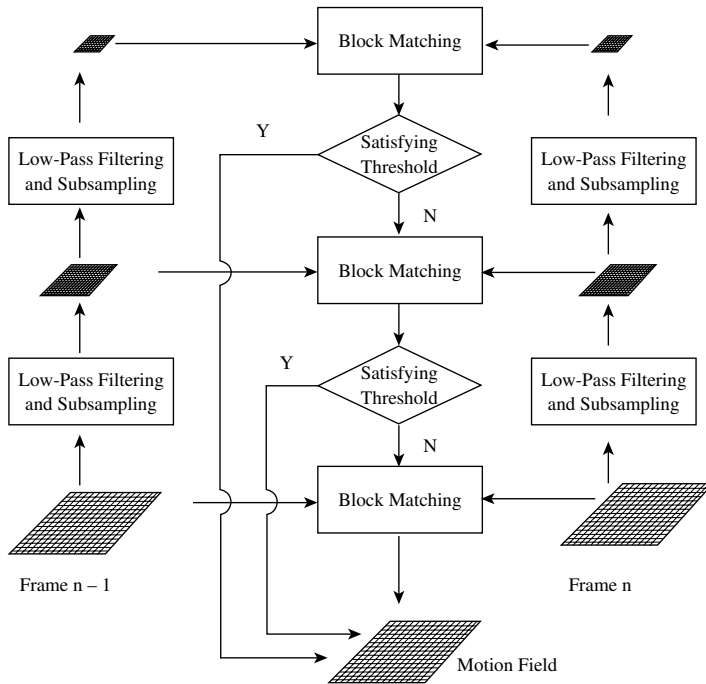


FIGURE 4.10 Block diagram for a three-level threshold multiresolution block matching. (From Shi, Y.Q. and Sun, H., *Image and Video Compression for Multimedia Engineering*, CRC Press, Boca Raton FL, 2000.)

4.5.1 MULTIREOLUTION BLOCK-MATCHING SYSTEM SPECIFICATION USING A BLOCK DIAGRAM AND FLOWCHART

The example shown in Figure 4.10 from Shi and Sun (2000) indicates the use of flowcharts in the operational specification for a multiresolution block-matching algorithm such as those used in motion estimation. In such a system, an image is partitioned into a set of nonoverlapped, equally spaced, fixed-size, small rectangular blocks. Then displacement vectors for these blocks are estimated by finding their best-matched counterparts in the previous frame. In this manner, motion estimation is significantly easier than that for arbitrarily shaped blocks.

A procedural description of the block-matching component denoted in Figure 4.10 is then depicted using a flowchart, as shown in Figure 4.11.

No further specification is given for the technique, and it is expected that an “ordinary” imaging engineer would be able to implement a design and build the systems using this level of specificity. There is no consideration for the overall software architecture, the data structures and their relationships, or the timing constraints that must be met. While this technique is suitable for the specification of individual tasks or algorithms, it is not scalable for large systems.

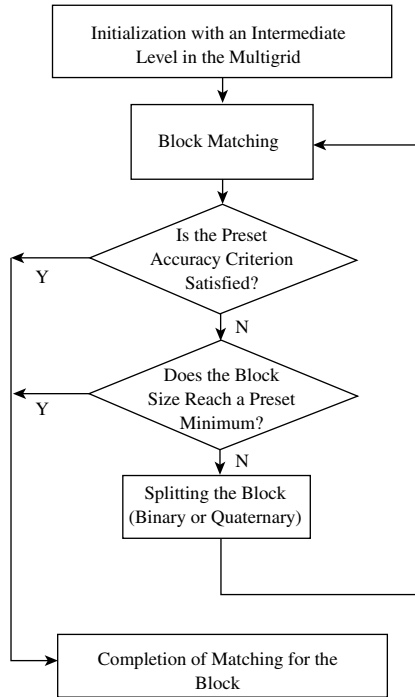


FIGURE 4.11 Flowchart of multigrid block matching. (From Shi, Y.Q. and Sun, H., *Image and Video Compression for Multimedia Engineering*, CRC Press, Boca Raton FL, 2000.)

4.5.2 COLLISION TESTING OF GRAPHICAL OBJECTS USING PSEUDO-CODE

Pseudo-code and PDLs are specification techniques with similar problems of scalability. An example of a pseudo-code specification is shown in Figure 4.12, which depicts the collision testing between graphical objects frequently used in video games, taken from Möller and Haines (1999).

This type of specification is again sufficient for individual tasks or algorithms, but only at late stages of design where the majority of decisions have been made because the pseudo-code representation is so close to code that it will likely be regarded as the final design version of system behavior, rather than a description of system expectation.

4.5.3 FUNCTIONAL REPRESENTATION OF MACHINE VISION SYSTEM USING A STRUCTURED APPROACH

Rajeswari and Rodd (1999) apply a functional description, similar to a context diagram used in structured design, to describe a machine vision system for inspecting flaws in an integrated circuit wire bond. The functional description is given in Figure 4.13. Further decomposition is given using their Q-model, which is a novel technique for the specification of temporal properties of systems similar to Petri nets.

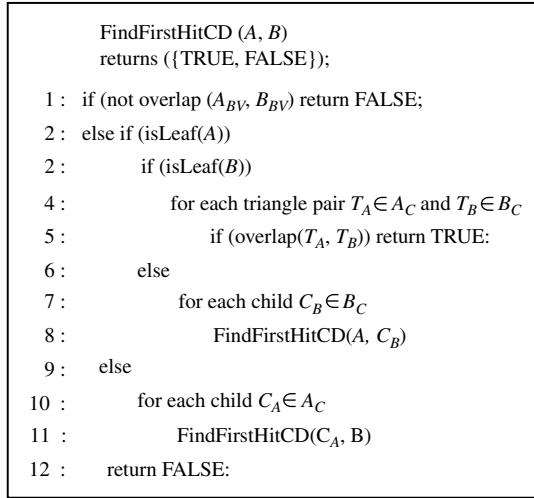


FIGURE 4.12 Hierarchical collision testing algorithm. (Redrawn from Möller, T. and Haines, E., *Real-Time Rendering*, A. K. Peters, Natick, MA, 1999.)

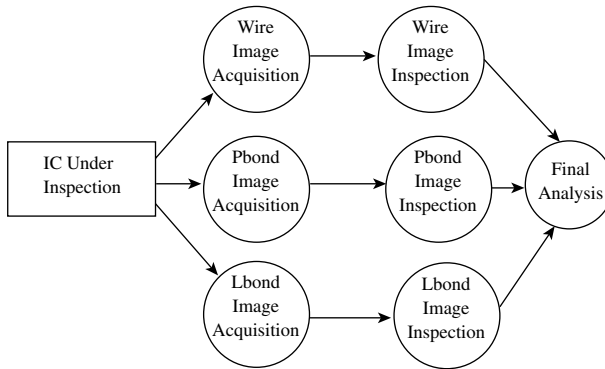


FIGURE 4.13 Functional representation of machine vision system for inspecting integrated circuit wire bonds. (From Rajeswari, M. and Rodd, M.G., *Real-Time Imaging*, 5, 409–421, 1999. With permission from Elsevier.)

This represents a significant advance over the previous examples, since there is a segregation of perspectives in the overall specification: data flow is indicated in one diagram, and timing and control information in another. The use of structured approaches has its own drawbacks, however, which will be discussed later with reference to a case study.

4.5.4 MARKOV RANDOM FIELDS IMAGE RECONSTRUCTION USING OBJECT-ORIENTED DESIGN

The final example found in the literature is the use of object-oriented techniques. It is important to note that there is a distinction between object-oriented design and

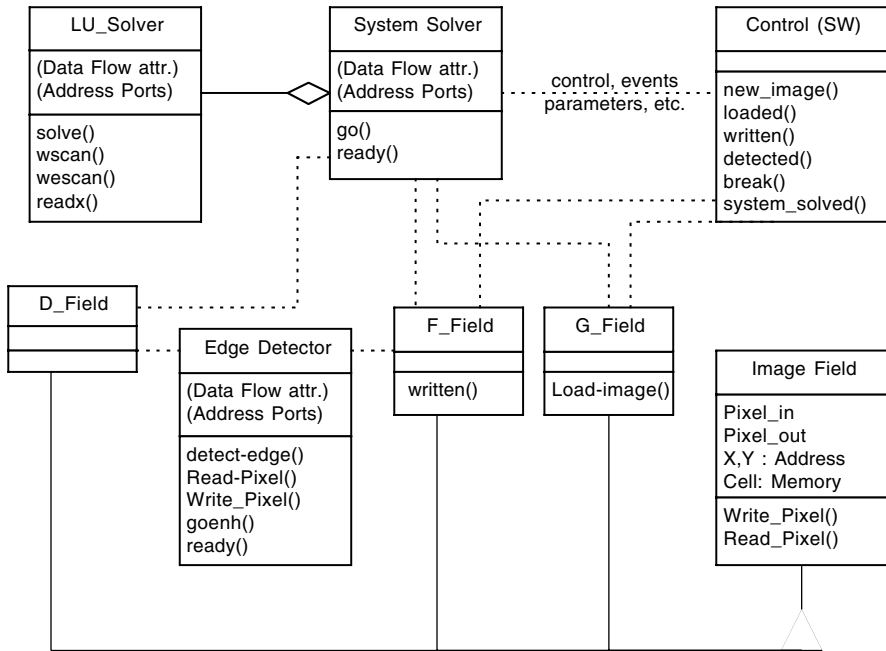


FIGURE 4.14 Class diagram of the image reconstruction system model. (From Mariatos, E.P., Birbas, M.K., Birbas, A.N., and Petrellis, N., in *Proceedings of the Seventh IEEE International Workshop on Rapid System Prototyping*, June 19–21, 1996, IEEE Computer Society Press, Los Alamitos, CA, pp. 90–95. With permission.)

specification and object-based image processing techniques. The latter is a technique used in image processing for identifying visual objects in a scene based on certain features. This is clearly different from the former, which is a software engineering approach that is interested in modeling systems as communities of entities that encapsulate state and behavior.

Mariatos et al. (1996) proposed an object-oriented design for an image reconstruction system used to reconstruct images damaged by transmission errors or lossy compression. The system applies edge-preserving smoothing transforms on an input image. The outputs are then a surface field (the transformed image) and a discontinuity field (the edge information).

It can be seen from the class model in Figure 4.14 that rather than the system being decomposed into constituent processes — a characteristic present in all the previous examples — the model is formed by representing the entities involved (F_Field is the surface field, for example).

This aspect of object-oriented analysis and design is studied in detail below.

4.6 CASE STUDY

The previous survey of requirements specification techniques indicates that there is certainly no standard way to write requirements for imaging systems. Moreover,

most of the techniques that are used are informal or even *ad hoc*. It can be concluded, then, that much of what is going on in the engineering of imaging systems lacks software engineering rigor. Because of this lack of rigor, many systems are probably costing more than they should to build and maintain.

The lack of a *de facto* standard technique for specification and design further raises the question of whether any technique is more appropriate than another for the modeling of an imaging system. While both structured and object-oriented techniques are appropriate for specification and design of imaging systems, in fact the semiformal UML is a very desirable way to develop object-oriented SRSs in this regard.

Nevertheless, either the structured or object-oriented approach can be used. To show how this might be done, the visual inspection system example is used in the specification of an imaging system using both an object-oriented technique and a non-object-oriented counterpoint, structured analysis and structured design (SASD).

4.6.1 STRUCTURED ANALYSIS AND DESIGN

Methods for SASD have evolved over almost 30 years and are widely used in image processing applications — probably because the techniques are closely associated with the programming languages with which they co-evolved (Fortran and C) and in which many image processing applications are written. Structured methods appear in many forms, but the *de facto* standard is Yourdon's modern structured analysis (Yourdon, 1991).

Yourdon's modern structured analysis uses three viewpoints to describe a system: an environmental model, a behavioral model, and an implementation model. The elements of each model are shown in Figure 4.15.

The environmental model embodies the analysis aspect of SASD and consists of a context diagram and an event list. The purpose of the environmental model is to model the system at a high level of abstraction.

The behavioral model embodies the design aspect of SASD as a series of DFDs, entity relationship diagrams (ERDs), process specifications, STDs, and a data dictionary. Using various combinations of these tools, the designer models the processes, functions, and flows of the system in detail.

Finally, in the implementation model, the developer uses a selection of structure charts, natural language, and pseudo-code to describe the system to a level that can be readily translated to code.

4.6.2 STRUCTURED ANALYSIS

Structured analysis (SA) is a way to try to overcome the problems of classical analysis using graphical tools and a top-down, functional decomposition method to define system requirements. SA deals only with aspects of analysis that can be structured — the functional specifications and the user interface.

SA is used to model a system's context (where inputs come from and where outputs go), processes (what functions the system performs, how the functions interact, how inputs are transformed to outputs), and content (the data the system needs to perform its functions).

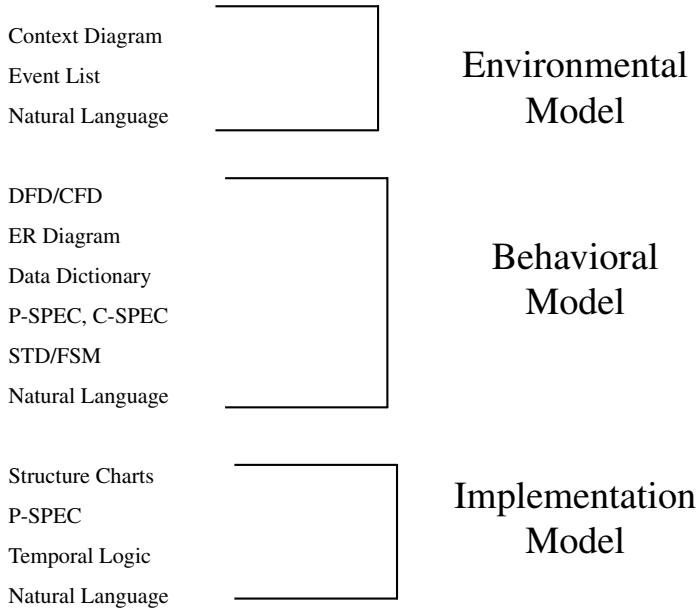


FIGURE 4.15 Elements of structured analysis and design. (From Laplante, P.A. and Neill, C.J., in *Real-Time Imaging VI*, vol. 4666, January 2002, SPIE Press, Bellingham, WA, 2002, pp. 55–64.)

SA seeks to overcome the problems inherent in analysis through:

- Maintainability of the target document
- Use of an effective method of partitioning
- Use of graphics
- Building of a logical model of the system for the user before implementation
- Reduction of ambiguity and redundancy

The target document for SA is called the structured specification. It consists of a system context diagram, an integrated set of DFDs showing the decomposition and interconnectivity of components, and an event list to represent the set of events that drive the system.

To illustrate the SA technique, consider the visual inspection system previously introduced in its operational mode (calibration and diagnostic modes are ignored for simplicity). The following discussion is largely adapted from Laplante and Neill (2003b).

Figure 4.16 depicts the context diagram. Here, the visual inspection system is shown with the other constituent system parts — camera, product detector, production conveyor controller system, and reject mechanism. Solid arcs indicate the flow of data between system components. In the example, the only data flow involves the transmission of the captured image to the visual inspection system.

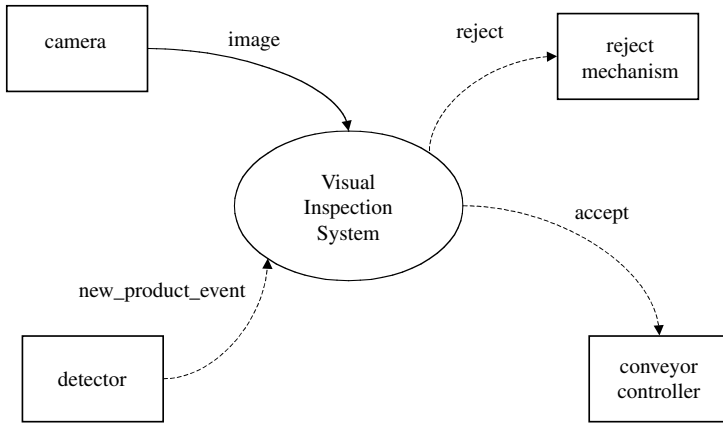


FIGURE 4.16 Context diagram for visual inspection system. (From Laplante, P.A. and Neill, C.J., in *Real-Time Imaging VI*, vol. 4666, January 2002, SPIE Press, Bellingham, WA, 2002, pp. 55–64.)

The dashed lines represent the flow of control information. This facility is one of the extensions needed for dealing with real-time systems, which is discussed in Hatley and Pirbhay (1987).

In the example, the event list consists of the “new_product_event,” which indicates the detection of the next image on the line “accept,” which indicates that the product has passed inspection and causes a signal to be sent to the conveyor controller, or “reject,” which causes a signal to be sent that directs the conveyor to move the product into a rejected product bin. The rejection mechanism automatically causes the next product item to be moved along by the conveyor controller.

It should be reiterated that this context diagram is not complete owing to the omission of the calibration and diagnostic modes. While the intent here is not to provide a complete system design, and so there are some omissions, a point to be made is that missing functionality is more easily identified during the requirements elicitation process if some form of graphical aid, such as the context diagram, is available. In the case of object-oriented analysis, a use case diagram will be helpful.

4.7 OBJECT-ORIENTED ANALYSIS

As an alternative to the SA approach to developing software requirements for the visual inspection system, consider using an object-oriented approach. There are various “flavors” of object orientation, each using their own tool sets. In the approach developed here, the system specification begins with the representation of externally accessible functionality as use cases.

Use cases are an essential artifact in object-oriented analysis (OOA) design and are described graphically by any of several techniques. The use case diagram can be considered analogous to the context diagram in SA in that it represents the interactions of the software system with its external environment.

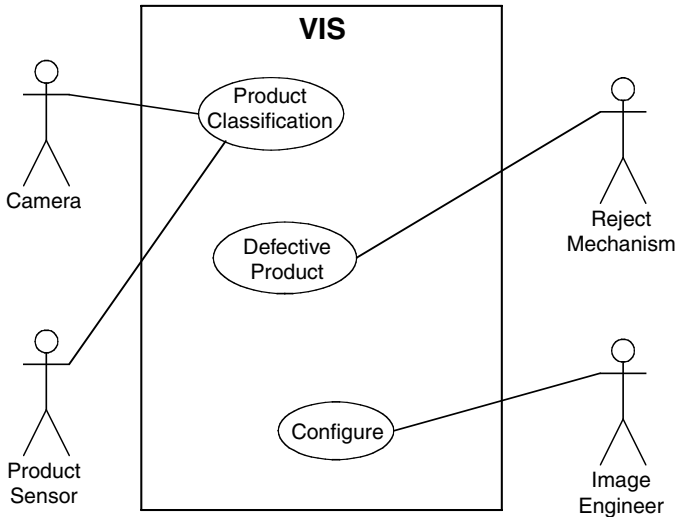


FIGURE 4.17 Use case diagram of visual inspection system. (From Laplante, P.A. and Neill, C.J., in *Real-Time Imaging VI*, vol. 4666, January 2002, SPIE Press, Bellingham, WA, 2002, pp. 55–64.)

Use cases are represented graphically as ellipses, as can be seen in Figure 4.17.

However, each use case is a document that describes scenarios of operation of the system under consideration as well as pre- and postconditions, and exceptions. In an iterative development life cycle, these use cases will become increasingly refined and detailed as the analysis and design work flows progress. Interaction diagrams are then created to describe the behaviors defined by each use case. In the first iteration, these diagrams depict the system as a black box, but once domain modeling has been completed, the black box is transformed into a collaboration of objects, as will be seen later.

4.8 OBJECT-ORIENTED VS. STRUCTURED ANALYSIS

The above observations beg the question of whether OOA is more suitable than SA for the visual inspection system in particular and for image processing applications in general. SA and OOA are often compared and contrasted, and indeed, they are similar in some ways. This should be no surprise as both have their roots in the work of Parnas (1972, 1979) and his successors. Table 4.6 provides a side-by-side comparison of the methodologies.

Both structured and OOA are full life-cycle methodologies and use some similar tools and techniques. However, there are major differences. SA describes the system from a functional perspective and separates data flows from the functions that transform them, while OOA describes the system from the perspective of encapsulated entities that possess both function and form.

Additionally, OOA models include inheritance, while SA does not. Although SA has a definite hierarchical structure, this is a hierarchy of composition rather

TABLE 4.6
A Comparison of SA and OOA

	SA	OOA
System components	Functions	Objects
Data processes	Separated through internal	Encapsulated within objects
Control processes	decomposition	
Data stores		
Characteristics	Hierarchical structure	Inheritance
	Classifications of functions	Classification of objects
	Encapsulation of knowledge within functions	Encapsulation of knowledge within objects

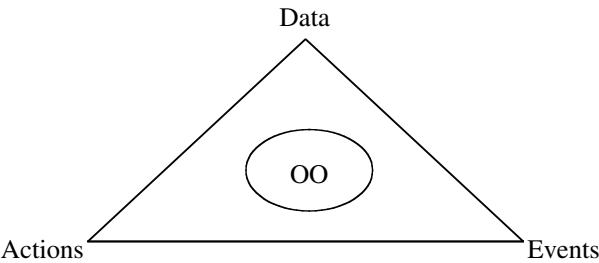


FIGURE 4.18 A project’s applicability to either object-oriented or structured analysis according to system focus. (From Laplante, P.A. and Neill, C.J., in *Real-Time Imaging VI*, vol. 4666, January 2002, SPIE Press, Bellingham, WA, 2002, pp. 55–64.)

than heredity. This shortcoming leads to difficulties in maintaining and extending both the specification and design, such as in the case of changes in the visual inspection system example.

The purpose of this discussion is not to dismiss SA, or even to conclude that it is better than OOA in all cases. An overriding indicator of suitability of OOA vs. SA to image processing is the nature of the application. To see this, consider the vertices of the triangle in Figure 4.18 representing three distinct viewpoints of a system: data, actions, and events.

Events represent stimuli and responses such as measurements in process control systems, as in the case study. Actions are rules that are followed in complex algorithms, such as binarize, threshold, and classify. The majority of early computer systems were focused on one, or at most two, of these vertices. For example, early, non-real-time image processing systems were data and action intensive but did not encounter much in the way of stimuli and response.

Image processing is data intensive and would seem well suited to SA. But often the image itself contains control information (e.g., reject, accept), which is not well suited to SA. Moreover, while it is true that image processing is data intensive and an image has high information content, this is not the same as the data intensity found in, say, a database management system. It is likely that image processing is

as much event or activity based as it is data based, which makes it quite suitable for object-oriented techniques.

The evidence and case study point out that requirements specification can be done in image processing systems using both structured and object-oriented approaches.

4.8.1 RECOMMENDATIONS ON SPECIFICATION APPROACH FOR IMAGING SYSTEMS

The preceding discussions illustrate some of the challenges (in fact, one might consider them habits) encountered by engineers specifying imaging systems:

- Mixing of operational and descriptive specifications
- Combining low-level hardware functionality and high-level systems and software functionality in the same functional level
- Omission of timing information

It is risky to prescribe a preferred technique because it is well known that there is no silver bullet when it comes to software specification and design, and each system should be considered on its own merits. Nevertheless, regardless of approach, imaging system modeling should incorporate the following best practices:

- Use consistent modeling approaches and techniques throughout the specification, for example, a top-down decomposition, structured or object-oriented approach.
- Separate operational specification from descriptive behavior.
- Use consistent levels of abstraction within models and conformance between levels of refinement across models.
- Model nonfunctional requirements as a part of the specification models — in particular timing properties.
- Omit hardware and software assignment in the specification (another aspect of design rather than specification).

4.9 ORGANIZING THE REQUIREMENTS DOCUMENT

There are many ways to organize the SRS, but IEEE Standard 830-1998 provides a template of what an SRS should look like and is IEEE's recommended practice for SRSs.

The SRS is described as a "binding contract among designers, programmers, customers, and testers," and it encompasses different design views or paradigms for system design. The recommended design views include some combination of decomposition, dependency, interface, and detail descriptions. Together with boilerplate front matter, these form a standard template for SRSs, depicted in Figure 4.19.

Sections 1 and 2 are self-evident; they provide front matter and introductory material for the SRS. The remainder of the SRS is devoted to the four description sections.

1. Introduction
1.1 Purpose
1.2 Scope
1.3 Definitions and Acronyms
1.4 References
1.5 Overview
2. Overall Description
2.1 Product Perspective
2.2 Product Functions
2.3 User Characteristics
2.4 Constraints
2.5 Assumptions and Dependencies
3. Specific Requirements
Appendices
Index

FIGURE 4.19 Table of contents outline for an SRS in IEEE Standard 830-1998 format.

3. Functional Requirements
3.1 Calibration Mode
3.2 Operational Mode
3.2.1 Initialization
3.2.2 Normal Operation
3.2.2.1 Image Capture
3.2.2.2 Image Error Correction
3.2.2.2.1 Position Error Reduction
3.2.2.2.2 Noise Error Reduction
3.2.2.3 Captured Image Analysis
3.2.2.4 Conveyor System Control
3.2.2.5 Reject Mechanism Control
3.2.2.6 Error Handling
3.3 Diagnostic Mode
4. Non-Functional Requirements

FIGURE 4.20 Some specific requirements for the visual inspection system in IEEE 830 format.

An outline of some specific requirements, or work breakdown structure (WBS), for the visual inspection system in IEEE Standard 830 format is given in Figure 4.20. The section headings can be decomposed further using a technique such as SA.

IEEE Standard 830 provides for several alternative means to represent the requirements specifications, aside from a function perspective. In particular, the software requirements can be organized by:

- Functional mode (e.g., operational, diagnostic, calibration)
- User class (e.g., operator, diagnostic)
- Object
- Feature (what the system provides to the user)

- Stimulus (e.g., sensors 1, 2, and so forth)
- Functional hierarchy (as shown in Figure 4.20)
- Mixed (combining two or more of the above)

4.9.1 WRITING GOOD REQUIREMENTS

Whatever approach is used in organizing the SRS, IEEE Standard 830 describes the characteristics of good requirements. That is, good requirements must be:

- **Correct** — They must correctly describe the system behavior.
- **Unambiguous** — The requirements must be clear, not subject to different interpretations.
- **Complete** — There must be no missing requirements. Ordinarily, the note “TBD” (to be defined later) is unacceptable in a requirements document. IEEE 830 sets out some exceptions to this rule.
- **Consistent** — One requirement must not contradict another.
- **Ranked for importance and stability** — Not every requirement is as critical as another. By ranking the requirements, designers will find guidance in making trade-off decisions.
- **Verifiable** — A requirement that cannot be verified is a requirement that cannot be shown to have been met.
- **Modifiable** — The requirements need to be written in such a way as to be easy to change. In many ways this approach is similar to the information hiding principle.
- **Traceable** — The benefits of traceability have been mentioned at length. The requirements provide the starting point for the traceability chain. Numbering the requirements in hierarchical fashion can aid in this regard.

To meet these criteria and to write clear requirements documentation, there are several best practices that the requirements engineer can follow:

- Invent and use a standard format and use it for all requirements.
- Use language in a consistent way.
- Use *shall* for mandatory requirements.
- Use *should* for desirable requirements.
- Use text highlighting to identify key parts of the requirement.
- Avoid the use of technical language unless it is warranted.

To illustrate, consider the following bad requirements:

1. “The systems shall be completely reliable.”
2. “The system shall be modular.”
3. “The system shall be maintainable.”
4. “The system will be fast.”
5. “Errors shall be less than 99%.”

These requirements are bad for a number of reasons. None are verifiable, for example, how is reliability supposed to be measured. Even number 5 is vague. What does less than 99% mean?

Now consider the following requirements:

1. "Response times for all level 1 actions will be less than 100 msec."
2. "The cyclomatic complexity of each module shall be in the range of 10 to 40."
3. "Ninety-five percent of the transactions shall be processed in less than 1 sec."
4. "An operator shall not have to wait for the transaction to complete."
5. "MTBF shall be 100 h of continuous operation."

These requirements are better versions of the preceding ones. Each is measurable because each makes some attempt to quantify the qualities that are desired. For example, cyclomatic complexity is a measure of structure, MTBF is a measure of the mean time between failures, and processing time is a measure of speed. Although improved, these requirements could stand some refinement based on the context of requirements specification as a whole.

4.10 REQUIREMENTS VALIDATION AND REVIEW

Verification of the software product means ensuring that the software is conforming to the SRS. Verification is akin to asking the question "Am I building the software right?" in that it requires satisfaction of requirements.

Requirements validation, on the other hand, is tantamount to asking the question "Am I building the right software?" Too often, software engineers deliver a system that conforms to the SRS only to discover that it is not what the customer really wanted.

Performing requirements validation involves checking the following:

- Validity — Does the system provide the functions that best support the customer's needs?
- Consistency — Are there any requirements conflicts?
- Completeness — Are all functions required by the customer included?
- Realism — Can the requirements be implemented given the available budget and technology?
- Verifiability — Can the requirements be checked?

There are a number of ways of checking an SRS for conformance to the IEEE Standard 830 best practices and for validity. These approaches include:

1. Requirements reviews
2. Systematic manual analysis of the requirements
3. Prototyping

4. Using an executable model of the system to check requirements
5. Test case generation
6. Developing tests for requirements to check testability
7. Automated consistency analysis
8. Checking the consistency of a structured requirements description

Of these, automated checking is the most desirable and the least likely because of the context sensitivity of natural languages and the impossibility of verifying such things as requirements completeness. However, simple tools can be developed to perform simple spelling and grammar checking (which is obviously undesirable but also can indicate ambiguity and incompleteness), flag key words that may be ambiguous (e.g., fast, reliable), and identify missing requirements (e.g., search for the phrase “to be determined”) and overly complex sentences (which can indicate unclear requirements).

4.11 SOME SURPRISES ABOUT CURRENT SOFTWARE SPECIFICATION PRACTICES

In Chapter 3, a 2002 survey of almost 200 software practitioners in a wide range of applications areas was mentioned. In that survey, several interesting notions about SRS practices were uncovered (Laplante et al, 2002e). While it is inappropriate to impute these findings exclusively to imaging systems, it is not unlikely that they apply. Some of the more surprising findings are discussed below.

4.11.1 SURPRISE 1

Object-oriented specification techniques have been not been significantly adopted by industry. Only 26% reported using object-oriented techniques. About the same percentage used structured techniques. Only 7% use formal methods and notations. Thirty-one percent responded that they used no software methodology at all.

4.11.2 SURPRISE 2

When formal methods are used, there is a perceived penalty in reduced ease of use in the end product. Only 70% of respondents who used formal representations agreed with the statement that “the capabilities of the finished product fitted well with customer or user needs,” compared to 79% that indicated informal and 77% that indicated semiformal. Similar results were found for the statement “end users found the finished product easy to use”: 62%, formal; 70%, informal; and 73%, semiformal.

4.11.3 SURPRISE 3

While there is a belief that not enough requirements engineering is occurring, this does not seem to negatively affect customer perceptions of the end product. Fifty-two percent of respondents did not think that their company did enough requirements

engineering; 29% thought that enough was being done. Fifty-nine percent indicated that they are performing requirements inspections; 32% are not using them; and 9% did not know. Of those respondents who felt that they did enough requirements engineering, 84% thought that the product fit the customer's needs and 73% thought that the product was easy to use. Of those who felt that their company did not do enough requirements engineering, 79% thought the product fit the customer's needs and 70% thought the product was easy to use.

4.11.4 SURPRISE 4

Using object-oriented methods is perceived to lead to systems that are easier to use and more fitting of customer needs than structured methods. Of those that used OOA, 78% felt that the capabilities of the finished product fit well with customer needs. This compares to 74% for users of structured methods — only a 5% difference. For perceived end-product usability, 76% of the OOA practitioners felt that the end users found the product easy to use, but only 61% of the SA practitioners held that opinion.

4.12 EXERCISES

- 4.1 What are the problems and ramifications of translating a requirement from one modeling technique (e.g., FSMs) to another (e.g., Petri nets)?
- 4.2 Who should write, analyze, and critique SRSs?
- 4.3 Under what circumstances and when should SRSs be changed?
- 4.4 Use statecharts instead of FSMs to represent the visual inspection system as it is described in the examples. Do the same using Petri nets.
- 4.5 Redraw the use case diagram for the visual inspection system in Figure 4.16 to include the calibration and diagnostic modes.
- 4.6 For a system with which you are familiar, find three good requirements and three bad requirements in the SRS. Rewrite the bad requirements so that they comply with IEEE Standard 830.

5 Software System Design

Experience is a dear teacher, but the fool will learn from no other.

Benjamin Franklin

5.1 THE DESIGN ACTIVITY

The design activity is involved in identifying the components of the software design and their interfaces from the software requirements specification (SRS). The principle artifact of this activity is the software design description (SDD).

During the design period in particular, the engineer must design the software architecture, which involves the following tasks:

- Performing hardware and software trade-off analysis
- Designing interfaces to external components (hardware, software, and user interfaces)
- Designing interfaces between components
- Making the determination between centralized or distributed processing schemes
- Determining concurrency of execution
- Designing control strategies
- Determining data storage, maintenance, and allocation strategy
- Designing databases, structures, and handling routines
- Designing the startup and shutdown processing
- Designing algorithms and functional processing
- Designing error processing and error message handling
- Conducting performance analyses
- Specifying physical location of components and data
- Designing any test software identified in test planning
- Creating documentation for the system, including (if applicable):
 - Computer system operator's manual
 - Software user's manual
 - Software programmer's manual
- Conducting internal reviews
- Developing the detailed design for the components identified in the software architecture
- Developing the test cases and procedures to be used in the formal acceptance testing

- Documenting the software architecture in the form of the SDD
- Presenting the detail design information at a formal design review

This intimidating set of tasks is further complicated because many of them must occur simultaneously or be iterated several times. There is no algorithm, per se, for conducting these tasks. Instead, it takes many years of practice, experience, learning from the experience of others, and good judgment to guide the software engineer through this maze of tasks.

Two methodologies, process- or procedural-oriented design and object-oriented design (OOD), are related to structured analysis and object-oriented analysis, respectively, and can be used to begin performing the design activities from the SRS produced by either structured analysis or structured design (SD). Each methodology seeks to arrive at a model containing small, detailed components.

Much of this chapter is based on a series of papers that first studied the nature of software specification and design in imaging systems (Laplante and Neill, 2002a, 2003b; Laplante et al., 2002c, 2002e; Neill and Laplante, 2002b, 2003a).

5.2 PROCEDURAL-ORIENTED DESIGN

Procedural-oriented design methodologies, such as SD, involve top-down or bottom-up approaches centered on procedural languages such as C and Fortran. The most common of these approaches utilizes design decomposition via Parnas partitioning.

5.2.1 PARNAS PARTITIONING

Software partitioning into software units with low coupling and high cohesion can be achieved through the principle of information hiding. In this technique, a list of difficult design decisions or things that are likely to change is prepared. Modules are then designated to “hide” the eventual implementation of each design decision or feature from the rest of the system. Thus, only the function of the module is visible to other modules, not the method of implementation. Changes in these modules are therefore not likely to affect the rest of the system.

This form of functional decomposition is based on the notion that some aspects of a system are fundamental, whereas other are arbitrary and likely to change. Moreover, it is those arbitrary things, which are likely to change, that contain “information.” Arbitrary facts are hard to remember and usually require lengthier descriptions; therefore, they are the sources of complexity.

The following steps can be used to implement a design that embodies information hiding:

1. Begin by characterizing the likely changes.
2. Estimate the probabilities of each type of change.
3. Organize the software to confine likely changes to a small amount of code.
4. Provide an “abstract interface” that abstracts from the differences.
5. Implement “objects”; that is, abstract data types and modules that hide changeable data structures.

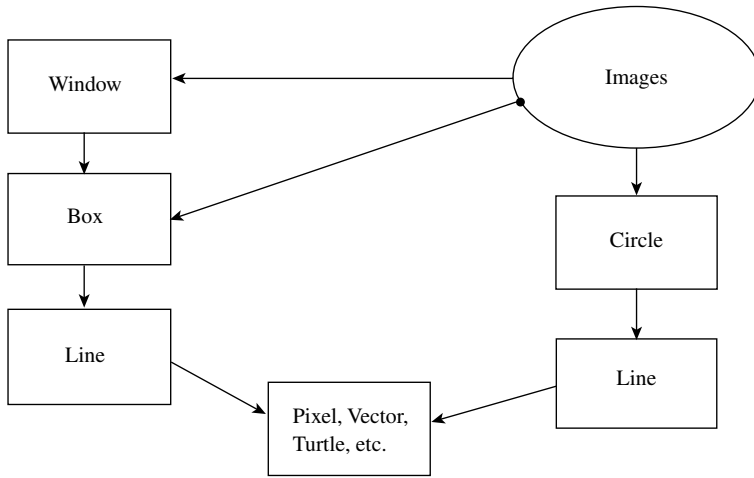


FIGURE 5.1 Parnas partitioning of graphics-rendering software.

These steps reduce coupling and increase module cohesion. Parnas also indicated that although module design is easy to describe in textbooks, it is difficult to achieve. He suggested that extensive practice and examples are needed to illustrate the point correctly (Parnas, 1979).

As an example, consider a portion of the display function of the visual inspection system shown in hierarchical form in Figure 5.1. It consists of graphics that must be displayed (for example, a representation of the conveyor system, units moving along it, sensor data, and so forth) that are essentially composed from circles and boxes. Different objects can also reside in different display windows. The implementation of circles and boxes is based on the composition of line-drawing calls. Thus, line drawing is the most basic hardware-dependent function. Whether the hardware is based on pixel, vector, turtle, or another type of graphics does not matter; only the line-drawing routine needs to be changed. Hence, the hardware dependencies have been isolated to a single code unit.

Parnas partitioning “hides” the implementation details of software features, design decisions, low-level drivers, etc., in order to limit the scope of impact of future changes or corrections. By partitioning things likely to change, only that module need be touched when a change is required, without the need to modify unaffected code. This technique is particularly applicable and useful in embedded systems; since they are so directly tied to hardware, it is important to partition and localize each implementation detail with a particular hardware interface. This allows easier future modification due to hardware interface changes and reduces the amount of code affected.

If in designing the software modules, increasing level of detail is deferred until later, subordinate code units, then the software approach is top-down. On the other hand, if the design detail is dealt with first and then increasing levels of abstraction are used to encapsulate those details, the approach is bottom-up.

For example, in Figure 5.1 it would be possible to design the software by first describing the characteristics of the various components of the system and the

functions that are to be performed on them, such as opening, closing, and sizing windows. Then the window functionality could be decomposed into its constituent parts, such as boxes and text. Still further, these could be decomposed; for example, all boxes consist of lines, and so on. The top-down refinement continues until the lowest level of detail needed for code development has been reached.

Alternatively, it is possible to begin by encapsulating the details of the most volatile part of the system, the hardware implementation of a single line or pixel, into a single code unit. Then working upward, increasing levels of abstraction are created until the system requirements are satisfied. This is a bottom-up approach to design.

In the object-oriented paradigm, Parnas partitioning is often referred to as protected variation.

5.2.2 STRUCTURED DESIGN

SD is the companion methodology to structured analysis. It is a systematic approach concerned with the specification of the software architecture and involves a number of techniques, strategies, and tools. Further, it provides a step-by-step design process intended to improve software quality, reduce risk of failure, and increase reliability, flexibility, maintainability, and effectiveness.

The data flow diagrams (DFDs) partition system functions and document that partitioning inside the specification.

5.2.2.1 Transitioning from Structured Analysis to Structured Design

Structured analysis is related to SD in the same way that a requirements representation is related to the software architecture; that is, the former is functional and flat, and the latter is modular and hierarchical. Data structure diagrams give information about logical relationships in complex data structures.

The transition mechanisms from structured analysis to SD are manual and involve significant analysis and trade-offs of alternative approaches. Normally, SD proceeds from structured analysis in the following manner. Once the context diagram is drawn, a set of DFDs is developed. The first DFD, the level 0 diagram, shows the highest level of system abstraction. Decomposing processes to lower and lower levels until they are ready for detailed design renders new DFDs with successive levels of increasing detail. This decomposition process is called leveling.

Typically, DFD boxes represent terminators that are labeled with a noun phrase that describes the system, agent, or device from which data enter or to which data exit. Each process, depicted by a circle, is labeled as a verb phrase describing the operation to be performed on the data, although it may be labeled with the name of a system or operation that manipulates the data. Solid arcs are used to connect terminators to processes and between processes to indicate the flow of data through the system. Each arc is labeled with a noun phrase that describes the data. Dashed arcs are discussed later. Parallel lines indicate data stores, which are labeled by a noun phrase naming the file, database, or repository where the system stores data.

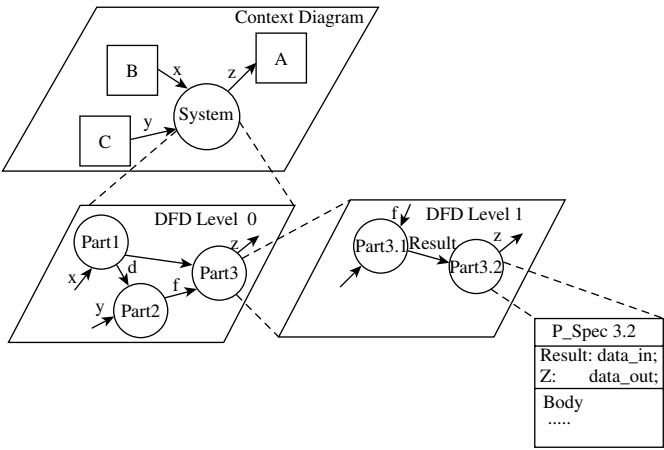


FIGURE 5.2 Context diagram evolution from level 0 DFD to level 1 DFD and, finally, to a P-SPEC, which is suitable for coding. (From Laplante, P.A. and Neill, C.J., *J. Electron. Imaging*, 12, 252–262, 2003.)

Each DFD should have between three and nine processes only. The descriptions for the lowest level, or primitive, processes are called process specifications (P-SPECs) and are expressed in either structured English, pseudo-code, decision tables, or decision trees and are used to describe the logic and policy of the program (Figure 5.2).

Returning to the visual inspection system example, Figure 5.3 shows the level 0 DFD. Here the details of the system are given at a high level. First, the system

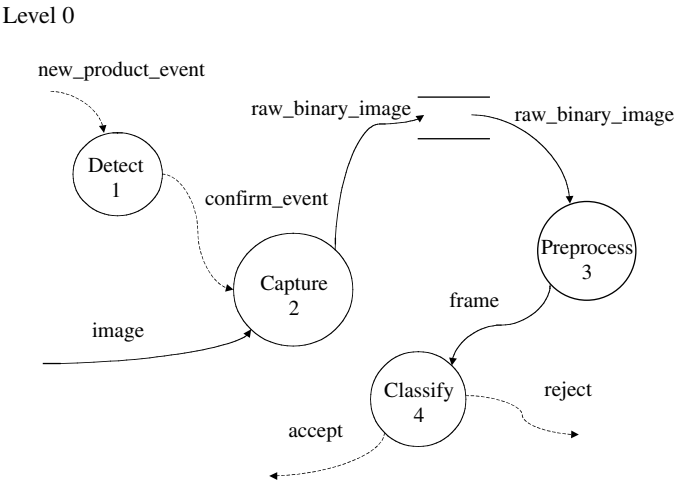


FIGURE 5.3 The level 0 DFD for the visual inspection system; the dashed arcs represent control flows, which are described later. (From Laplante, P.A. and Neill, C.J., *J. Electron. Imaging*, 12, 252–262, 2003.)

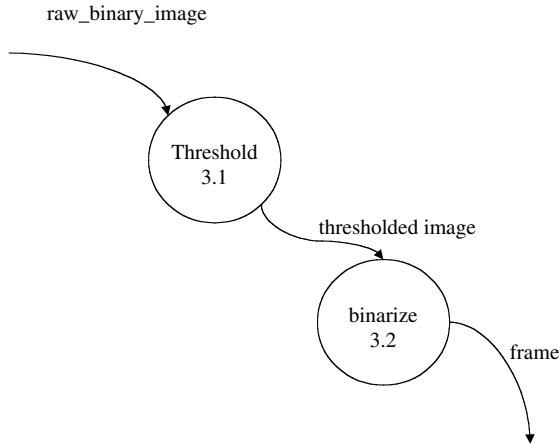


FIGURE 5.4 The level 1 DFD for process 3, preprocessing, of the visual inspection system. (From Laplante, P.A. and Neill, C.J., *J. Electron. Imaging*, 12, 252–262, 2003.)

reacts to the arrival of a new product by confirming that the image data are available. Next, the system captures the image by buffering the raw data from the capture device to a file. Preprocessing of the raw data is performed to produce an image frame to be used for classification and generation of the appropriate control signals to the conveyor system.

Proceeding to the next level provides more detail for processes 1 to 4. Process 1 is essentially an interrupt service routine assigned to a photodiode detector that senses when a new product for inspection reaches the designated point on the conveyor. Process 2 is a buffering routine whose characteristics depend on the specifications of the camera. Hence, without knowing these details, it is not possible to go deeper into the design.

Figure 5.4 depicts the level 1 DFD for process 3. Notice how the internal processes 3.1 and 3.2 are labeled to denote that they are a finer degree of detail of process 3 shown in the level 0 diagram. Successive levels of detail will follow a similar numbering system (e.g., 3.1.1, 3.1.2). This convention provides for simple traceability from specification through design and on to the code. Proceeding with the design example, Figure 5.5 shows the level 1 DFD for process 4.

In addition to the DFDs, SD uses a data dictionary to document and control interfaces. Entity relationship diagrams are frequently used to define the relationship between the components of the system — much as in the object-oriented paradigm. The data dictionary documents each interface flow in the DFD. Data structure diagrams are also used to describe information about logical relationships in complex data structures. The entity relationship model (which is optional) and data dictionary for the visual inspection system are not shown, for brevity's sake.

5.2.2.2 Data Dictionaries

A data dictionary is a repository of data about data that describes every data entity in the system. The data dictionary is an essential component of the SD. The data

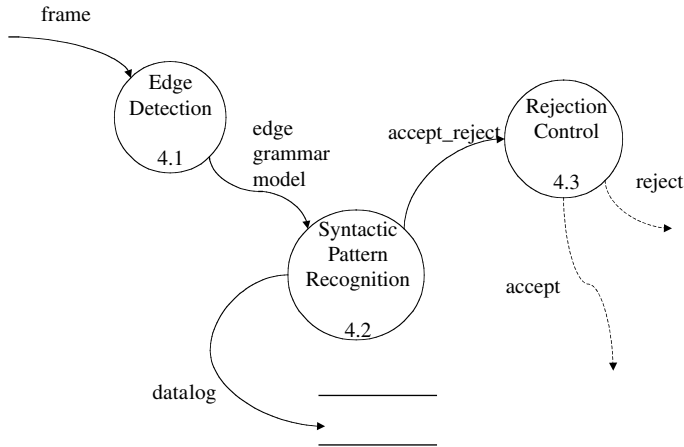


FIGURE 5.5 The level 1 DFD for process 4, classification, of the visual inspection system. (From Laplante, P.A. and Neill, C.J., *J. Electron. Imaging*, 12, 252–262, 2003.)

dictionary includes entries for data flows, control flows, data stores, data elements, and control elements. Each entry is identified by name, range, rate, units, etc. The dictionary is organized alphabetically for ease of use.

There is no standard format, but every design element should have an entry in the data dictionary. Most CASE tools provide the data dictionary feature. For example, each entry might be organized to contain the following information:

Entry type (data flow, data store, terminator, process)

Name

Alias

Description

Found in

In particular, for the visual inspection system, one entry might look like:

Type: Data flow

Name: Binarized image

Alias: Image

Description: The raw binary image after it has been subjected to thresholding

...

Found in:

The missing information for the “Found in” module will be added as the code is developed. In this way, data dictionaries help provide substantial traceability between code elements.

5.2.2.3 Problems with SASD in Imaging Applications

There are several apparent problems in using structured analysis and structured design (SASD) to model the visual inspection system, including difficulty in mod-

eling time and events. For example, what if the visual inspection system captures a new image in parallel with preprocessing of the last image capture? Concurrency is not easily depicted in this form of SASD.

Another problem arises in the context diagram. Control flows are not easily translated directly into code, such as reject and accept, because they are hardware dependent. In addition, the control flows do not really make sense since there is no connectivity between portions of them, a condition known as floating.

Details of the detector and camera hardware also need to be known for further modeling of process 1. What happens if the hardware changes? What if a different strategy for classification in process 2 is needed? In the case of process 3 (preprocessing), what if the algorithm or even the sensitivity levels change because of the installation of new hardware? In each case, the changes would need to propagate into the level 1 DFD for each process, any subsequent levels, and, ultimately, into the code.

Clearly, making and tracking any of these changes is fraught with danger. Moreover, any change means that significant amounts of code would need to be rewritten, recompiled, and properly linked with the unchanged code to make the system work. None of these problems arise when using the object-oriented paradigm.

5.2.2.4 Real-Time Extensions of SASD

It is well known that the standard SASD methodology is not well equipped for dealing with time, as it is a data-oriented and not a control-oriented approach. In order to address this shortcoming, Hatley and Pirbhai (1987) extended the SASD method by allowing for the addition of control flow analysis. To do this, the following artifacts were added to the standard approach: arcs made of dashed lines to indicate the flow of control information, and solid bars indicating “stored” control commands (control stores), which are left unlabeled (Hatley and Pirbhai, 1987).

Additional tools, such as Mealy finite state machines, are used to represent the encapsulated behavior and process activations. The addition of the new control flows and control stores allows for the creation of a diagram containing only these elements, called a control flow diagram (CFD). These CFDs can be decomposed into control specifications (C-SPECs), which can then be described by a finite state machine. The relationship between the control and process models is shown in Figure 5.6.

Although the Hatley–Pirbhai extensions suggest that the CFD and C-SPECs stand alone, the CFD by itself makes little sense (see Figure 5.7). Hence, the CFD and DFD are generally combined, as shown in Figure 5.5.

5.2.3 DESIGN IN PROCEDURAL FORM USING FINITE STATE MACHINES

One of the advantages of using finite state machines in the SRS and later in the software design is that they are easily converted to code and test cases. Again

* This scenario would be desirable if the reject mechanism were further down the inspection line and the conveyor system were running at a high rate.

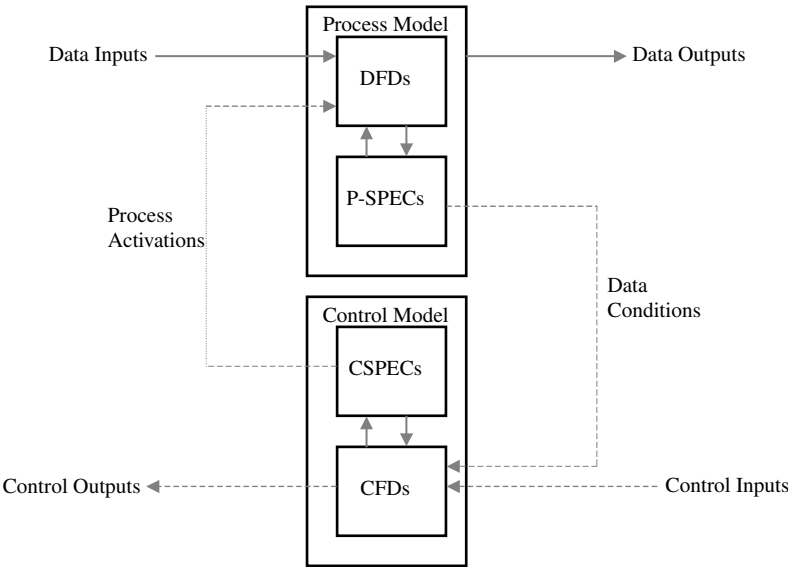


FIGURE 5.6 The relationship between the control and process models.

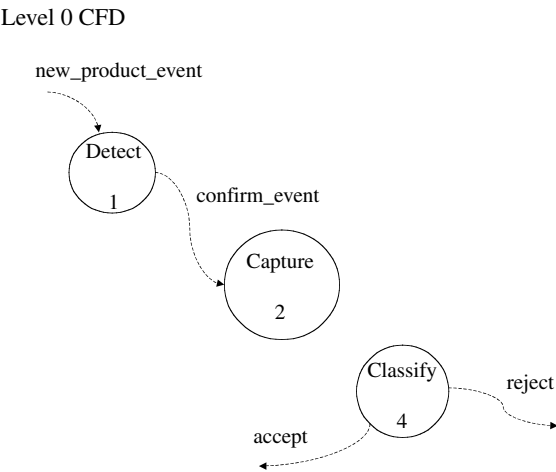


FIGURE 5.7 A CFD for level 0 of the visual inspection system.

consider the visual inspection system. The tabular representation of the state transition function (Table 4.2), which describes the system’s high-level behavior, can be easily transformed into a design using the Pascal-like pseudo-code shown in Figure 5.8.

Each procedure associated with the three operational modes (operational, diagnostic, and calibration) will be structured code that can be viewed as executing in one of any number of process states at an instant in time. This functionality can be described by the pseudo-code shown in Figure 5.9.

```

typedef states: (state1,...,staten);    {n is# of states}
    alphabet: (input1,...,inputn);
    table_row: array [1..n] of states;
procedure move_forward;    {advances FSM one state}
var
    state: states;
    input: alphabet;
    table: array [1..m] of table_row; {m is the size of the alphabet}
begin
    repeat
        get(input);{read one token from input stream}
        state:=table[ord(input)] [state];    {next state}
        execute_process (state);
    until input = EOF;
end;

```

FIGURE 5.8 Pascal-like pseudo-code that can implement the system behavior of the visual inspection system depicted in Table 4.2.

```

Procedure execute_process (state: states);
begin
    case state of
        state 1: process 1; {execute process 1}
        state 2: process 2; {execute process 2}

        ...
        staten: processn;    {execute process n}
    end

```

FIGURE 5.9 FSM code for executing a single operational process of the visual inspection system. Each process can exist in multiple states, allowing for partitioning of the code into appropriate modules.

The pseudo-codes shown in Figure 5.8 and Figure 5.9 can easily be coded in any procedural language.

5.3 OBJECT-ORIENTED DESIGN

Object-oriented programming languages are those characterized by data abstraction, inheritance, and polymorphism. Data abstraction has been previously defined. Inheritance allows the software engineer to define new objects in terms of previously defined objects so that the new objects “inherit” properties. Function polymorphism allows the programmer to define operations that behave differently depending on the type of object involved. For example, a filter operation would act differently depending on the type of image and filtering needed. How the filter operation is applied is implemented at run time.

Object-oriented languages provide a natural environment for information hiding, through encapsulation. The state (or data) and behavior (or methods) of objects are

encapsulated and accessed only via a published interface or private methods. For example, in image processing systems, one may wish to define a class of type pixel with characteristics (attributes) describing its position, color, brightness, and so on, and operations that can be applied to a pixel, such as add, activate, deactivate, and so on. The engineer may then wish to define objects of type image as a collection of pixels with other attributes, and so on. In some cases, expression of system functionality is easier to do in an object-oriented manner.

OOD is an approach to systems design that views the system components as objects and data processes, control processes, and data stores that are encapsulated within objects. Early forays into OOD were led by attempts to reuse some of the better features of conventional methodologies, such as the DFDs and entity relationship models, by reinterpreting them in the context of object-oriented languages. This can be seen in the unified modeling language (UML).

Over the last several years the object-oriented framework has gained significant acceptance into the software engineering community.

5.3.1 BENEFITS OF OBJECT ORIENTATION

The previous section has highlighted some considerations concerning the appropriateness of the object-oriented paradigm to various application areas. There are, however, some additional benefits to using object-oriented analysis and design. When considering the benefits of object-oriented approaches to image processing applications, it is easy to get wrapped up in the ideas of combining data and behavior into an encapsulated entity that better approximates the “things” in our problem domain, and to consider this closeness between reality and the modeling domain to be the central benefit of using objects. While this can be considered an advantage, the purported intuitiveness of the approach is, in fact, something of a fortuitous side effect. The real advantages of applying object-oriented paradigms are the future extensibility and reuse that can be attained and the relative ease of future changes.

Software systems are subject to near-continuous change: requirements change, merge, emerge, and mutate; target languages, platforms, and architectures change; and, most significantly, the way the software is employed in practice changes. This flexibility places considerable burden on the software design: How can systems that must support such widespread change be built without compromising quality? There are four basic principles of object-oriented engineering that can answer this question, and they have been recognized as supporting reuse.

5.3.1.1 Open–Closed Principle

First recorded by Meyer (1998), the open–closed principle (OCP) states that classes should be open to extension, but closed to modification. That is, it should be possible to extend the behavior of a class in response to new or changing requirements, but modification to the source code is not allowed. While these expectations may seem at odds, the key is abstraction. In OOD, a superclass can be created that is fixed, but can represent unbounded variation by subclassing. This is evident in the above case study in the classification strategies, where subclasses for each of the various classification algorithms are created, which inherit their interface from an abstract superclass. This

aspect is clearly superior to structured approaches and top-down design in, for example, changes in classification strategies, which would require new function parameter lists and wholesale recompilation of any modules calling that code in the SD.

5.3.1.2 Once and Only Once

While once and only once (OAOO) is certainly not a new idea, Beck (1999) put a name to the principle that any aspect of a software system — be it an algorithm, a set of constants, documentation, or logic — should exist in only one place. This isolates future changes and makes the system easier to comprehend and maintain, and, through the low coupling and high cohesion that the principle instills, increases the reuse potential of these aspects. The encapsulation of state and behavior in objects, and the ability to inherit properties between classes, allows for the rigorous application of these ideas in an object-oriented system, but is difficult to implement in structured techniques.

5.3.1.3 Dependency Inversion Principle

The dependency inversion principle (DIP) states that high-level modules should not depend upon low-level modules. Both should depend upon abstractions. Restated, abstractions should not depend upon details, but details should depend upon abstractions. Martin (1996) introduced this idea as an extension to OCP with reference to the proliferation of dependencies that exist between high- and low-level modules. For example, in a structured decomposition approach, the high-level procedures reference the lower-level procedures, but changes often occur at the lowest levels. This infers that high-level modules or procedures that should be unaffected by such detailed modifications may be affected due to these dependencies. Again, consider the case where the camera characteristics change and, even though perhaps only one routine needs to be rewritten, the calling module(s) need to be modified and recompiled as well. A preferable situation is to invert these dependencies, such as is evident in the Liskov substitution principle (LSP). This principle is at work in the decorator pattern, which is used in the object-oriented case study for the image preprocessing. The intent here is to allow dynamic changes in the preprocessing scheme, which is achieved by ensuring that all image processing objects conform to the same interface, and are therefore interchangeable.

5.3.1.4 Liskov Substitution Principle

Liskov (1988) expressed the principle of substitutivity of subclasses for their base classes as:

What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T .

This principle has led to the concept of type inheritance and is the basis of polymorphism in object-oriented systems, where instances of derived classes can be

substituted for each other provided they fulfill the obligations of a common super-class. Again, the strategies and decorators introduced in the object-oriented case study implement this type of conformance and substitutability such that any new variations that are desired need merely conform to the supertypes, yet no existing code need be modified.

5.3.2 DESIGN PATTERNS

Developing software is hard, and developing reusable software is even harder. Designs should be specific to the current problem, but general enough to address future problems and requirements. Experienced designers know not to solve every problem from first principles, but to reuse solutions encountered previously. They find recurring patterns and use them as a basis for new designs. This is simply an embodiment of the principle of generality.

While object-oriented systems can be designed to be as rigid and resistant to extension and modification as in any other paradigm, OOD has the ability to include distinct design elements that can cater to future changes and extensions. These design patterns were first introduced to the mainstream of software engineering practice by Gamma et al. (1994). The Gamma, Helm, Johnson, and Vlissides patterns, or more commonly, the “gang of four” (GoF) patterns, are based on the four key design principles that have just been discussed.

The formal definition of a pattern is not consistent in the literature. Simply, a pattern is a named problem–solution pair that can be applied in new contexts, with advice on how to apply it in novel situations.

Patterns can be distinguished as three types: architectural patterns, design patterns, and idioms. An architectural pattern occurs across subsystems; a design pattern occurs within a subsystem but is independent of the language. An idiom is a low-level pattern that is language specific.

In general, a pattern consists of four essential elements: a name, such as strategy, bridge, façade, and so on; the problem to be solved; the solution to the problem; and the consequences of the solution.

More specifically, the problem describes when to apply the pattern in terms of specific design problems, such as how to represent algorithms as objects. The problem may describe class structures that are symptomatic of an inflexible design. Finally, the problem section may include conditions that must be met before it makes sense to apply the pattern.

The solution describes the elements that make up the design, though it does not describe a particular concrete design or implementation. Rather, the solution provides how a general arrangement of objects and classes solves the problem.

There are several sets of patterns, known as pattern languages, but perhaps the most famous is the GoF, popularized in a well-known text (Gamma et al., 1994). It describes 23 patterns, each organized by being either creational, behavioral, or structural in its intent (Table 5.1).

Table 5.1 is provided for illustration only and is not intended to be a detailed description. Instead, a few of these will be introduced later in the context of the ongoing case study.

TABLE 5.1
Set of Design Patterns Popularized by the GoF

Creational	Behavioral	Structural
Abstract factory	Chain of responsibility	Adapter
Builder	Command	Bridge
Factory method	Interpreter	Composite
Prototype	Iterator	Decorator
Singleton	Mediator	Facade
	Memento	Flyweight
	Observer	Proxy
	State	
	Strategy	
	Template method	
	Visitor	

**5.3.3 OBJECT-ORIENTED DESIGN USING UNIFIED
MODELING LANGUAGE**

The UML is widely accepted as the *de facto* standard language for the specification and design of software-intensive systems using an object-oriented approach. By bringing together the “best of breed” in specification techniques, the UML has become a family of languages (diagram types), and users can choose which members of the family are suitable for their domain.

The UML is a graphical language based upon the premise that any system can be composed of communities of interacting entities and that various aspects of those entities, and their communication, can be described using the set of nine diagrams: use case, sequence, collaboration, statechart, activity, class, object, component, and deployment. Of these, five render behavioral views (use case, sequence, collaboration, statechart, and activity), while the remaining diagrams are concerned with architectural or static aspects.

With respect to imaging systems, it is these behavioral models that are of interest. The use case diagrams document the dialog between external actors and the system under development. Sequence and collaboration diagrams describe interactions between objects. Activity diagrams illustrate the flow of control between objects and statecharts and represent the internal dynamics of active objects. The principle artifacts generated when using the UML and their relationships are shown in Figure 5.10.

While not aimed specifically at embedded system design, some notion of time has been included in the UML through the use of sequence diagrams.

5.3.4 MODELING TIME EXPLICITLY

It is clear from the description above that the UML in its current form does not provide sufficient facilities for the specification and analysis of real-time systems.

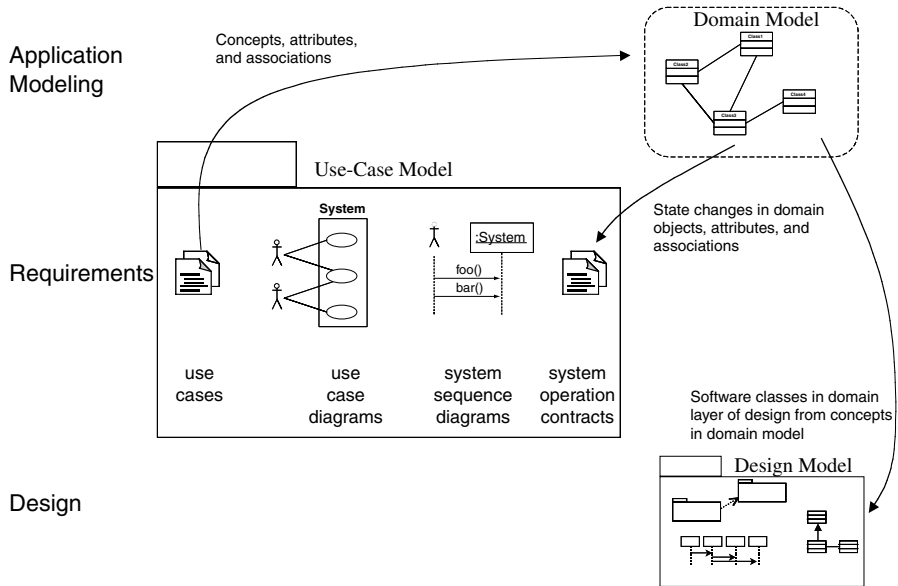


FIGURE 5.10 The UML and its role in specification and design. (Adapted from Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd ed., Prentice Hall, New York, 2002.)

It is also stated, however, that the UML is a family of languages, and there is no compelling reason for not adding to the family if a suitable language is found. Unfortunately, the majority of appropriate candidates are formal methods — specification languages with a sound mathematical basis — and these are traditionally shunned by the user community. An approach intended to overcome this is generated from annotated UML specifications via the Q-model. Much of the discussion that follows has been adapted from Neill and Laplante (2003a).

The Q-model was originally developed by Quirk and Gilbert (1977) at the U.K. Atomic Energy Research Establishment and was intended as a formal method for the description of temporal characteristics of complex real-time systems. The method was advanced by Motus and Rodd (1994), and a series of prototype tools were developed to support model simulation and verification. The Q-model is based upon the concept that a real-time computer system is composed of a set of loosely coupled, repeatedly activated terminating processes. The couplings, termed *channels*, define the synchronization properties of the interprocess communication, as well as the time selectivity of data transfer between processes. The resultant model can then be checked for completeness, noncontradiction, and correctness, as well as for timeliness, liveness, and constraint conformance.

The crucial aspect of the Q-model that makes it suitable for the specification and design of real-time systems is the ability to specify and subsequently verify and validate the temporal characteristics of the proposed system. To achieve this, a number of process parameters must be defined (illustrated graphically in Figure 5.11, where P_1 and P_2 represent two processes):

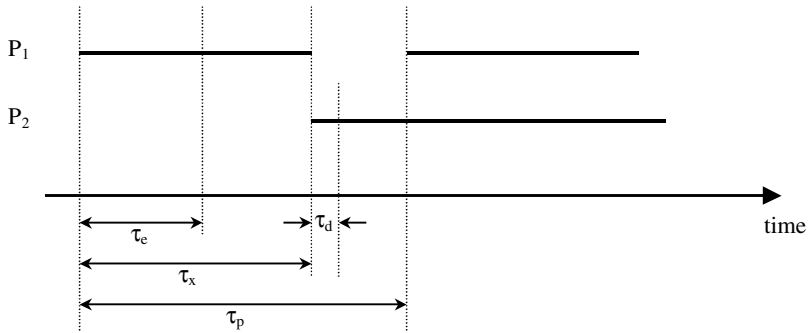


FIGURE 5.11 Temporal parameters of individual processes. (From Neill, C.J. and Laplante, P.A., Specification of real-time imaging systems using UML, *Real-Time Imaging*, 9, 2, 125–137, 2003. With permission from Elsevier.)

- Process time set — The set of all start times for a process. This set can be expressed by both extension and comprehension, or can be linked to an external event or to the time set of another process.
- Process start period (τ_p) — Every process that does not have its time set defined by another process (has no synchronous or sequential channel inputs) must have a start period defined.
- Process execution time (τ_x) — Every process must have an interval defined that describes the best- and worst-case execution times.
- Data consumption time (τ_d) — An interval can be defined for any process that consumes data that describes an interval that must elapse before data is consumed by that process. Again, this parameter is defined by minimum and maximum values.
- Equivalence interval (τ_e) — This is defined as the time interval during which all occurring events can be considered to have occurred simultaneously. A side effect of this is that the equivalence interval defines the time that must elapse before a process can be reactivated.

In addition, there are two further parameters that apply to groups of processes called synchronous clusters, illustrated in Figure 5.12. A synchronous cluster is formed when two or more processes are linked with synchronous channels such that they all activate at the same instant. This simultaneous activation is physically impossible, however, so the two parameters are defined to specify the interval during which all of the cluster processes will activate.

- Null channel delay (ξ) — This represents the time necessary to detect the synchronizing event and then activate the processes (as this would likely be a broadcast mechanism).
- Simultaneity interval (τ_s) — This is the interval, following the null channel delay, within which all the synchronized processes will be activated.

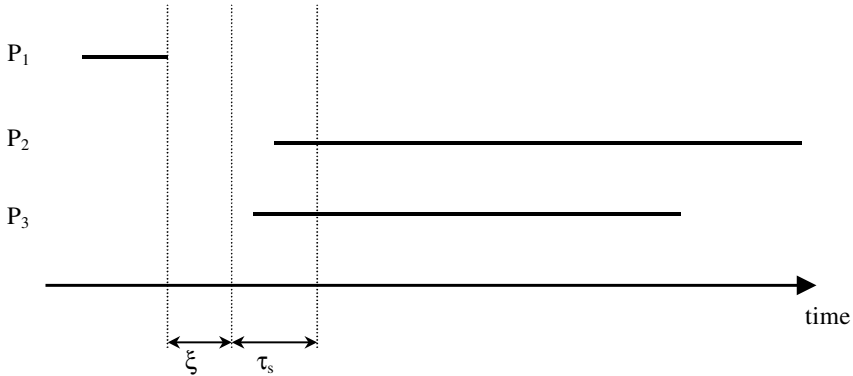


FIGURE 5.12 Temporal parameters of synchronous clusters. (From Neill, C.J. and Laplante, P.A., Specification of real-time imaging systems using UML, *Real-Time Imaging*, 9, 2, 125–137, 2003. With permission from Elsevier.)

Once the Q-models have been constructed and the parameters specified, formal analysis can be performed for verification and simulations can be run for validation.

There are three levels to the analysis of a Q-model specification. These are checks on:

1. Individual P-SPECs — This is the simplest level of analysis and is concerned only with the time parameters for each process. This includes checking that every process has a defined time set or a pointer to a time set via a channel, that the elements of the time set are well ordered, that a valid channel function has been specified, and that the other parameters have been set (execution time, equivalence interval, data consumption time, etc.).
2. Process pair interaction — All communication within a Q-model is via channels. Several types of channels have been defined, and each of these types imposes different rules on the interprocess communication. For each group of synchronously activated processes (synchronous clusters), the null channel delay and simultaneity interval must be determined (from the dynamic properties of the environment). The analysis then verifies that dependencies and channel functions do not violate rules concerning processes waiting for input data or having their process time sets altered. For groups of sequentially activated processes, the analysis is more complicated, since the consumer process time set is generated by the producer process and one consumer process could conceivably have many producer processes. In this situation, the equivalence interval determines whether the consumer process can be activated (successive activations of the process will be inhibited until the interval has elapsed). For asynchronously connected processes, the pair-wise analysis is only interested in the estimation of delays that occur during communication.

3. Process group behavior — This is the most complex level of analysis performed on the Q-model. It is here that information deadlocks (“circular” message-waiting conditions) are checked for in sequential chains and synchronous loops, message transfer paths are analyzed, and the time required for data to pass through each path is calculated to verify the time constraints imposed upon the designed system by the environment.

This is only a brief overview of the analysis capabilities of the Q-model; it is provided to highlight the usefulness of the approach. A thorough treatment is given in Motus and Rodd (1994).

5.3.5 VISUAL INSPECTION SYSTEM CASE STUDY

Given the utility of a formal method such as the Q-model, the obvious next question is “Why use the UML, or any other less formal modeling language?” There are many answers to this question, ranging from the need for uniform communication standards (and the predominance of the UML in that arena) to the lack of appreciation and willingness of practitioners to adopt rigorous representations (in practical terms, very few software development professionals possess the appropriate mathematical skills).

To overcome these issues, and yet still maintain rigor, an approach is needed to transform the behavioral UML models into Q-model structures. This transformation diminishes the exposure of the Q-model to the user and allows for an essentially standard object-oriented development process. Hence, the system specification begins with the representation of the externally accessible functionality as use cases.

However, each use case is a document that describes scenarios of operation of the system under consideration, as well as pre- and postconditions and exceptions. In the specification of an embedded system, this is also where overall time constraints, sampling rates, and deadlines are specified.

As stated above, the domain model is created based upon the use cases, and through further exploration of system behavior via the interaction diagrams, the domain model evolves systematically into the design class diagram. The construction of the domain model is therefore analogous to the analysis stage in SASD, described earlier. In domain modeling, the central objective is to represent the real-world entities involved in the domain as concepts in the domain model. This is a key aspect of object-oriented systems and is seen as a significant advantage of the paradigm since the resultant model is “closer” to reality than it is in alternative modeling approaches, including SASD. Part of the design class diagram that results from evolution of the domain model is shown in Figure 5.13.

The design class diagram is used to show the static structural view of the system by describing the classes of objects that will comprise the software solution. As can be seen in Figure 5.13, the system is composed of image capture elements (CameraProxy, FrameGrabber) and image classification elements (Classifier, ImageProcessor, decorators, and strategies). The decorators and strategies are aspects of the design introduced by applying well-known design patterns from Gamma et al. (1994). The principle aim of these patterns is to allow for dynamic (i.e., run time)

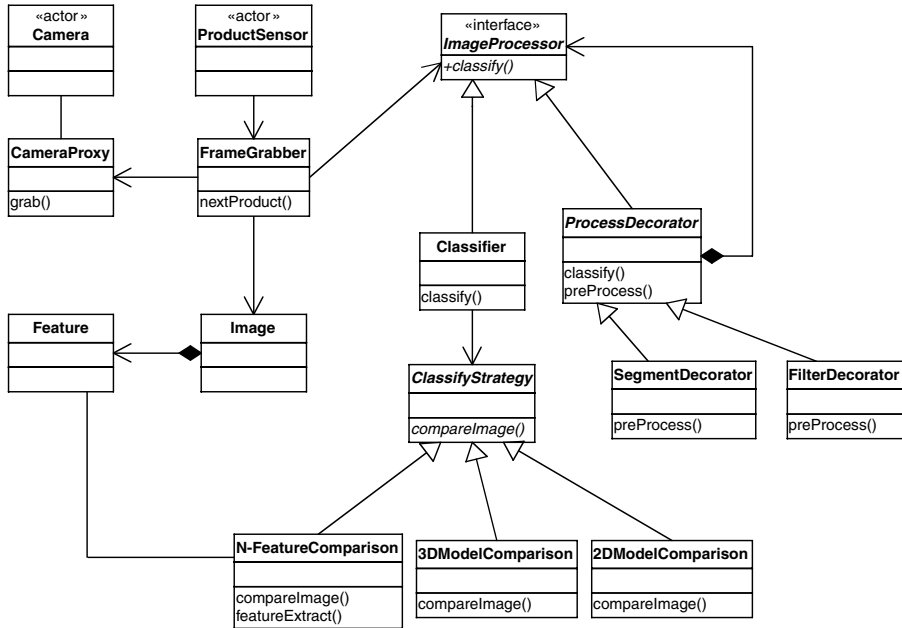


FIGURE 5.13 Partial design class diagram of the visual inspection system. (From Laplante, P.A. and Neill, C.J., *J. Electron. Imaging*, 12, 252–262, 2003.)

changes to the preprocessing and classification schemes. This is achieved by abstracting invariant behavior into supertypes (*ImageProcessor*, *ProcessDecorator*, and *ClassifyStrategy*) and allowing subtypes to implement variant behavior (*N-FeatureComparison*, *FilterDecorator*).

It can be argued that this generality is speculative, and therefore overcomplicates the design, but the intent of this work is to build a framework for image processing systems where this generality is critical for reuse. This also highlights a key advantage with object-oriented analysis and design: the reuse potential. In SASD development, it can be very difficult to extend the functionality of the complete system because of the degree and direction of dependencies that are created between high- and low-level modules. That is, high-level modules call lower-level modules as a consequence of the top-down decomposition approach. When changes need to be made at these lower levels, the more abstract elements must also be modified due to these dependencies. In object-oriented systems, this dependency hierarchy is normally inverted.

For example, the *N-FeatureComparison* classification strategy could involve syntactic pattern recognition, correlation measures, or morphological approaches, which all depend on the internal representation of the image. By subclassing *N-FeatureComparison*, this extension can be accommodated without modifying any existing code or affecting any other requirements. In the top-down SD fostered by SASD, this is not always the case; in fact, it is likely that every module in the system that used the feature comparison module would need to be rewritten.

Behavioral aspects of the design can be represented by a number of different diagrams in the UML. Perhaps the most popular choice is to use sequence diagrams. The sequence diagram shown in Figure 5.14 represents the ordering of messages between objects in the system in response to the arrival of the next product on the conveyor. It is clear that the `nextProduct` message is generated from the external sensor, which triggers the `FrameGrabber`. The image created by the output from the camera (via the `CameraProxy`) is preprocessed using one of the decorators (`FilterDecorator`) and then classified using a classification strategy (`N-FeatureComparison`). The result of the classification is then sent to the `RejectController`, which will log the result and trigger the rejection mechanism if required.

It is evident from these diagrams that no timing information is included beyond the logical timing (ordering) of the messages in the sequence diagram. This is insufficient for real-time systems, as stated earlier. Not only are we unable to prove the correctness of the temporal specification, but also we have neglected to represent a critical aspect of the domain in our model. To remedy this situation, we generate the additional formal model: the Q-model. This is generated from the UML models of the system (with additional information regarding timing properties and constraints).

The Q-model generated from the sequence diagram in Figure 5.14 is shown in Figure 5.15. The transformation rules applied are basically a mapping of activations to Q-model processes and messages to Q-model channels. The descriptions of the processes and channels in this Q-model are presented in Table 5.2 and Table 5.3, respectively, and are representative of typical timing information for automated visual inspection systems.

Initially all channels are selected as sequential (the source process triggers the target process upon completion), except for `k2`, which is sequential null, indicating that no data are transferred. The alternatives are synchronous (where the time sets of the producer and consumer processes are identical) and asynchronous (the producer and consumer processes have independent time sets — data are transferred, but there is no triggering of activation). The channel functions indicate the range of data generations that are consumed by the process (allowing for time-selective communication if necessary), but here all are set to the most recent generation only. During simulation, the channel types and functions can be altered to examine the effect of parallelizing aspects of the design.

Each Q-model process can then be decomposed into a separate Q-model, such as the one shown in Figure 5.16 for process `P6`, the `compareImage` process. In this case, elements of the Q-model structure are transformed from the `ClassifyStrategy` hierarchy, but individual algorithmic differences must be constructed directly. The fragment shown in Figure 5.16 is the Q-model of the `N-FeatureComparison` object. The bubbles represent processing tasks, and the directed arcs represent the channels that connect and coordinate the tasks.

Once the Q-model representation is complete, the model can be analyzed as described earlier. Typical timeline diagrams that are generated during simulation are shown in Figure 5.17 and Figure 5.18. It is clear from Figure 5.17 that if the process `P4–P6` chain was on a separate processor, the arrival rate could increase to every 20 msec.

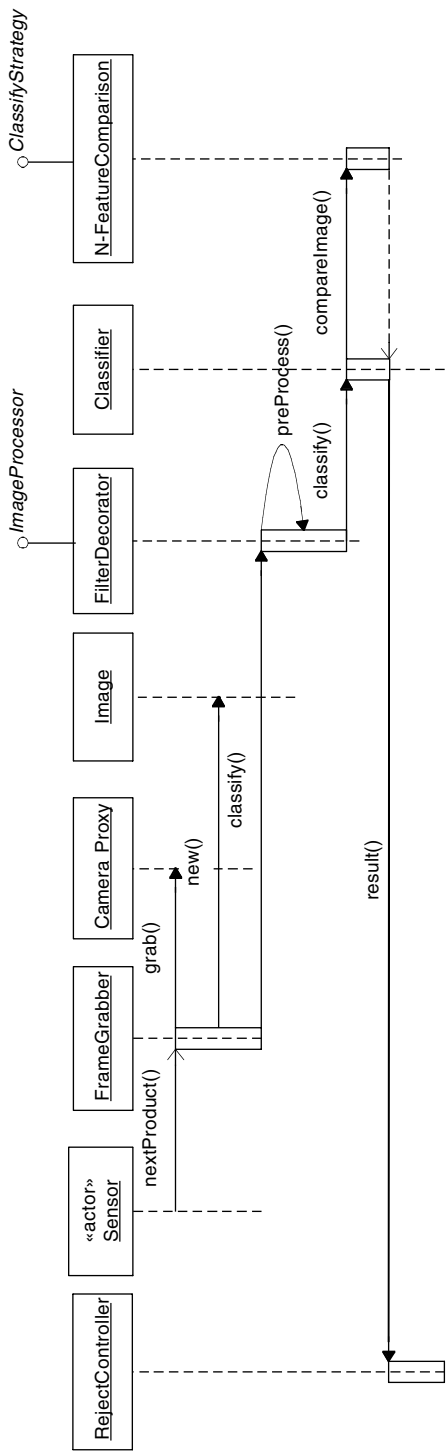


FIGURE 5.14 Sequence diagram of use case product classification. (From Laplante, P.A. and Neill, C.J., *J. Electron. Imaging*, 12, 252–262, 2003.)

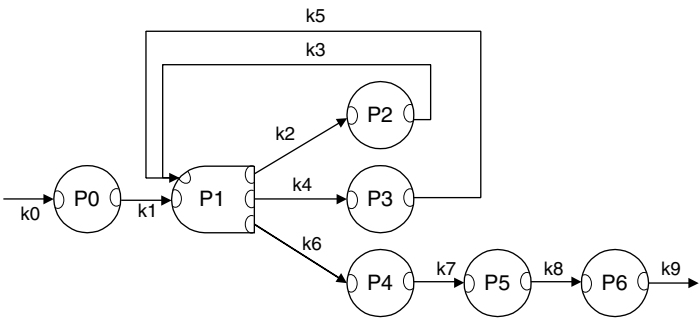


FIGURE 5.15 Q-model of product classification. (From Neill, C.J. and Laplante, P.A., Specification of real-time imaging systems using UML, *Real-Time Imaging*, 9, 2, 125–137, 2003. With permission from Elsevier.)

TABLE 5.2
Description of Product Classification Q-Model

Process	Name	τ_p (msec)	τ_x	τ_d	τ_e
P0	Null process (sense event)	33	0	0	31
P1	nextProduct()	—	1–2	0	1
P2	grab()	—	5–7	—	1
P3	new()	—	4–5	3–4	4
P4	classify().preProcess()	—	6–8	3–4	4
P5	_classify()	—	1–2	0	1
P6	compareImage()	—	7–10	3–4	4

Source: Adapted from Neill, C.J. and Laplante, P.A., *Real-Time Imaging*, 9, 2, 125–137, 2003.

TABLE 5.3
Channel Descriptions for Product Classification Q-Model

Channel	Type	Function
k0	Sequential	[0,0]
k1	Sequential	[0,0]
k2	Sequential null	—
k3	Sequential	[0,0]
k4	Sequential	[0,0]
k5	Sequential	[0,0]
k6	Sequential	[0,0]
k7	Sequential	[0,0]
k8	Sequential	[0,0]
k9	Sequential	[0,0]

Source: Adapted from Neill, C.J. and Laplante, P.A., *Real-Time Imaging*, 9, 2, 125–137, 2003.

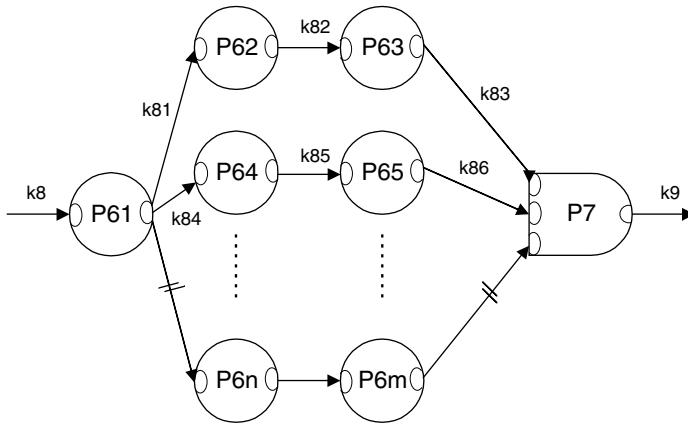


FIGURE 5.16 Q-model of compareImage for N-FeatureComparison classification strategy. (From Neill, C.J. and Laplante, P.A., Specification of real-time imaging systems using UML, *Real-Time Imaging*, 9, 2, 125–137, 2003. With permission from Elsevier.)

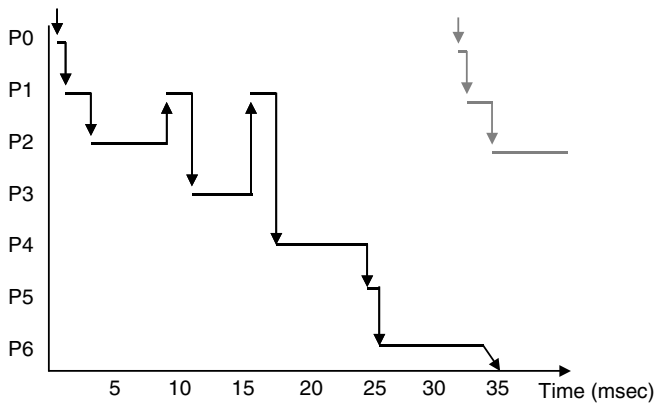


FIGURE 5.17 Simulation timeline for product classification Q-model. (From Neill, C.J. and Laplante, P.A., Specification of real-time imaging systems using UML, *Real-Time Imaging*, 9, 2, 125–137, 2003. With permission from Elsevier.)

In the case of Figure 5.18, the compareImage process is triggered by the arrival of a preprocessed image, which is then subject to two concurrent feature extraction and feature-matching chains, the results of which are aggregated in process P7. Since the temporal properties of each process are defined, the simulation can highlight the minimum and maximum arrival rates based upon the available resources.

5.4 HARDWARE CONSIDERATIONS IN IMAGING SYSTEM DESIGN

Understanding the underlying hardware of the imaging system during design helps one to use hardware and software resources more efficiently. Although the role of

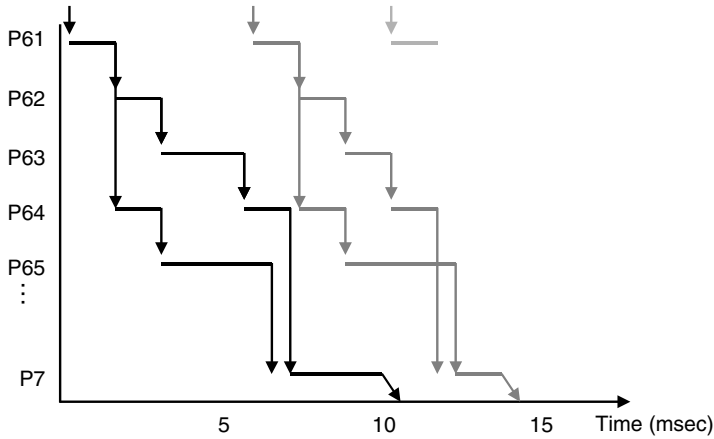


FIGURE 5.18 Simulation timeline for N-FeatureComparison process ($n = 2$). (From Neill, C.J. and Laplante, P.A., Specification of real-time imaging systems using UML, *Real-Time Imaging*, 9, 2, 125–137, 2003. With permission from Elsevier.)

programming languages is to isolate the programmer from the underlying hardware, those who have implemented practical embedded systems realize that this expectation is probably unrealistic — if not during design, certainly during development. While it is impossible to provide a complete review of hardware issues, a brief review with respect to software engineering issues is appropriate.

5.4.1 PROCESSORS

Most imaging systems are based on a microprocessor, but others involve mainframe or minicomputers, while still others are based in the microcontroller. A microcontroller is a computer system that is programmable via microinstructions. Because the complex macroinstruction decoding process is not supported, program execution tends to be very fast.

The basic architecture of a computer consists of a central processing unit (CPU), memory, and input and output devices connected by a bus. One at a time, the computer continuously fetches binary encoded instructions from memory and executes them. Computers that satisfy this basic description — stored program, serial fetch, and execution — are generically called von Neumann computers. The limiting factor in this type of architecture is the serial nature of the bus; that is, at any instant in time only one instruction or one datum can be on the bus. For data-intensive applications such as image processing, this limitation can be significant.

Other architectural features such as the instruction set and addressing modes can affect overall performance. The more complex the instruction set, the fewer the instructions needed to achieve a particular function, but the longer each individual instruction will take. For example, in reduced instruction set (RISC) architectures, short, simple instructions prevail and thus are ideal for high-performance applications.

Some computer configurations include a second specialized CPU, called a coprocessor, to perform certain operations such as digital signal processing (DSP) instruc-

tions. DSP coprocessors improve real-time performance because they extend the instruction set to support faster, specialized instructions. These devices typically are used to extend the instruction set, and not for multiprocessing. The main processor loads certain registers with data for the coprocessor, issues an instruction starting the coprocessor, and then suspends itself until the coprocessor finishes. Usually, this involves two handshaking signals between the main processor and coprocessor. For example, consider a typical microprocessor and associated coprocessor. Suppose there are two signals between them. When the main processor wishes to use the coprocessor, it loads global variables with the operands and sends a signal to the coprocessor. Then the main processor suspends itself. When the coprocessor finishes the operation, it places the result in another global variable, issues a signal to awaken the main processor, and suspends itself. The main processor then resumes its fetch–execute cycle.

5.4.2 NON-VON NEUMANN ARCHITECTURES

The limitations of the serial bus structure in most computer systems and the data-intensive needs of imaging applications have led to the use of a variety of non-von Neumann-style architectures in image processing systems. A brief description of these special computing environments in the context of software engineering is valuable, because the design must be such that it exploits the advantages of the underlying hardware. In other words, while the architecture is often selected to fit the application, the application must be designed to fit the architecture.

To describe these non-von Neumann or parallel architectures, a generally accepted taxonomy is that of Flynn (1966). The classification is based on the notion of two streams of information flow to a processor: instructions and data. These two streams can be either single or multiple, giving four classes of machines:

1. Single instruction single data (SISD)
2. Single instruction multiple data (SIMD)
3. Multiple instruction single data (MISD)
4. Multiple instruction multiple data (MIMD)

Table 5.4 shows the four primary classes and some of the architectures that fit in those classes. Most of these architectures will be briefly discussed.

5.4.2.1 Single Instruction Single Data

The SISD architectures encompass standard serial von Neumann architecture computers. In a sense, the SISD category is the base metric for Flynn’s taxonomy.

5.4.2.2 Single Instruction Multiple Data

The SIMD computers are essentially array processors. This type of parallel computer architecture has n -processors, each executing the same instruction, but on different data streams. Often each element in the array can only communicate with its nearest neighbor. Computer architectures that are usually classified as SIMD are the systolic

TABLE 5.4
Flynn’s Classification Scheme for Parallel Computer Architectures

	Single Data Stream	Multiple Data Stream
Single Instruction Stream	von Neumann processors RISC	Systolic processors Wave-front processors
Multiple Instruction Stream	Pipelined architectures VLIW processors	Data flow processors Transputers Grid computers Multiprocessors

and wave-front array computers. In both types of processor, each processing element executes the same (and only) instruction, but on different data. Hence these architectures are SIMD.

SIMD machines are widely used for such imaging computation as matrix arithmetic and convolution.

5.4.2.3 Multiple Instruction Single Data

The MISD computer architecture lends itself naturally to those computations requiring an input to be subjected to several operations, each receiving the input in its original form. These applications include classification problems and digital signal processing. MISD architectures include pipelined and very long instruction word architectures (VLIW).

In pipelined architectures, more than one instruction can be processed simultaneously (one for each level of pipeline). Similarly, VLIW computers tend to be implemented with microinstructions that have very long bit-lengths (and hence more capability). Thus, rather than breaking down macroinstructions into numerous microinstructions, several (nonconflicting) macroinstructions can be combined into several microinstructions.

5.4.2.4 Multiple Instruction Multiple Data

MIMD computers involve large numbers of processors capable of executing more than one instruction on more than one datum at any instant. Except for networks of distributed multiprocessors working on the same problem (grid computing), these are “exotic” architectures. MIMD computers include data flow computers, grid computers, networks of heterogeneous processors, and transputers.

5.4.3 INTERRUPT HANDLING

Hardware interrupts generated externally, such as by the camera in the visual inspection system (VIS), indicate events to the CPU. Interrupts can be caused by

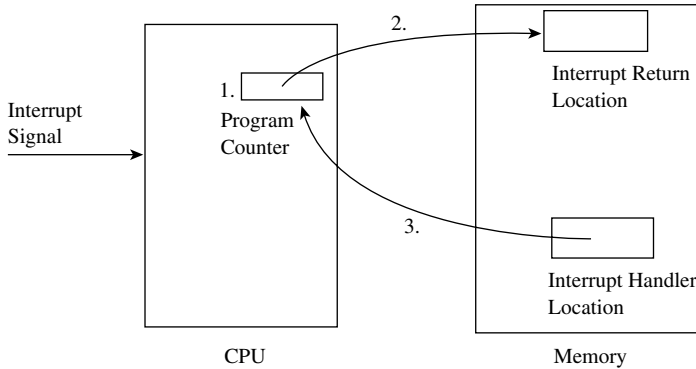


FIGURE 5.19 The interrupt handling process in a single interrupt system.

internal or external events. Internal events, sometimes called traps, include divide-by-zero errors, overflow conditions, and the interrupt to signal the DSP.

Upon receipt of the interrupt signal, the processor completes the instruction that is currently being executed. Next, the contents of the program counter are saved to a designated memory location, called the interrupt return location. The contents of a memory location, called the interrupt handler location, are loaded into the program counter. Execution then proceeds with the special code stored at this location, called the interrupt handler. This process is outlined in Figure 5.19.

Nearly every processor is equipped to handle more than one interrupt in a prioritized fashion.

5.4.4 MEMORY

Memory access times have a profound effect on performance and should influence the choice of instruction modes used — both when coding assembly language and through the careful selection of high-order language constructs.

While the software engineer is often involved in the overall hardware or software function allocation, and even in the hardware design, her real concern is in those characteristics that are visible to the developer and that affect performance. The most important performance characteristic of memory is access time, the interval between when a datum is requested from memory and when it arrives in the processor. The access time depends on the memory type and technology, the memory layout, and other factors. Other important design considerations are volatility (data are lost when power is removed), power requirements, density, and cost.

A summary of four important classifications of memory, their characteristics, and some possible applications in imaging systems is given in Table 5.5.

5.4.5 INPUT AND OUTPUT

Input and output (I/O) of data to a computer system are accomplished through one of three different methods: programmed I/O, memory-mapped I/O, or direct memory

TABLE 5.5
Summary of Memory Device Characteristics

Memory Type	Main Characteristics	Suitability	Applications
Static random access memory (SRAM)	Volatile Small cell size Slower than fast SRAM	Systems with a relatively large amount of memory, where memory cost is critical	Main memory; graphics and peripheral subsystems
Dynamic random access memory (DRAM)	Volatile Large cell size Fast or low power	Systems with a relatively small amount of memory, where memory performance or low power are critical	Cache memory (fast SRAM); high-speed networking systems (fast SRAM); handheld devices (low-power SRAM)
Flash (code storage type)	Nonvolatile Large cell size Fast random access Slow block read/write access	Data that must be retained when power is turned off	Code storage for PC built-in operating system (BIOS)
Flash (data storage type)	Nonvolatile Small cell size Slow random access Fast block read/write access	Data that must be retained when power is turned off and is necessary for application software and user data	Data storage in digital cameras

address (DMA). Each method has advantages and disadvantages with respect to performance, cost, and ease of implementation in imaging systems.

In programmed I/O, special instructions in the CPU instruction set are used to transfer data to and from the CPU. An IN instruction will transfer data from a specified I/O device into a specified CPU register. An OUT instruction will output from a register to some I/O device. Normally, the identity of the operative CPU register is embedded in the instruction code. Both the IN and OUT instructions require the efforts of the CPU and thus cost time that could impact real-time performance.

Memory-mapped I/O provides a data transfer mechanism that is convenient because it does not require the use of special CPU I/O instructions, and it has an additional advantage: the CPU and other devices can share memory. In memory-mapped I/O, certain designated locations of memory appear as virtual I/O ports.

In DMA, access to the computer's memory is given to other devices in the system without CPU intervention. That is, information is deposited directly into main memory by the external device. Here, the cooperation of a device called a DMA controller is required. Because CPU participation is not required, data transfer is fast.

Because of its speed, DMA is the best method for input and output for moving large blocks of data, for example, from a frame grabber to main memory. In this

case, it is helpful to have the device also issue an interrupt when a frame of data has been transferred in order to signal the main program to process it.

5.5 FAULT-TOLERANT DESIGN

Fault tolerance is the tendency to function in the presence of hardware or software failures. In embedded systems, fault tolerance includes design choices that transform hard real-time deadlines into soft ones. These are often encountered in interrupt-driven systems, which can provide for detecting and reacting to a missed deadline.

Fault tolerance designed to increase reliability in embedded systems can be classified as either spatial or temporal. Spatial fault tolerance includes methods involving redundant hardware or software, whereas temporal fault tolerance involves techniques that allow for tolerating missed deadlines. Of the two, temporal fault tolerance is the more difficult to achieve because it requires careful algorithms design. We discuss variations of both techniques in the next several sections.

5.5.1 SPATIAL FAULT TOLERANCE

The reliability of most hardware can be increased by using some form of spatial fault tolerance with redundant hardware. In one common scheme, two or more pairs of redundant hardware devices provide inputs to the system. Each device compares its output to that of its companion. If the results are unequal, the pair declares itself in error and the outputs are ignored. An alternative is to use a third device to determine which of the other two is correct. In either case, the penalty is increased cost, space, and power requirements.

Voting schemes can also be used in software to increase algorithm robustness. Often information is processed from more than one course and reduced to some sort of best estimate of the actual value. For example, an aircraft's position can be determined via information from satellite positioning systems, inertial navigation data, and ground information. A composite of these readings is made using either simple averaging or a Kalman filter.

5.5.2 USING A KALMAN FILTER IN THE CASE STUDY SYSTEM

For example, in our VIS the camera is known to be subject to noise. This discussion, excerpted from Laplante and Neill (2003c), is used to illustrate an object-oriented fault-tolerant design approach for an important component of the VIS: image capture.

The Kalman filter is used to estimate the state variables of a multivariable feedback control system subject to stochastic disturbances caused by noisy measurements of input variables. The Kalman filtering algorithm works by combining the information regarding the system dynamics with probabilistic information regarding the noise. The filter is very powerful in that it supports estimations of past, present, and even future states and, in particular, can do so even when the precise nature of the noise is unknown.

The Kalman filter estimates a process using a form of feedback control — the filter estimates the process state at some time and then obtains feedback in the form

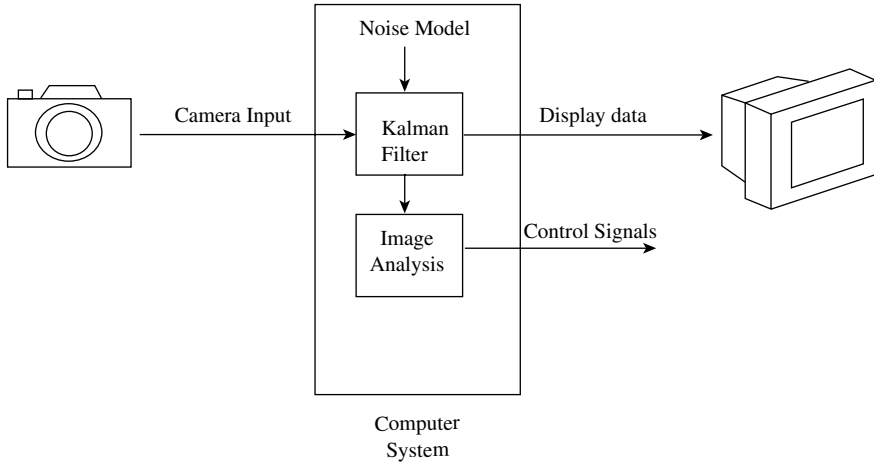


FIGURE 5.20 Kalman filter for image processing. (From Laplante, P. and Neill, C., A Class of Kalman Filters for Real-Time Image Processing, paper presented at Proceedings of the Real-Time Imaging Conference, SPIE, Santa Clara, CA, January 2003, pp. 22–29.)

of noisy measurements (Figure 5.20). There are two kinds of equations for the Kalman filter: time update equations and measurement update equations. The time update equations project forward in time the current state and error covariance estimates to obtain the *a priori* estimates for the next time step. The measurement update equations are responsible for the feedback in that they incorporate a new measurement into the *a priori* estimate to obtain an improved estimate.

The basic problem to be considered in imaging applications is dealing with the degradation of image sequences due to noise introduced during the image acquisition process.

The image is captured by one or more cameras and processed for display or for further algorithmic analysis and processing. The objective is to construct a filter that will reduce the noise and, in particular, remain insensitive to a sudden spike error that can easily fool other types of filters, such as mean square error.

The representation of an image in a Kalman filter can be either pixel-wise or block-wise. The advantage of the latter approach is that interpixel statistical characteristics can be taken into account, however, at significant computational cost. The former approach offers the advantage that it is faster, and we follow that construction.

Let x_k represent a particular pixel found in frame k (taken at time k) of the image (see Figure 5.21). The ideal noise-free pixel value is given by s_k , which is assumed to be a first-order autoregressive (AR) model. This is a model that is generally used to represent the behavior of pixels in video signals. The process model is

$$s_{k+1} = as_k + n_k \quad (5.1)$$

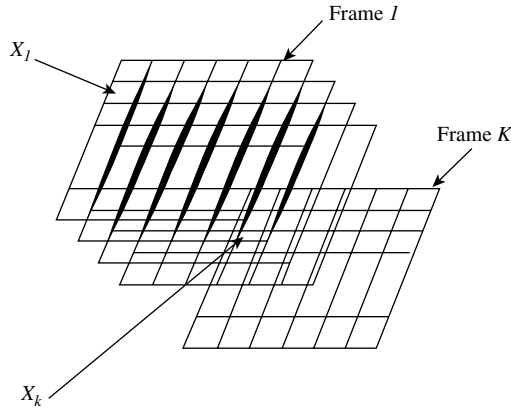


FIGURE 5.21 A series of images captured from a camera. Each pixel represents a time series in frame k . (From Laplante, P. and Neill, C., A Class of Kalman Filters for Real-Time Image Processing, paper presented at Proceedings of the Real-Time Imaging Conference, SPIE, Santa Clara, CA, January 2003, pp. 22–29.)

where a is a constant that depends on the signal statistics and n_k is the process noise, which is assumed to be white Gaussian with a zero mean and a variance of σ_n^2 . While it might be desirable to assume that the noise is Poisson distributed, making the assumption that the noise is Gaussian simplifies the design. Ultimately, however, it will be desirable to construct the class of filters in such a way as to be extensible to different noise sources.

The measured signal is given by

$$x_k = s_k + v_k \quad (5.2)$$

where v_k is the independent additive zero mean Gaussian white noise with a variance of σ_v^2 .

It is implicitly assumed that the noise and signal are stationary random processes that are fully determined by their second-order statistics. This is an approximation that can be safely used.

The Kalman filter output is represented by y_k , which is the estimate of the signal at time k . The variance of the estimation error variance is defined by

$$\sigma_k^2 = E[(y_k - s_k)^2] \quad (5.3)$$

The Kalman filter gain is K_k . The algorithm is as follows: let $y_{-1} = 0$, $\sigma_{-1}^2 = \sigma_v^2$, and $k = 0$. Then the following iterative process, given in C++-like pseudo-code, is as follows:

```

for ( $i = 0$ ;  $i < k$ ;  $i++$ )
{

$$K_i = \frac{a^2 \sigma_{i-1}^2 + \sigma_n^2}{a^2 \sigma_{i-1}^2 + \sigma_n^2 + \sigma_v^2};$$


$$y_i = K_i x_i + a[1 - K_i] y_{i-1};$$


$$\sigma_i^2 = a^2 [1 - K_i] \sigma_{i-1}^2 + \sigma_n^2;$$

 $i++;$ 
}

```

This algorithm is applied to each pixel and each time instant i to provide the filtered image output for display or image analysis purposes.

An OOD of a Kalman filter is presented in Figure 5.22. The design is annotated to indicate the GoF patterns used. The motivation for the design was to produce a filter that could be extended for multiple applications, supporting a number of variant algorithms and noise models. The filter must accept n -inputs, and the configuration chosen was based upon the “pipes and filters” architectural style, supporting both pull and push modes.

The central class in the model is the KalmanMediator; this is both a concrete factory responsible for the creation of instances of the KalmanFilter, InputStream, and OutputStream classes, and a mediator class that coordinates communication between those classes so that they can remain uncoupled, and therefore immune to changes in one another. It is this mediation role that allows for both push and pull mode operations; without a mediator, every instance of the KalmanFilter, InputStream, and OutputStream classes would have to know the mode that the other instances were using.

The design also makes use of interface and implementation independence in the form of the bridge and strategy GoF patterns. The KalmanFilter contains a NoiseModel and a FilterImp instance, each of which are abstract superclasses defined concretely for variant types of noise and filter algorithms, respectively. These instances are therefore interchangeable, allowing for a number of different applications without modification to the existing software system (the OCP). In addition, other subclasses of NoiseModel and FilterImp can be defined for future uses.

5.5.3 CHECKPOINTS

Another way to increase fault tolerance is to use checkpoints. In this scheme, intermediate results are written to memory at fixed locations in code for diagnostic purposes (Figure 5.23). These locations, called checkpoints, can be used during system operation and system verification. If the checkpoints are used only during testing, then this code is known as a test probe. Test probes can introduce subtle timing errors, which are discussed later.

5.5.4 RECOVERY BLOCKS

Fault tolerance can be further increased by using checkpoints in conjunction with predetermined reset points in software. These reset points mark recovery blocks in

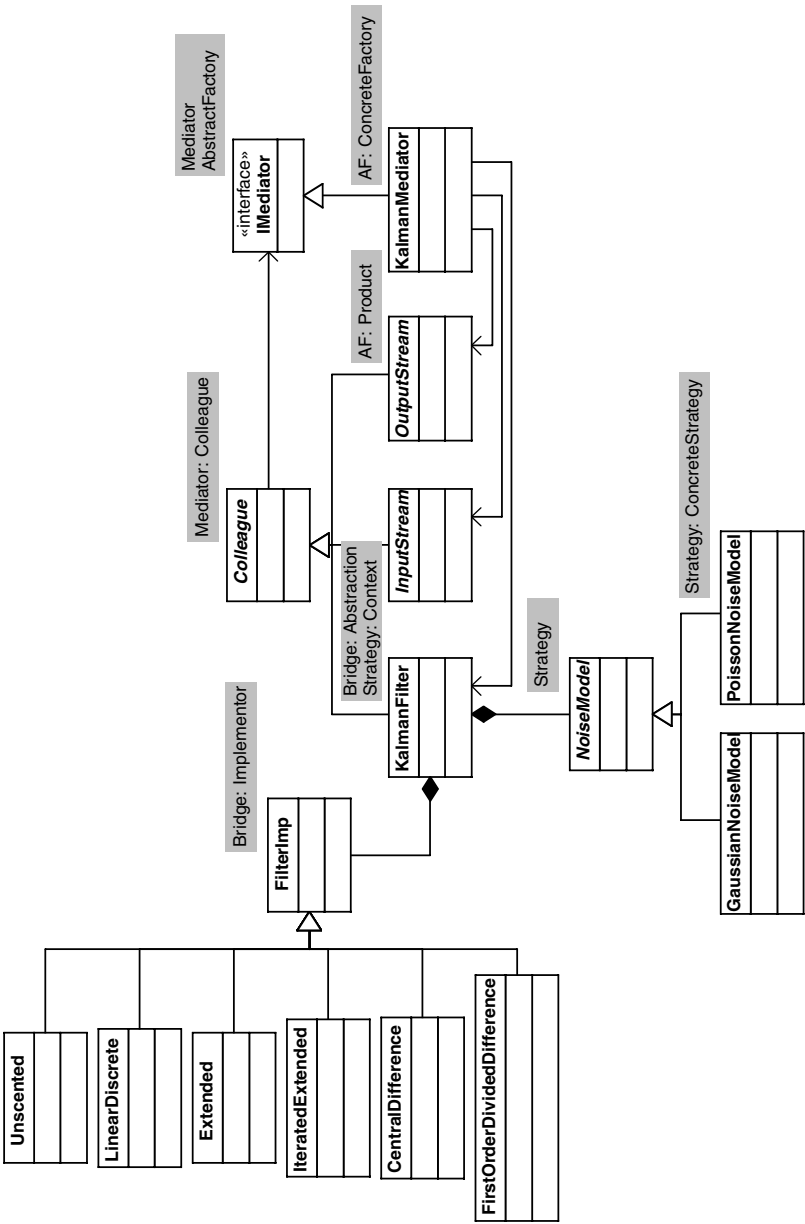


FIGURE 5.22 A class design for a Kalman filter for image processing. (From Laplante, P. and Neill, C., A Class of Kalman Filters for Real-Time Image Processing, paper presented at Proceedings of the Real-Time Imaging Conference, SPIE, Santa Clara, CA, January 2003, pp. 22–29.)

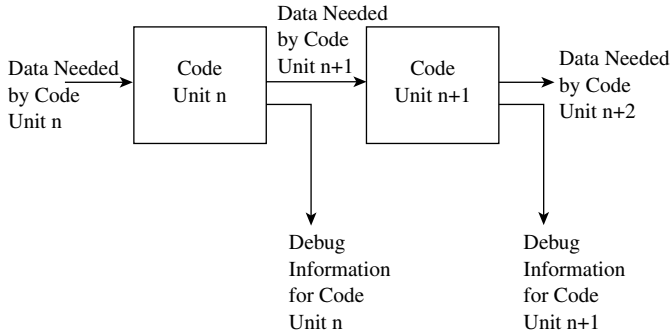


FIGURE 5.23 Checkpoint implementation.

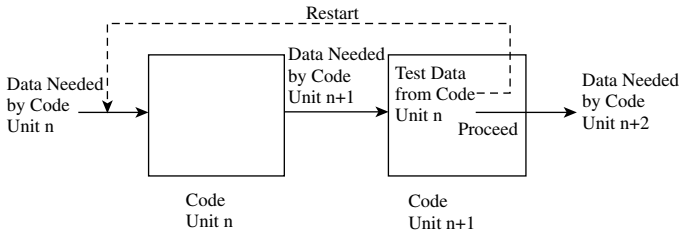


FIGURE 5.24 Recovery block implementation.

the software. At the end of each recovery block, the checkpoints are tested for reasonableness. If the results are not reasonable, then processing resumes at the beginning of that recovery block or some previous one (see Figure 5.24).

The point, of course, is that some hardware device (or another process that is independent of the one in question) has provided faulty inputs to the block. By repeating the processing in the block, with presumably valid data, the error will not be repeated.

In the process block model, each recovery block represents a redundant parallel process to the block being tested. Unfortunately, although this strategy increases system reliability, it can have a severe impact on performance because of the overhead added by the checkpoint and repetition of the processing in a block.

5.5.5 SOFTWARE BLACK BOXES

The software black box is related to checkpoints and is used in certain mission-critical systems to recover data to prevent future disasters. The objective of a software black box is to recreate the sequence of events that led to the software failure for the purpose of identifying the faulty code. The software black box recorder is essentially a checkpoint that stores behavioral data during program execution, while attempting to minimize perturbation in recording this information.

The execution of program functionalities results in the sequence of module transitions where the system can be described as modules and their interaction. When

software is running, it passes from one module to the next, passing the control of execution to it. Passing from one module to the next is considered transition. Call graphs can be developed from these transitions using an $N \times N$ matrix, where N represents the number of modules in a system.

When each module is called, each transition is recorded by incrementing that element in a transition frequency matrix. From this, a transition probability matrix can be derived that records the likeliness that a transition will occur. The transition frequency and transition probability matrices indicate the number of observed transitions and the probability that the sequence is missing in this data, respectively.

Recovery begins after the system has failed and the software black box has been recovered. The software black box decoder generates possible functional scenarios, quantifies the normality of data, and allows further analysis. The data used to generate functional scenarios are from the recorder and the mapping between the modules and functionalities. The generation process attempts to map the modules in the transition or modular sequence to functionalities, which allows for the isolation of the likely cause of the failure.

5.5.6 N-VERSION PROGRAMMING

In any system, a state can be entered where the system is rendered ineffective or locks up. This is usually due to some untested flow of control in the software for which there is no escape.

In order to reduce the likelihood of this sort of catastrophic error, redundant processors are added to the system. These processors are coded to the same specifications but by different programming teams. It is therefore highly unlikely that more than one of the systems can lock up under the same circumstances. Since each of the systems usually resets a watchdog timer, it quickly becomes obvious when one of them is locked up, because it fails to reset its timer. The other processors in the system can then ignore this processor, and the overall system continues to function. This technique is called n-version programming.

The redundant processors can use a voting scheme to decide on outputs, or, more likely, there are two processors: master and slave. The master processor is on-line and produces the actual outputs to the system under control, whereas the slave processor shadows the master off-line. If the slave detects that the master has become hung up, it then goes on-line.

5.5.6.1 Built-In Test Software

Built-in test software can enhance fault tolerance by providing ongoing diagnostics of the underlying hardware for processing by the software. Built-in test is especially important in embedded systems. For example, if an I/O channel is functioning incorrectly as determined by its on-board circuitry, the software may be able to shut off the channel and redirect the I/O.

Although built-in testing is an important part of embedded systems, it adds significantly to the worst-case time-loading analysis.

5.5.6.2 CPU Testing

It is probably more important that the health of the CPU be checked than any other component of the system. A set of carefully constructed tests can be performed to test the efficacy of its instruction set in all addressing modes. Such a test suite will be time-consuming and thus should be relegated to background processing. Interrupts should also be disabled during each subtest to protect the data being used.

5.5.6.3 Memory Testing

All types of memory, including nonvolatile memory, can be corrupted via electrostatic discharge, power surging, vibration, or other means. This damage can manifest itself either as a permutation of data stored in memory cells or as permanent damage to the cell itself. Damage to the contents of memory is called soft error, whereas damage to the cell itself is called hard error. All memory should be tested both at initialization and during normal processing, if possible.

The contents of ROM are often checked by comparing a known checksum. The known checksum, which is usually a simple binary addition of all program-code memory locations, is computed at link time and stored in a specific location in ROM. The new checksum can be recomputed in a slow cycle or background processing, and compared against the original checksum. Any deviation can be reported as a memory error. Checksums are not a very desirable form of error checking because errors to an even number of locations can result in error cancellation. For example, an error to bit 12 of two different memory locations may cancel out in the overall checksum, resulting in no error being detected. In addition, although an error may be reported, the location of the error in memory is unknown.

A reliable method for checking ROM uses a cyclic redundancy code (CRC). The CRC treats the contents of memory as a stream of bits and each of these bits as the binary coefficient long binary polynomial. A second binary polynomial of much lower order (for example, 16 is the usual standard), called the generator polynomial, is divided (modulo-2) into the message, producing a quotient and a remainder. Before dividing, the message polynomial is appended with a 0 bit for every term. In addition to checking memory, the CRC can be employed to perform nonvisual validation of screens by comparing a CRC of the actual output with the CRC of the desired output.

Because of the dynamic nature of RAM, checksums and CRCs are not viable. One way of protecting against errors to memory is to equip it with extra bits used to implement a Hamming code. Depending on the number of extra bits, known as the syndrome, errors to one or more bits can be detected and corrected. Such coding schemes can be used to protect ROM as well.

This device significantly reduces the number of soft errors, which will be removed upon rewriting to the cell, and hard errors, which are caused by stuck bits or permanent physical damage to the memory. The disadvantages of error detection and correction are that as additional memory is needed for the scheme (6 bits for every 16 bits — a 37% increase), additional power is required. An access time

penalty of about 50 nsec per access is incurred if an error correction is made. Finally, multiple bit errors cannot be corrected.

In the absence of error detecting and correcting hardware, basic techniques can be used to test the integrity of RAM memory. These tests are usually run upon initialization, but they can also be implemented in slow cycles if interrupts are appropriately disabled.

For example, suppose a computer system has 8-bit data and address buses to write to 8-bit memory locations. We wish to exercise the address and data buses as well as the memory cells. This is accomplished by writing and then reading back certain bit patterns to every memory location. Traditionally, the following hexadecimal bit patterns are used: AA, 00, 55, FF. The bit patterns are selected so that any cross talk between wires can be detected. Bus wires are not always laid out consecutively, however, so that other cross-talk situations can arise. For instance, the above bit patterns do not check for coupling between odd-numbered wires. The following test set does: AA, 00, 55, FF, 0F, 33. This test set, however, does not isolate the problem to the offending wire (bit). For complete coverage of 8 bits, 28 ($7 + 6 + 5 + 4 + 3 + 2 + 1$) bits are needed in combinations of 2 bits at a time. Since there are 8-bit words, four of these combinations can be tested per test. Thus, the following 8-bit patterns are needed: AA, 00, 55, FF, 0F, 33, CC.

In general, for n -bit data and address buses writing to n -bit memory, where n is a power of 2, a total of $n(n-1)/2$ patterns of 2 are needed, which can be implemented in $n-1$ patterns of n bits each.

5.5.6.4 Other Devices

Devices such as analog to digital (A/D) converters, digital to analog (D/A) converters, bus multiplexers, I/O cards, frame grabbers, cameras, and the like, need to be tested continually, or their self-testing needs to be monitored. Many of these devices have built-in watchdog timer circuitry to indicate that the device is still on-line. The software can check for watchdog timer overflows and either reset the device or indicate failure.

In addition, the test software can rely on the individual built-in tests of the devices in the system. Typically, these devices will send a status word via DMA to indicate their health. The software should check this status word and indicate failures as required.

5.6 EXERCISES

- 5.1 Why is it that there is no one, universally accepted strategy for software design modeling?
- 5.2 How would you handle the situation in which the SRS contains numerous, if not excess, design specifications?
- 5.3 Who should write the design specification?
- 5.4 What are the differences between object-oriented modeling and using DFDs?

- 5.5 Why is it that the code, even though it is a model of behavior, is insufficient in serving as either a software requirements document or a software design document?
- 5.6 What are the pitfalls of n-version programming?
- 5.7 Why is it important that the code be traceable to the software design specification and, in turn, to the SRS? What happens, or should happen, if it is not?
- 5.8 Redraw the visual inspection system context diagram in Figure 4.15 to take into account the calibration and diagnostic modes.

6 The Software Production Process

The art of progress is to preserve order amid change and to preserve change amid order.

Alfred North Whitehead

6.1 PROGRAMMING LANGUAGES

A programming language represents the nexus of design and structure. Hence, because the actual “build” of software depends on tools to compile, generate, link, and create binary objects, “coding” should take relatively little time if the design is solid. Nevertheless, coding (or programming) is more craft-like than mass production, and as with any craft, the best practitioners are known for the quality of their tools and their skill with them.

The main tool in the software production process is the programming language. Imaging systems have been built with a wide range of programming languages, including various dialects of:

- C
- C++
- Java
- Fortran
- Ada
- Assembly language
- Visual BASIC
- BASIC

Each programming language offers its own strengths and weaknesses with respect to imaging systems.

There are several programming language features that stand out in procedural languages that are desirable for use in embedded imaging systems, particularly:

- Versatile parameter passing mechanisms
- Dynamic memory allocation facilities
- Strong typing
- Abstract data typing
- Exception handling
- Modularity

These language features help promote the desirable properties of software and best engineering practices.

6.1.1 PARAMETER PASSING TECHNIQUES

There are several methods of parameter passing, including the use of parameter lists and global variables. While each of these techniques has preferred uses, each has a different performance impact.

6.1.2 CALL-BY-VALUE AND CALL-BY-REFERENCE

The two base parameter passing methods are call-by-value and call-by-reference.* In call-by-value parameter passing, the value of the actual parameter in the subroutine or function call is copied into the procedure's formal parameter. Since the procedure manipulates the formal parameter, the actual parameter is not altered. This technique is useful when either a test is being performed or the output is a function of the input parameters. For example, in an edge detection algorithm, an image is passed to the procedure and some description of the location of the edges is returned, but the image itself need not be changed. When parameters are passed using call-by-value, they are copied onto a run-time stack, at considerable execution time cost. For example, large image arrays must be passed pixel by pixel.

In call-by-reference, or call-by-address, the address of the parameter is passed by the calling routine to the called procedure so that it can be altered there. Execution of a procedure using call-by-reference can take longer than one using call-by-value, since in call-by-reference, indirect mode instructions are needed for any calculations involving the variables passed. However, in the case of passing images between procedures, it is more desirable to use call-by-reference, since passing a pointer to an image is more efficient than passing the image pixel-wise.

Parameter lists are likely to promote modular design because the interfaces between the modules are clearly defined. Clearly defined interfaces can reduce the potential of untraceable corruption of data by procedures using global access. However, both call-by-value and call-by-reference parameter passing techniques can impact performance when the lists are long, since interrupts are frequently disabled during parameter passing to preserve the time correlation of the data passed. Moreover, call-by-reference can introduce subtle function side effects that depend on the compiler.

6.1.3 GLOBAL VARIABLES

Global variables are variables that are within the scope of all modules of the software system. This usually means that references to these variables can be made in direct mode, and thus are faster than references to variables passed via parameter lists. For

* There are three other historical parameter passing mechanisms: call-by-constant, which was removed almost immediately from the Pascal language; call-by-value-result, which is used in Ada; and call-by-name, which was a mechanism peculiar to Algol-60.

example, in many image processing applications, global arrays are defined to represent images; hence, costly parameter passing can be avoided.

Global variables introduce no timing problems but are dangerous because reference to them may be made by an unauthorized code, thus introducing subtle bugs. For this and other reasons, unwarranted use of global variables is to be avoided. Global parameter passing is only recommended when timing warrants and its use must be clearly documented.

The decision to use one method of parameter passing or the other represents a trade-off between good software engineering practice and performance needs. For example, often timing constraints force the use of global parameter passing in instances when parameter lists would have been preferred for clarity and maintainability.

6.1.4 RECURSION

Most programming languages provide recursion in that a procedure can call itself. Recursion is widely used in image processing, for example, in implementing self-referential algorithms and data structures such as the Hadamard matrices, fractal compression, segmentation via divide-and-conquer methods, quad trees, and thinning.

While recursion is elegant and is often necessary in imaging algorithms, its adverse impact on performance must be considered. Procedure calls require the allocation of storage on one or more stacks for the passing of parameters and for storage of local variables. The execution time needed for the allocation and deallocation, and for the storage of those parameters and local variables, can be costly. In addition, recursion necessitates the use of a large number of expensive memory direct and register indirect instructions. Finally, the use of recursion often makes it impossible to determine the size of run-time memory requirements. Thus, iterative techniques such as while and for loops must be used if performance prediction is crucial or in those languages that do not support recursion.

6.1.5 DYNAMIC MEMORY ALLOCATION

The ability to dynamically allocate memory is important in the construction and maintenance of many data structures needed in an imaging system. While dynamic allocation can be time-consuming, it is usually necessary, especially when creating intermediate images needed in most algorithms. Linked lists, trees, heaps, and other dynamic data structures can benefit from the clarity and economy introduced by dynamic allocation. Furthermore, in cases where just a pointer is used to pass a data structure, the overhead for dynamic allocation can be quite reasonable. When writing imaging code, however, care should be taken to ensure that the compiler will pass pointers to large data structures and not the data structure itself.

Languages that do not allow dynamic allocation of memory require data structures of fixed size. While this may be faster, flexibility is sacrificed and memory requirements must be predetermined.

Languages such as C, Pascal, Ada, and Modula-2 have dynamic allocation facilities, while most versions of Fortran, for example, do not.

6.1.6 TYPING

Typed languages require that each variable and constant be of a specific type (e.g., pixel, Boolean, and integer) and that each be declared as such before use. Languages that provide specialized types for imaging applications are rare. Generally, high-level languages provide integer and floating-point types, along with Boolean, character, and string types. In some cases, abstract data types are supported. These allow programmers to define their own type (such as pixel) along with the associated operations. Use of abstract data types, however, will incur an execution time penalty, as complicated internal representations are often needed to support the abstraction. Strongly typed languages prohibit the mixing of different types in operations and assignments, and thus force the programmer to be precise about the way data are to be handled. Precise typing can prevent corruption of data through unwanted or unnecessary type conversion. Hence, strongly typed languages are desirable for imaging.

Some languages are typed, but do not prohibit mixing of types in arithmetic operations. Since these languages generally perform mixed calculations using the type that has the highest storage complexity, they must promote all variables to that type. For example, in C, the following code fragment illustrates automatic promotion and demotion of variable types:

```
int x,y;  
  
float k,l,m;  
  
.  
.  
  
j = x*k + m;
```

Here the variable *x* will be promoted to a float (real) type, and then multiplication and addition will take place in floating point. Afterward, the result will be truncated and stored in *j*. The performance impact is that hidden promotion and more time-consuming arithmetic instructions can be generated, with no additional accuracy. In addition, accuracy can be lost due to the truncation, or worse, an integer overflow can occur if the floating-point value is larger than the allowable integer value. Programs written in languages that are weakly typed need to be scrutinized for such effects. Some C compilers will catch type mismatches in function parameters. This can prevent unwanted type conversions.

6.1.7 EXCEPTION HANDLING

Certain languages provide facilities for dealing with errors or other anomalous conditions that arise during program execution. These conditions include the obvious, such as floating-point overflow, square root of a negative, divide by zero, and image-related ones such as boundary violation, wraparound, and pixel overflow. The ability to define and handle exceptional conditions in the high-level language aids in the construction of interrupt handlers and other code used for real-time event processing. Moreover, poor handling of exceptions can degrade performance. For

example, floating-point overflow errors can propagate bad data through an algorithm and instigate time-consuming error recovery routines.

Of all the languages widely used in imaging applications, Ada has the most explicit exception handling facility, although Java also has excellent exception handling using the “try, throw, catch, finally” approach used by many mainstream object-oriented languages. ANSI-C also provides some exception handling capability through the use of signals.

Finally, exception handling can often be implemented in languages such as C, Pascal, and Modula-2 as a user-definable library when permitted by the compiler.

6.1.8 MODULARITY

Procedural languages that are amenable to the principle of information hiding tend to make it easy to construct subprograms. While C and Fortran both have mechanisms for this (procedures and subroutines), other languages such as Ada (which can be considered either procedural or object oriented) tend to foster more modular design because of the requirement to have clearly defined inputs and outputs in the module parameter lists.

In Ada the notion of a package embodies the concept of Parnas information hiding exquisitely. The Ada package consists of a specification and declarations that include its public or visible interface and its invisible or private parts. In addition, the package body, which has further externally invisible components, contains the working code of the package. Packages are separately compilable entities, which further enhances their application as black boxes. In Fortran there is the notion of a SUBROUTINE and separate compilation of source files. These language features can be used to achieve modularity and design abstract data types. The C language also provides for separately compiled modules and other features that promote a rigorous top-down design approach that should lead to a good modular design.

While modular software is desirable, there is a price to pay in the overhead associated with procedure calls and parameter passing. This adverse effect should be considered when sizing modules.

Object-oriented languages provide a natural environment for information hiding. For example, in image processing systems, it might be useful to define a class of type pixel, with attributes describing its position, color, and brightness; and operations that can be applied to a pixel, such as add, activate, deactivate, and so on. It might also be desirable to define objects of type image as a collection of pixels with other attributes of width, height, and so on. In some cases, expression of system functionality is easier to do in an object-oriented manner.

Object-oriented techniques can increase programmer efficiency, reliability, and the potential for reuse. Generally, there is an execution time penalty in object-oriented languages. This is due in part to late binding (resolution of memory locations at run time rather than at compile time) necessitated by function polymorphism and inheritance. Late binding can often present a significant delay. Another problem results from the collection garbage generated by these types of languages. One possible way to reduce these penalties is not to define too many classes and to define only classes that contain coarse detail and high-level functionality.

6.1.9 BRIEF SURVEY OF LANGUAGES

For purposes of illustrating the aforementioned language properties, it is helpful to review some of the more widely used languages in imaging systems. The languages are presented in alphabetical order, and their pros and cons with respect to their use in imaging applications are discussed.

It is noteworthy that functional languages, such as LISP and ML, have been omitted from the discussions. This is not because they are useless in the context of imaging applications, but simply because they are rare in their use in this setting. The discussion also omits object-oriented scripting languages, which have become popular for writing tools, and test harnesses such as Python and Ruby, simply because they are not appropriate for embedded targets.

6.1.9.1 Ada 95

Ada was originally intended to be the mandatory language for all U.S. Department of Defense projects. The first version, which became standardized in 1983, had significant problems. The programming language community had long been aware of the problems with the first release of the Ada standard and, practically since the first delivery of an Ada 83 compiler, had sought to resolve them, which resulted in a new version. The new language, now called Ada 95, was the first internationally standardized object-oriented programming language. However, Ada's original intent has been consistently undermined by numerous exceptions that were granted, and it seems inevitable that Ada is not destined to fulfill its original intent.

Ada was designed specifically for embedded real-time systems, but systems builders have typically found the language to be too bulky and inefficient. Moreover, significant problems were found when trying to implement multitasking using the limited tools supplied by the language, such as the roundly criticized rendezvous mechanism.

Three pragmas were introduced in Ada 95 to resolve some of the uncertainty in scheduling, resource contention, and synchronization:

1. One controls how tasks are dispatched.
2. Another controls the interaction between task scheduling.
3. The last controls the queuing policy of task or resource entry queues.
First-in-first-out and priority queuing policies are available.

Other expansions to the language were intended to make Ada 95 an object-oriented language. These include:

- Tagged types
- Packages
- Protected units

Proper use of these constructs allows for the construction of objects that exhibit the three characteristics of object-oriented languages (abstract data typing, inheritance, and polymorphism).

However, as mentioned, Ada has never lived up to its promise of universality. Nevertheless, even though the number of available Ada developers continues to dwindle, the language is staging somewhat of a mini-comeback, particularly because of the availability of open-source versions of Ada for Linux (Linux is an open-source derivative of the Unix operating system).

6.1.9.2 Assembly Language

Though lacking most of the features discussed for the high-level languages, assembly language does have certain advantages in that it provides more direct control of the computer hardware. Unfortunately, because of its unstructured and limited abstraction properties, and because it varies widely from machine to machine, coding in assembly language is usually difficult to learn, tedious, and error prone. The resulting code is also unportable.

Until just a few years ago, most assembly language programmers could generate code that was more efficient than the code generated by a compiler. But with improvements in optimizing compilers, only the very best assembly language programmers can generate code that is faster and more compact than those of the best compilers. Thus, the need to write assembly code exists only in cases where the compiler does not support certain macroinstructions or when the timing constraints are so tight that hand-tuning is needed to produce optimal code. In any case, a system will likely find that 99% of the code will be written in the high-order language, while the rest is written in assembly language.

In cases where complex prologues and epilogues are needed to prepare an assembly language program, often a shell of the program is written in the high-order language and compiled to an assembly file, which is then massaged to obtain the desired effect. Some languages such as Ada and versions of Pascal provide a pragma pseudo-op, which allows for assembly code to be placed in-line with the high-order language code.

Assembly language programming should be limited to use in very tight timing situations or in controlling hardware features that are not supported by the compiler. In general, however, it should be discouraged.

6.1.9.3 C

The C programming language, invented around 1971, is a good language for low-level programming. The reason for this is that it is descended from the language BCPL (whose successor, C's parent, was B), which supported only one type — machine word. Consequently, C supported machine-related objects like characters, bytes, bits, and addresses, which could be handled directly in high-level language. These entities can be manipulated to control interrupt controllers, CPU registers, and other hardware needed by a real-time system. C is also often used as a high-level cross-platform assembly language.

C provides special variable types such as register, volatile, static, and constant, which allows for control of code generation at the high-order language level. For example, declaring a variable as a register type indicates that it will be used frequently. This encourages the compiler to place such a declared variable in a register, which

often results in smaller and faster programs. C supports call-by-value only, but call-by-reference can be implemented by passing the pointer to anything as a value.

Variables declared as type `volatile` are not optimized by the compiler. This is useful in handling memory-mapped input and output (I/O) and other instances where the code should not be optimized.

The C language provides for exception handling through the use of signals, and two other mechanisms, `setjmp` and `longjmp`, are provided to allow a procedure to return quickly from a deep level of nesting, a useful feature in procedures requiring an abort. The `setjmp` procedure call, which is really a macro (but often implemented as a function), saves environment information that can be used by a subsequent `longjmp` library function call. The `longjmp` call restores the program to the state at the time of the last `setjmp` call. Suppose a procedure process is called to perform some processing and error checking. If an error is detected, a `longjmp` is performed that changes the flow of execution directly to the first statement after the `setjmp`.

Overall, however, the C language is good for embedded programming because it provides for structure and flexibility without complex language restrictions.

6.1.9.4 C++

C++ is a hybrid object-oriented programming language that was originally implemented as a macroextension of C. Today, C++ stands by itself as a separately compiled language, although strictly speaking, C++ compilers should accept standard C code.

C++ exhibits all three characteristics of an object-oriented language. It promotes better software engineering practice through encapsulation and better abstraction mechanisms than C.

Significantly, more embedded systems are being constructed in C++ and many practitioners are asking, “Should I implement the system in C or C++?” My answer to them is always “it depends.” Choosing C in lieu of C++ in embedded imaging applications is, roughly speaking, a trade-off between a “lean and mean” C program that will be faster and easier to predict, but harder to maintain, and a C++ program that will be slower and less predictable, but potentially easier to maintain.

C++ still allows for low-level control (without falling back to C features); for example, it can use in-line methods rather than a run-time call. This kind of implementation is not completely abstract or completely low level, but is acceptable in embedded environments.

However, there is some tendency to take existing C code and objectify it by wrapping the procedural code into objects with little regard for the best practices of object orientation. This kind of approach is to be avoided because it has the potential to incorporate all of the disadvantages of C++ and none of the benefits.

6.1.9.5 Fortran

The Fortran* language is the oldest high-order language extant (developed circa 1955). Because in its earlier versions Fortran lacked recursion and dynamic alloca-

* Although Fortran is an acronym for Formula Translator, it is often written as *Fortran* because the word has entered the mainstream in the same way that the acronyms *laser* and *sonar* have.

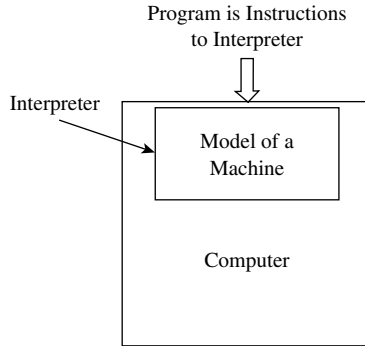


FIGURE 6.1 The Java interpreter as a model of a virtual machine.

tion facilities, embedded systems written in this language typically included a large portion of assembly language code to handle interrupts and scheduling, and communication with external devices was through the use of memory-mapped I/O, direct memory addressing, and I/O instructions. Later versions of the language included such features as reentrant code, but even today, an embedded Fortran system requires some assembly language code to accompany it.

Fortran was developed in an era when efficient code was essential to optimizing performance in small, slow machines. As a result, the language constructs were selected for efficiency, and early Fortran code generators were unusually so.

To its detriment, Fortran is weakly typed, but because of the subroutine construct and the if-then-else construct, it can be used to design highly structured code. Fortran has no built-in exception handling or abstract data types. Fortran is still used to write many imaging applications.

6.1.9.6 Java

Java is an interpreted language; that is, the code compiles into machine-independent code that runs in a managed execution environment. This environment is a virtual machine (Figure 6.1) that executes “object” code instructions as a series of program directives. The advantage of this arrangement is that the Java code can run on any device that implements the virtual machine. This “write once, run anywhere” philosophy has important applications in embedded and portable computing, as well as in Web-based computing.

Java is in some ways not just a language, but an environment through its virtual machine. There are some native-code Java compilers, however, that allow Java to run directly “on the bare metal”; that is, the compilers convert Java directly to assembly code or object code.

Java is an object-oriented language that looks very much like C++. Like C, Java supports call-by-value, but the value is the reference to an object, which is in essence call-by-reference for all objects. Primitives are passed-by-value.

Some features of Java that are different than C++ and are of interest in imaging applications are:

- There are no global functions or constants — everything belongs to a class.
- Arrays and strings have built-in bounds checking.
- All values are initialized and use special defaults if none are given.
- All classes in Java ultimately inherit from the object class.
- Java does not support multiple inheritance (although it can be simulated with the “implements” construct).
- Java does not support the GOTO statement; however, it supports labeled breaks.
- Java does not support automatic type conversions (except where guaranteed safe).
- Types are all references to objects, except the primitive types.

One of the best-known problems with Java involves its garbage collection utility. Garbage is memory that has been allocated but is unusable because of the loss of a pointer to it, for example, through the destruction of an object. The allocated memory must be reclaimed through garbage collection.

Garbage collection algorithms generally have unpredictable performance (although average performance may be known). The loss of determinism results from the unknown amount of garbage, the tagging time of the nondeterministic data structures, and the fact that many incremental garbage collectors require that every memory allocation or deallocation from the heap be willing to service a page-fault trap handler.

6.2 WRITING AND TESTING CODE

A compiler translates a program from high-level source code language into relocatable machine instructions. Usually this process is broken into a series of phases. The overall process is illustrated in Figure 6.2. Once the program has been processed by the linker, it is ready for execution.

First, the high-level language program is translated into a symbolic machine code form called assembly language. Next, a separate program called an assembler is used to translate the symbolic assembly language into relocatable machine code or object code. At some stage in the compilation process, optimization of the code may take place. Finally, the relocatable code output from the compiler is made absolute, and all external references are resolved, by a program called a linker, or linking loader.

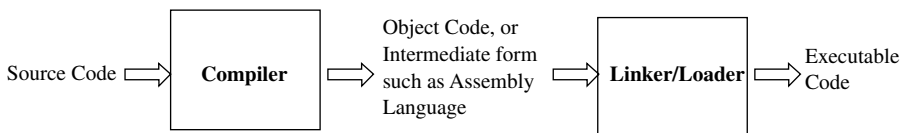


FIGURE 6.2 The compilation and linking processes.

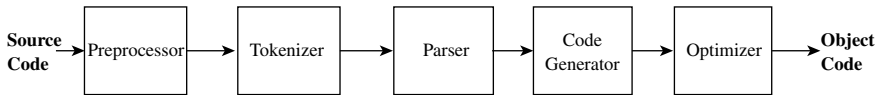


FIGURE 6.3 Phases of compilation provided by a typical compiler, such as Unix/Linux C.

6.2.1 EXAMPLE: THE UNIX/LINUX C COMPILER

To further illustrate the process of compiling, linking, and some of the issues in source code unit testing and debugging, consider the widely used Unix-based C compiler. This compiler is also found in many Linux implementations.

The Unix C compiler **cc** provides a utility that controls the compilation and linking processes. In particular, in Unix System V version 3 (SVR3), the **cc** program provides for the following phases of compilation:

- Preprocessing
- Compilation
- Optimization
- Assembly
- Linking and loading

These phases are illustrated in Figure 6.3 and summarized in Table 6.1.

The preprocessing phase of the compiler, performed by the program **cpp**, takes care of such things as converting symbolic values into actual values and evaluating and expanding code macros. The compilation of the program, that is, the translation of the program from C language to assembly language, is performed by the program **ccom**. Optimization of the code is performed by the program **c2**, and assembly or translation of the assembly language code into machine codes is taken care of by **as**. Finally, the object modules are linked together, all external references (for example, library routines) are resolved, and the program (or an image of it) is loaded into memory by the program **ld**. The executable program is now ready to run.

TABLE 6.1
Phases of Compilation and
Their Associated Program for
the Unix C Compiler

Phase	Program
Preprocessing	cpp
Compilation	ccom
Optimization	c2
Assembly	as
Linking and loading	ld

The fact that many of these phases can be bypassed or run alone is an important feature that will help in program debugging and optimization.

6.2.2 HANDLING COMPILER ERRORS

It is beyond the scope of this text to discuss the many program warnings and errors that the user can encounter in the course of compiling and linking programs. A discussion of this is best left to the reference book of the language in question.

However, one technique that can help is to redirect errors to a file. When a program with syntax errors is compiled, errors may be displayed to the screen too fast to read. These errors can be redirected to a file that can be looked at leisurely. This technique is called redirecting standard error.

6.2.3 SOME DEBUGGING TIPS: UNIT-LEVEL TESTING

Programs can be affected by syntactic or logic errors. Syntactic or syntax errors arise from the failure to satisfy the rules of the language. A valid compiler will always detect syntax errors, although the way that it reports the error can often be misleading.

For example, in a C program a missing `}` may not be detected until many lines after it should have appeared. Some compilers only report “syntax error,” rather than, for example, “missing `}`.”

In logic errors, the code adheres to the rules of the language, but the algorithm that is specified is somehow wrong. Logic errors are more difficult to diagnose because the compiler cannot detect them; however, a few basic rules may help find and eliminate logic errors:

- Document the program carefully. Ideally, each nontrivial line of code should include a comment. In the course of commenting, this may detect or prevent logical errors.
- Where a symbolic debugging is available, use steps, traces, break points, skips, and so on, to isolate the logic error (discussed later).
- In the case of a command line environment (such as Unix/Linux), use print statements to output intermediate results at checkpoints in the code. This may help detect logic errors.
- In case of an error, comment out portions of the code until the program compiles and runs. Add in the commented out code, one feature at a time, checking to see that the program still compiles and runs. When the program either does not compile or runs incorrectly, the last code you added is involved in the logic error.

Finding and eliminating logic errors are more art than science, and the software engineer develops these skills only with time and practice. In many cases, code audits or walk-throughs can be quite helpful; they are discussed later.

6.2.4 EXTENDED SYNTAX AND SEMANTIC CHECKING

While it is impossible to provide automatic logic validation, and the compiler can only check for syntactical correctness, many programming environments provide

tools that are helpful in eliminating logical errors. For example, two tools are associated with Unix and Linux.

One of these is called **lint**. As its name implies, **lint** is a nitpicker that does checking beyond that of an ordinary compiler. For example, C compilers are often not very particular about certain inconsistencies, such as parameter mismatches, declared variables that are not used, and type checking; however, **lint** is particular, often preventing or diagnosing very difficult bugs.

Another Unix tool, the C beautifier, or **cb**, is simply used to transform a sloppy-looking program into a readable one. It does not change the program code. Instead, **cb** just adds plenty of tabs, line feeds, and spaces where needed to make things look nice. This is very helpful in finding badly matched or missing curly braces, erroneous if-then-else and case statements, and incorrectly terminated functions. As with **lint**, **cb** is run by typing “cb” and a file name at the command prompt.

6.2.5 SYMBOLIC DEBUGGING

Source-level debuggers are software programs that provide the ability to step through code at either a macroassembly or high-order language level. They are extremely useful in module-level testing. They are less useful in system-level debugging because the run-time aspect of the system is necessarily disabled or affected.

Debuggers can be obtained as part of compiler support packages or in conjunction with sophisticated logic analyzers. For example, **sdb** is a generic name for the symbolic debugger associated with Unix and Linux. It allows the engineer to single-step through source language code and view the results of each step.

In order to use the symbolic debugger, the source code must be compiled with the appropriate option set. This has the effect of including a special run-time code that interacts with the debugger. Once the code has been compiled for debugging, it can be executed normally.

For example, in the Unix/Linux environment, the program can be started normally from the **sdb** debugger at any point by typing certain commands at the command prompt. However, it is more useful to single-step through the source code. Lines of code are displayed and executed one at a time by using the “s” (for step) command. If the statement is an output statement, it will output to the screen accordingly. If the statement is an input statement, it will await user input. All other statements execute normally.

At any point in the single-stepping process, individual variables can be set or examined. There are many other features of **sdb**, such as break-point setting. In more sophisticated operating environments, a graphical user interface (GUI) is also provided, but essentially, these tools provide the same functionality.

Very often when debugging a new program, the Unix operating system will abort execution and indicate that a core dump has occurred. This is a signal that some fault has occurred. A core dump creates a rather large file named core, which many programs simply remove before proceeding with the debugging. But core contains some valuable debugging information, especially when used in conjunction with **sdb**. For example, core contains the last line of the program that was executed and the contents of the function call stack at the time of the catastrophe. The **sdb** can

be used to single-step up to the point of the core dump to identify its cause. Later on, you may learn to use break points to quickly come up to this line of code.

When removing code during debugging, it is inadvisable to use conditional branching. Conditional branching affects timing and can introduce subtle timing problems. Conditional compilation is more useful in these instances. In conditional compilation, selected code is included only if a compiler directive is set and does not affect timing in the production system.

6.2.6 TEST-FIRST CODING

Test-first coding is a code production approach in which the test cases for the code are written, by the software engineer who will eventually write the code, before the code is written. The advantage of this approach is that it forces the software engineer to think about the code in a very different way that involves focusing on “breaking down” the software. Software engineers who use this technique report that while it is sometimes difficult to change their way of thinking, once the test cases have been written, it is actually easier to write the code, and debugging becomes much easier because the unit-level test cases have already been written.

6.2.7 KNOW THE COMPILER

Understanding the mapping between high-order language input and assembly language output for a particular compiler is essential in generating code that is optimal in either execution time or memory requirements. The easiest and most reliable way to learn about any compiler is to run a series of tests on specific language constructs.

For example, in many C and Pascal compilers the case statement is efficient only if more than three cases are to be compared — otherwise nested if statements should be used. Sometimes the code generated for a case statement can be quite convoluted, for example, a jump through a register, offset by the table value. This can be time-consuming.

It has already been mentioned that procedure calls are costly in terms of passing of parameters via the stack. The software engineer should determine whether the compiler passes the parameters by byte or by word.

Other language constructs that may need to be considered include:

- Use of while loops vs. for loops or do-while loops
- When to “unroll” loops, that is, to replace the looping construct with repetitive code (thus saving the loop overhead as well as providing the compiler with the opportunity to use faster, direct or single indirect mode instructions)
- Comparison of variable types and their uses (for example, when to use short integer in C vs. Boolean, when to use single precision vs. double precision floating point, and so forth)
- Use of in-line expansion of code via macros vs. procedure calls

This by no means is an exhaustive list.

While good compilers should provide optimization of the assembly language code output in order to make the decisions listed above, it is important to discover what that optimization is doing to produce the resultant code. For example, compiler output can be affected by optimization for speed, memory and register usage, jumps, and so on, which can lead to inefficient code, timing problems, or critical regions. Thus, embedded systems engineers must be masters of their compilers. That is, at all times the engineer must know what assembly language code will be output for a given high-order language statement. A full understanding of each compiler can only be accomplished by developing a set of test cases to exercise it. The conclusions suggested by these tests can be included in the set of coding standards to foster improved use of the language and, ultimately, improved system performance.

6.3 CODING STANDARDS

Coding standards are different from language standards. A language standard, for example, ANSI C, embodies the syntactic rules of the language. A program violating those rules will be rejected by the compiler. Conversely, a coding standard is a set of stylistic conventions. Violating the conventions will not lead to compiler rejection. In another sense, compliance with language standards is mandatory, while compliance with coding standards is voluntary.

Adhering to language standards fosters portability across different compilers, and hence hardware environments. Complying with coding standards will not foster portability, but rather, in many cases, readability and maintainability. Some even contend that the use of coding standards can increase reliability. Coding standards may also be used to foster improved performance by encouraging or mandating the use of language constructs that are known to generate more efficient code. Many agile methodologies embrace coding standards, for example, extreme programming.

Coding standards involve standardizing some or all of the following elements of programming language use:

- Standard or boilerplate header format.
 - Frequency, length, and style of comments.
 - Naming of classes, methods, procedures, variable names, data, file names, and so forth.
 - Formatting of program source code, including use of white space and indentation.
 - Size limitations on code units, including maximum and minimum lines of code, number of methods, and so forth.
 - Rules about the choice of language construct to be used; for example, when to use case statements instead of nested if-then-else statements.
- Determination of these rules was discussed in the previous section.

While it is unclear if conforming to these rules fosters improvement in reliability, for example, clearly close adherence can make programs easier to read and understand and likely more reusable and maintainable.

There are many different standards for coding that are language independent, or language specific. Coding standards can be team-wide, company-wide, or user-group specific (for example, the Gnu software group has standards for C and C++), or customers can require conformance to specific standards that they own. Still other standards have become public domain.

One example is the Hungarian notation standard, named in honor of Charles Simonyi, who is credited with first promulgating its use. Hungarian notation is a public domain standard intended to be used with object-oriented languages, particularly C++. The standard uses a complex naming scheme to embed type information about the objects, methods, attributes, and variables in the name. Because the standard essentially provides a set of rules about naming variables, it can be and has been used with other languages such as Ada, Java, and even C.

One problem with standards such as the Hungarian notation is that they can promote very mangled variable names, in that they direct focus on how to name in Hungarian rather than a meaningful name of the variable for its use in code. In other words, the desire to conform to the standard becomes the end, not a particularly meaningful variable name.

Another problem is that the very strength of coding standard can be its own undoing. For example, in Hungarian notation what if the type information embedded in the object name is, in fact, wrong? There is no way for a compiler to check this. There are commercial rules wizards, reminiscent of **lint**, that can be tuned to enforce the coding standards, but they must be programmed to work in conjunction with the compiler.

Finally, adoption of coding standards is not recommended mid-project. It is much easier to start conforming than to be required to change existing code to comply.

The decision to use coding standards is an organizational one that requires significant forethought and debate.

6.4 REVIEWS AND AUDITS

Joint application design (JAD) is a requirements engineering process whereby highly structured group meetings or mini-retreats involving system users, system owners, and analysts occur in a single room for an extended period of time — 4 to 8 hrs per day, anywhere from one day to a couple weeks – to focus on requirements development. JAD-like techniques are becoming increasingly common in systems planning and systems analysis to obtain group consensus on problems, objectives, and requirements.

Similar kinds of immersive retreats can be used for reviews and audits of code and other software artifacts. These reviews and audits can be used for:

- Eliciting requirements and for the software requirements specification
- Design and software design description
- Code
- Tests and test plan
- User's manuals

And there can be multiple reviews for each.

Planning for a review or audit session involves three steps:

1. Selecting participants
2. Preparing the agenda
3. Selecting a location

Reviews and audits may include some or all of the following participants:

- Sponsors (for example, senior management)
- Team leader (facilitator, independent)
- Users and managers who have ownership of requirements and business rules
- Scribe(s)
- Engineering staff

The sponsor, analysts, and managers select a leader. The leader may be in-house or contracted. One or more scribes (recorders), normally selected from the engineering staff, are selected from the development team(s). The analyst and managers must select individuals from the user community; they should be knowledgeable about their business area and able to articulate it.

Before planning a session, the analyst and sponsor must determine the scope of the project and set the high-level requirements and expectations of each session. The session leader must also ensure that the sponsor is willing to commit people, time, and other resources to the effort. The code or documentation must also be sent to all participants well in advance of the meeting so that they have sufficient time to review them, make comments, and prepare to ask questions.

The agenda depends on the topic of the type of review, but it should be constructed to allow sufficient time.

Finally, the physical layout of the room also has much to do with the success of the session. A proposed layout is shown in Figure 6.4.

Some rules for conducting software requirements, design audits, or code walk-throughs follow. The session leader must make every effort to implement these practices.

- Stick to the agenda.
- Stay on schedule (agenda topics are allotted specific time).
- Ensure the scribe is able to take notes.
- Avoid technical jargon (if the review is a requirements review and involves nontechnical personnel).
- Resolve conflicts (try not to defer).
- Encourage group consensus.
- Encourage user and management participation without allowing individuals to dominate the session.
- Keep it impersonal.
- Take notes.
- Take as long as it takes.

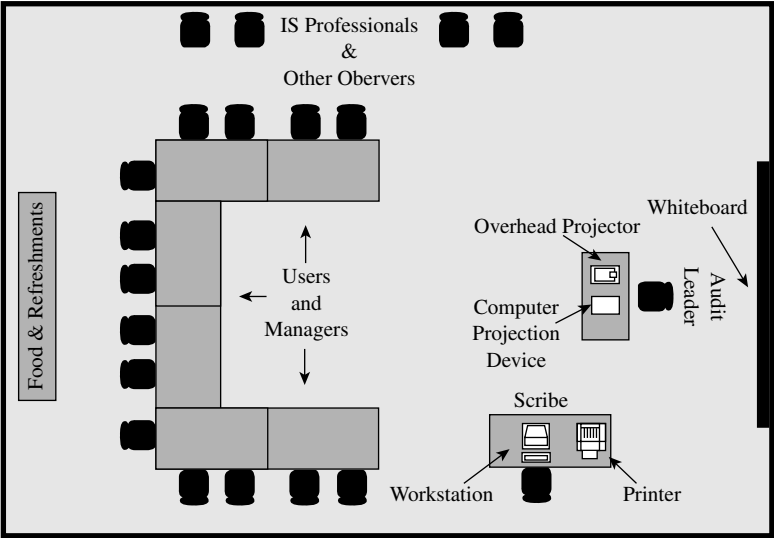


FIGURE 6.4 Typical room layout for a review session, such as a requirements, design, or code review.

The end product of any review session is typically a formal written document providing a summary of the items (specifications, design changes, code changes, and action items) agreed upon during the session. The content and organization of the document obviously depend on the nature and objectives of the session.

6.5 DOCUMENTATION

Traceability includes providing good documentation at important points in the software life cycle. In the waterfall model, documentation would be an expected output at the end of each of the phases. Table 6.2 summarizes these documents.

TABLE 6.2
Sample Documentation Produced throughout
the Software Life Cycle Corresponding to
Each Phase in the Waterfall Model

Phase	Document
Concept	White paper
Requirements	Software requirements specification
Design	Software design description
Code	Documented code
Test	Test report
Maintenance	Change requests, change reports

Similar documentation can be expected along the way for each of the other software life cycle models. The software requirements specification and software design description documentation have already been discussed, along with documentation of the code.

It is beyond the scope of this text to describe each document in detail. The form the documentation takes depends on organizational preferences, customer demands, and the software process model and standards that are adopted. In any event, carefully written and edited documentation is an important software artifact — not just an onerous chore undertaken to satisfy management. Indeed, the first step in dealing with legacy systems or acquired software systems is the hunt for extant documentation.

6.6 EXERCISES

- 6.1 Which of the languages discussed in this chapter provide for some sort of GOTO statement? Does the GOTO statement affect performance? If so, how?
- 6.2 It can be argued that in some cases there exists an apparent conflict between good software engineering techniques and real-time performance. Consider the relative merits of recursive program design vs. interactive techniques, and the use of global variables vs. parameter lists. Using these topics and an appropriate programming language for examples, compare and contrast real-time performance vs. good software engineering practices as you understand them.
- 6.3 What other compiler options are available for your compiler and what do they do?
- 6.4 Why do you think that it is impossible to bypass the preprocessor phase of the compilation process?
- 6.5 In the object-oriented language of your choice, design and code an “image” class that might be useful across a wide range of projects. Be sure to follow the best principles of object-oriented design.
- 6.6 In a procedural language of your choice, develop an abstract data type called image with associated functions. Be sure to follow the principle of information hiding.
- 6.7 Write a set of coding standards for use with imaging systems for the programming language of your choice. Document the rationale for each provision of the coding standard.
- 6.8 Develop a set of tests to exercise a compiler to determine the best use of the language in an image processing environment. For example, your tests should determine such things as when to use case statements vs. nested if-then-else statements, when to use integers vs. Boolean variables for conditional branching, whether to use while or for loops and when, and so on.
- 6.9 How can misuse or misunderstanding of a software technology impede a software project? For example, writing structured C code instead of

classes in C++, or reinventing a tool for each project instead of using a standard one.

- 6.10 Compare how Ada 95 and Java handle the infamous GOTO statement. What does this indicate about the design principles or philosophy of each language?
- 6.11 Java has been compared to Ada 95 in terms of hype and unification — defend or attack the arguments against this.
- 6.12 Are there language features that are exclusive to C and C++? Do these features provide any advantage or disadvantage in embedded environments?

7 Software Measurement and Testing

In a few minutes a computer can make a mistake so great that it would take many men months to equal it.

Merle L. Meacham

7.1 THE ROLE OF METRICS

The key to controlling anything is measurement. Software is no different in this regard, but the following question arises: What aspects of software can be measured? Chapter 2 introduced several important software properties and alluded to their measurement. It is now appropriate to examine the measurement of these properties and show how this data can be used to monitor and manage the development of software.

Metrics can be used in software engineering in several ways. First, certain metrics can be used during software requirements development to assist in cost estimation. Another useful application for metrics is benchmarking. For example, if a company has a set of successful systems, then computing metrics for those systems yields a set of desirable and measurable characteristics with which to seek or compare in future systems. Most metrics can be used for testing in the sense of measuring the desirable properties of the software and setting limits on the bounds of those criteria.

Of course, metrics can be used to track project progress. In fact, some companies reward employees based on the amount of software developed per day as measured by some of the metrics to be discussed (e.g., delivered source instructions (DSIs), function points, or lines of code).

Finally, metrics can be used during the testing phase and for debugging purposes to help focus on likely sources of errors.

7.1.1 LINES OF CODE

The easiest characteristic of software that can be measured is the number of lines of finished source code. Measured as thousands of lines of code (KLOC), the “clock” metric is also referred to as DSIs or noncommented source code statements (NCSSs). That is, the number of executable program instructions, excluding comment statements, header files, formatting statements, macros, and anything that does not show up as executable code after compilation or cause allocation of memory, are counted.

Another related metric is source lines of code (SLOCs), the major difference being that a single SLOC may span several lines. For example, an if-then-else statement could be a single SLOC, but many DSIs.

While the clock metric essentially measures the weight of a printout of the source code, thinking in these terms makes it likely that the usefulness of KLOC will be unjustifiably dismissed as supercilious. But is not it likely that 1000 lines of code is going to have more errors than 100 lines of code? Would it not take longer to develop the latter than the former? Of course, the answer is dependent on how complex the code is.

One of the main disadvantages of using lines of source code as a metric is that it can only be measured after the code has been written. While it can be estimated beforehand and during software production based on similar projects, this is far less accurate than measuring the code after the fact. Nevertheless, KLOC is a useful metric, and in many cases is better than measuring nothing. Moreover, many other metrics are fundamentally based on lines of code.

For example, a closely related metric is delta KLOC (sometimes derisively referred to as release turmoil). Delta KLOC measures how many lines of code change over some period of time. Such a measure is useful, perhaps, in the sense that as a project nears the end of code development, delta KLOC would be expected to be small. Other, more substantial metrics are also derived from KLOC.

7.1.2 MCCABE'S METRIC

A valid criticism of the KLOC metric is that it does not take into account the complexity of the software involved. For example, 1000 lines of print statements probably do have fewer errors than 100 lines of a complex imaging algorithm.

To attempt to measure software complexity, McCabe (1976) introduced the metric, cyclomatic complexity, which measures program flow of control. This concept fits well with procedural programming, but not necessarily with object-oriented programming, though there are adaptations for use with the latter. In any case, this metric has two primary uses:

1. To indicate escalating complexity in a module as it is coded, therefore assisting the coders in determining the "size" of their modules
2. To determine the upper bound on the number of tests that must be designed and executed

7.1.2.1 Measuring Software Complexity

The cyclomatic complexity is based on determining the number of linearly independent paths in a program module, suggesting that the complexity increases with this number, and reliability decreases.

To compute the metric, the following procedure is followed. Consider the flow graph of a program. Let e be the number of edges and n the number of nodes. Form the cyclomatic complexity, C , as follows:

$$C = e - n + 2 \tag{7.1}$$

This is the most generally accepted form.

To get a sense of the relationship between program flow and cyclomatic complexity, refer to Figure 7.1. Here, for example, a sequence of instructions has two nodes, one edge and one region, and hence would have a complexity of $C = 1 - 2 + 2 = 1$. This is intuitively pleasing, as nothing could be less complex than a simple sequence.

On the other hand, the case statement, which has six edges and five nodes, would contribute $C = 6 - 5 + 2 = 3$ to the overall complexity.

As a more substantial example, consider a segment of code extracted from the noise reduction portion of the visual inspection system. The procedure calls between modules a, b, c, d, e, and f are depicted in Figure 7.2. Here, then, $e = 9$ and $n = 6$, yielding a cyclomatic complexity of $C = 9 - 6 + 2 = 5$.

Computation of McCabe’s metric can be done easily during compilation by analyzing the internal tree structure generated during the parsing phase (see Chapter 6). However, commercial tools are available to perform this analysis.

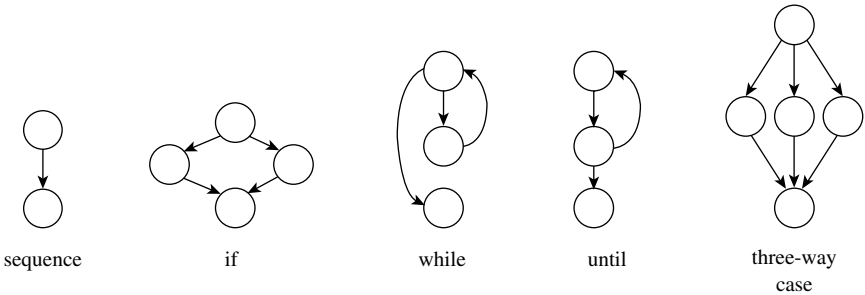


FIGURE 7.1 Correspondence of language statements and flow graph. (Adapted from Pressman, R.S., *Software Engineering: A Practioner’s Approach*, 5th ed., McGraw-Hill, New York, 2000.)

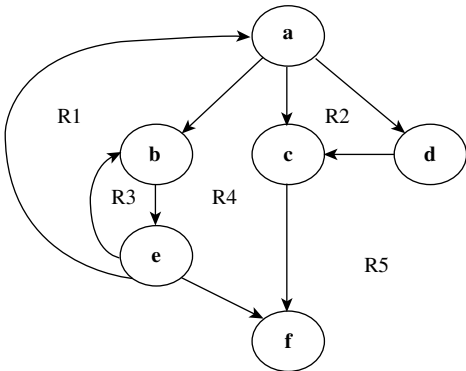


FIGURE 7.2 Flow graph for noise reduction code for the visual inspection system.

7.1.2.2 Determining the Limit on Number of Test Cases

To determine the upper limit on the number of test cases, McCabe developed an algorithmic procedure (called the baseline method) to determine a set of basis paths.

First, a clever construction is followed to force the complexity graph to look like a vector space by defining the notions of scalar multiplication and addition along paths. The basis vectors for this vector space are then determined.

The method proceeds with the selection of a baseline path, which should correspond to some “ordinary” case of program execution along one of the basis vector paths. McCabe advises choosing a path with as many decision nodes as possible. Next, the baseline path is retraced, and in turn, each decision is reversed; that is, when a node of outdegree of greater than 2 is reached, a different path must be taken. Continuing in this way until all possibilities are exhausted generates a set of paths representing the test set (Jorgensen, 2002). It turns out that the number of test cases generated is precisely C , the cyclomatic complexity.

The technique is rather complicated; thus, it is best to consult an excellent reference on testing, such as Jorgensen (2002), for the details.

7.1.3 HALSTEAD’S METRICS

One of the drawbacks of McCabe’s metric is that it measures complexity as a function of control flow. But complexity can exist internally in the way that the programming language is used.

Halstead’s metrics measure information content, or how intensively the programming language is used. Halstead’s metrics are computed using the following, slightly modified algorithm:

- First, find n_1 . This is essentially the number of distinct, syntactic begin–end pairs (or their equivalent), called operators.
- Next, find n_2 , the number of distinct statements. A statement is determined by the syntax of the language; for example, a line terminated by a semi-colon is a statement in C.
- Next, count N_1 , the total number of occurrences of n_1 in the program.
- Finally, count N_2 , the total number of occurrences of operands or n_2 in the program.

From these statistics the following Halstead’s metrics can be computed.

The program vocabulary, n , is defined as

$$n = n_1 + n_2 \quad (7.2)$$

The program length, N , is defined as

$$N = N_1 + N_2 \quad (7.3)$$

The program volume, V , is defined as

$$V = N \log_2 n \quad (7.4)$$

The potential volume, V^* , is defined as

$$V^* = (2 + n_2) \cdot \log_2 (2 + n_2) \quad (7.5)$$

The program level, L , is defined as

$$L = V^*/V \quad (7.6)$$

L is a measure of the level of abstraction of the program. It is believed that increasing this number will increase system reliability.

Another Halstead metric measures the amount of mental effort required in the development of the code. The effort, E , is defined as

$$E = V/L \quad (7.7)$$

Again, decreasing the effort level is believed to increase reliability as well as ease of implementation.

In principle, the program length, N , can be estimated, and therefore is useful in cost and schedule estimation. The length is also a measure of the complexity of the program in terms of language usage; thus, it can be used to estimate defect rates.

Halstead's metrics, though dating back almost 30 years, are still widely used, and tools are available to completely automate their determination.

7.1.4 FUNCTION POINTS

Function points were introduced in the late 1970s as an alternative to metrics based on simple source line count. The basis of function points is that as more powerful programming languages are developed, the number of source lines necessary to perform a given function decreases. Paradoxically, however, the cost/lines of code measure indicated a reduction in productivity, as the fixed costs of software production were largely unchanged.

The solution is to measure the functionality of software via the number of interfaces between modules and subsystems in programs or systems. A big advantage of the function point metric is that it can be calculated before any coding occurs based solely on the design description.

The following five software characteristics for each module, subsystem, or system represent its function points:

1. Number of inputs to the application (I)
2. Number of outputs (O)
3. Number of user inquiries (Q)
4. Number of files used (F)
5. Number of external interfaces (X)

Now consider empirical weighting factors for each aspect that reflect their relative difficulty in implementation. For example, one set of weighting factors for a particular kind of system might yield the function point (*FP*) value

$$FP = 4I + 4O + 5Q + 10F + 7X \quad (7.8)$$

Intuitively, the higher *FP*, the more difficult the system is to implement. Moreover, a great advantage of the function point metric is that it can be computed before any coding occurs.

The weights given in Equation 7.8 can be adjusted to compensate for factors such as application domain and software developer experience. For example, if W_i are the weighting factors, F_j the complexity adjustment factors, and A_i the item counts, then *FP* is defined as

$$FP = \sum (A_i \times W_i) \times \left[0.65 + 0.01 \times \sum F_j \right] \quad (7.9)$$

The complexity factor adjustments can be adapted for other application domains, such as embedded and real-time systems and even, specifically, imaging systems. To determine the complexity factor adjustments, a set of 14 questions (see below) are answered by the software engineer(s) with responses from a scale of 0 to 5:

- 0 = No influence
- 1 = Incidental
- 2 = Moderate
- 3 = Average
- 4 = Significant
- 5 = Essential

For example, in the visual inspection system suppose the engineering team was queried and the following answers to the questions were obtained:

- Question 1: Does the system require reliable backup and recovery? Yes, this is a rather critical system; assign a 4.
- Question 2: Are data communications required? Yes, there is communication between various components of the system over a standard bus; assign a 5.
- Question 3: Are there distributed processing functions? Yes; assign a 5.
- Question 4: Is performance critical? Absolutely, this is a hard real-time system; assign a 5.
- Question 5: Will the system run in an existing, heavily utilized operational environment? In this case yes; assign a 5.
- Question 6: Does the system require on-line data entry? Yes, via sensors and some operator input; assign a 4.

- Question 7: Does the on-line data entry require the input transactions to be built over multiple screens or operations? Yes; assign a 4.
- Question 8: Are the master files updated on-line? Yes; assign a 5.
- Question 9: Are the inputs, outputs, files, or inquiries complex? Yes, they involve comparatively complex sensor inputs; assign a 4.
- Question 10: Is the internal processing complex? Clearly it is; the imaging and pattern recognition algorithms are nontrivial; assign a 4.
- Question 11: Is the code designed to be reusable? Yes, there are high up-front development costs, and multiple applications have to be supported for this investment to pay off; assign a 4.
- Question 12: Are the conversion and installation included in the design? In this case, yes; assign a 5.
- Question 13: Is the system designed for multiple installations in different organizations? Absolutely, this must be a highly flexible system; assign a 5.
- Question 14: Is the application designed to facilitate change and ease of use by the user? Yes, absolutely; assign a 5.

Then, applying Equation 7.9 yields

$$.01 \sum F_j = .01 \cdot (6 \cdot 4 + 8 \cdot 5) = 0.64$$

Now suppose that it was determined from the software requirements specification that the item counts were as follows:

$$\begin{aligned} A_1 &= I = 5 \\ A_2 &= U = 7 \\ A_3 &= Q = 8 \\ A_4 &= F = 5 \\ A_5 &= X = 5 \end{aligned}$$

Using the weighting factors from Equation 7.7,

$$\begin{aligned} W_1 &= 4 \\ W_2 &= 4 \\ W_3 &= 5 \\ W_4 &= 10 \\ W_5 &= 7 \end{aligned}$$

and putting them into Equation 7.9 yields

$$\begin{aligned}
 FP &= [4 \cdot 5 + 4 \cdot 7 + 5 \cdot 8 + 10 \cdot 5 + 7 \cdot 7] [0.65 + 0.64] \\
 &= 241
 \end{aligned}$$

For the purposes of comparison, and as a management tool, function points have been mapped to the relative lines of source code, in particular programming languages. These are shown in Table 7.1.

For example, it seems intuitively pleasing that it would take many more lines of assembly language code to express functionality than it would a high-level language like C. In the case of the visual inspection system, with $FP = 241$, it might be expected that about 31,000 lines of code would be needed to implement the functionality. In turn, it should take many less to express that same functionality in a more abstract language such as C++. The same observations that apply to software production might also apply to maintenance, as well as to the potential reliability of software.

Imaging applications like the visual inspection systems are highly complex and have many complexity factors rated at 5. In other kinds of systems, such as database applications, these factors would be much lower. This is an explicit statement about the difficulty in building and maintaining code for embedded systems vs. nonembedded ones.

The function point metric has mostly been used in business processing, and not nearly as much in embedded systems. However, there is increasing interest in the use of function points in real-time embedded systems, especially in large-scale real-time databases, multimedia, and Internet support. These systems are data driven and often behave like the large-scale transaction-based systems for which function points were developed.

The International Function Point Users Group maintains a Web database of weighting factors and function point values for a variety of application domains. These can be used for comparison.

TABLE 7.1
Programming Language and Lines of
Code per Function Point

Language	Lines of Code per Function Point
Assembly	320
C	128
Fortran	106
Pascal	90
C++	64

Source: Adapted from Jones, C., *Estimating Software Costs*, McGraw-Hill, New York, 1998.

7.1.5 FEATURE POINTS

Feature points are an extension of function points developed by Software Productivity Research, Inc., in 1986. Feature points address the fact that the classical function point metric was developed for management information systems, and therefore is not particularly applicable to many other systems, such as real-time, embedded, communications, and process control software. The motivation is that these systems exhibit high levels of algorithmic complexity, but sparse inputs and outputs.

The feature point metric is computed in a manner similar to that of the function point metric, except that a new factor for the number of algorithms, A , is added. The empirical weightings are as follows:

$$W_1 = 3$$

$$W_2 = 4$$

$$W_3 = 5$$

$$W_4 = 4$$

$$W_5 = 7$$

$$W_6 = 7$$

The feature point metric, \overline{FP} , is then

$$\overline{FP} = 3I + 4O + 5Q + 4F + 7X + 7A \quad (7.10)$$

For example, in the visual inspection system, using the same item counts as computed before, supposing that the item count for algorithms, A , is 10, and using the same complexity adjustment factor, \overline{FP} would be computed as follows:

$$\begin{aligned} \overline{FP} &= [3 \cdot 5 + 4 \cdot 7 + 5 \cdot 8 + 4 \cdot 10 + 7 \cdot 7 + 7 \cdot 10] [0.65 + 0.64] \\ &= 312 \end{aligned}$$

If the system were to be written in C, it could be estimated that approximately 40,000 lines of code would be needed — a slightly more pessimistic estimate than that computed using the function point metric.

7.1.6 METRICS FOR OBJECT-ORIENTED SOFTWARE

While any of the previously discussed metrics can be used in object-oriented code, other metrics are better suited for this setting. For example, some of the metrics that have been used include:

- A weighted count of methods per class
- The depth of inheritance tree
- The number of children in the inheritance tree
- The coupling between object classes
- The lack of cohesion in methods

As with other metrics, the key to use is consistency.

7.1.7 OBJECTIONS TO METRICS

There are many who object to the use of metrics in one or all of the ways that have been described.

One counterargument to the use of certain metrics (in some cases, any metrics) is that they can be misused or that they are a costly and unnecessary distraction. For example, metrics related to the number of lines of code imply that the more powerful the language, the less productive the programmer. Hence, obsessing with code production based on lines of code is a meaningless endeavor.

Another objection is that measuring the correlation effects of a metric without clearly understanding the causality is unscientific and dangerous. For example, while there are numerous studies suggesting that lowering the cyclomatic complexity leads to more reliable software, there just is not any real way to know why. Obviously, the arguments about the complexity of well-written code vs. “spaghetti code” apply, but there is just no way to show the causal relationship. So, the opponents of metrics might argue that if a study of several companies showed that software written by software engineers who always wore yellow shirts had statistically significant less defects in their code, then companies would start requiring a dress code of yellow shirts. This illustration is, of course, hyperbole, but the point of correlation vs. causality is made.

While it is possible that in many cases these objections might be valid, like most things, metrics can be either useful or harmful, depending on how they are used (or abused).

7.2 FAULTS, FAILURES, AND BUGS

There is more than a subtle difference between the terms fault, failure, bug, and defect. Use of bug is, in fact, discouraged, since it implies that an error somehow crept into the program through no one’s action. The preferred term for an error in requirement, design, or code is error or defect. The manifestation of a defect during the operation of the software system is called a fault. A fault that causes the software system to fail to meet one of its requirements is a failure.*

The text opened with a discussion of the economic cost of software errors as reported by National Institute of Standards Technology (NIST). Findings from the other survey referenced give some sense of the cost of these errors: 43% reported that error severity in their systems was significant and 34% believed that the cor-

* Some define a fault as an error found prior to system delivery and a defect as an error found post delivery.

rective hours needed to resolve run-time problems were not minimal (Laplante et al., 2002e).

7.3 THE ROLE OF TESTING

Verification determines whether the products of a given phase of the software development cycle fulfill the requirements established during the previous phase. Verification answers the question “Am I building the product right?”

Validation determines the correctness of the final program or software with respect to the user’s needs and requirements. Validation answers the question “Am I building the right product?”

Testing is the execution of a program or partial program with known inputs and outputs that are both predicted and observed for the purpose of finding faults or deviations from the requirements.

Although testing will flush out errors, this is just one of its purposes. The other is to increase trust in the system. Perhaps at one time software testing was thought of as intended to remove all errors. But testing can only detect the presence of errors, not the absence of them; therefore, it can never be known when all errors have been detected. Instead, testing must increase faith in the system, even though it still may contain undetected faults, by ensuring that the software meets its requirements. This objective places emphasis on solid design techniques and a well-developed requirements document. Moreover, a formal test plan must be developed that provides criteria used in deciding whether the system has satisfied the requirements documents.

7.4 TESTING TECHNIQUES

There are a wide range of testing techniques for unit- and system-level testing, desk checking, and integration testing. These techniques are often interchangeable, while others are not. Any one of these test techniques can be either insufficient or not computationally feasible. Therefore, some combination of testing techniques is almost always employed.

7.4.1 UNIT-LEVEL TESTING

Several methods can be used to test individual modules or units. These techniques can be used by the unit author and by the independent test team to exercise each unit in the system. These techniques can also be applied to subsystems (collections of modules related to the same function). The techniques to be discussed include black box and white box testing.

7.4.1.1 Black Box Testing

In black box testing, only inputs and outputs of the unit are considered; how the outputs are generated based on a particular set of inputs is ignored. Such a technique, being independent of the implementation of the module, can be applied to any number of modules with the same functionality. But this technique does not provide insight into the programmer’s skill in implementing the module. In addition, dead or unreachable code cannot be detected.

For each module, a number of test cases need to be generated. This number depends on the functionality of the module, the number of inputs, and so on. If a module fails to pass a single-module-level test, then the error must be repaired, and all previous module-level test cases are rerun and passed, to prevent the repair from causing other errors.

Some widely used black box testing techniques include:

- Exhaustive testing
- Boundary value testing
- Random test generation
- Worst-case testing

An important aspect of using black box testing techniques is that clearly defined interfaces to the modules are required. This places additional emphasis on the application of Parnas partitioning principles to module design.

7.4.1.1.1 Exhaustive Testing

Brute force or exhaustive testing involves presenting each code unit with every possible input combination. Brute force testing can work well in the case of a small number of inputs, each with a limited input range, for example, a code unit that evaluates a small number of Boolean inputs. A major problem with brute force testing, however, is the combinatorial explosion in the number of test cases. For example, for the code that will threshold the raw binary code for a 1024×1024 binary image, $2^{1024 \cdot 1024}$ test cases would be required, which is clearly beyond the realm of practicality.

7.4.1.1.2 Boundary Value Testing

Boundary value or corner case testing solves the problem of combinatorial explosion by testing some very tiny subset of the input combinations identified as meaningful “boundaries” of input.

For example, consider a code unit with five different inputs, each of which is a 16-bit signed integer. Approaching the testing of this code unit using exhaustive testing would require $2^{16} \cdot 2^{16} \cdot 2^{16} \cdot 2^{16} \cdot 2^{16} = 2^{80}$ test cases. However, if the test inputs are restricted to every combination of the minimum, maximum, and average values for each input, then the test set would consist of $3^5 = 243$ test cases. A test set of this size can be handled easily with automatic test case generation.

7.4.1.1.3 Random Test Case Generation

Random test case generation, or statistically based testing, can be used for both unit- and system-level testing. This kind of testing involves subjecting the code unit to many randomly generated test cases over some period. The purpose of this approach is to simulate execution of the software under realistic conditions.

The randomly generated test cases are based on determining the underlying statistics of the expected inputs. The statistics are usually collected by expert users of similar systems or, if none exist, by educated guessing. The theory is that system reliability will be enhanced if prolonged usage of the system can be simulated in a controlled environment.

The major drawback of such a technique is that the underlying probability distribution functions for the input variables may be unavailable or incorrect. In addition, randomly generated test cases are likely to miss conditions with a low probability of occurrence. Precisely this kind of condition is usually overlooked in the design of the module. Failing to test these scenarios is an invitation to disaster.

7.4.1.1.4 Worst-Case Testing

Worst-case or pathological case testing deals with those test scenarios that might be considered highly unusual and unlikely. It is often the case that these exceptional cases are exactly those for which the code is likely to be poorly designed, and therefore to fail.

For example, in the visual inspection system, while it might be highly unlikely that the system is to function at the maximum conveyor speed, this worst case still needs to be tested.

7.4.1.2 White Box Testing

One disadvantage of black box testing is that it can often bypass unreachable or dead code. In addition, it may not test all of the control paths in the module. Another way to look at this is that black box testing only tests what is expected to happen — not what was not intended. White or clear box testing techniques can be used to deal with this problem.

Whereas black box tests are data driven, white box tests are logic driven; that is, they are designed to exercise all paths in the code unit. For example, in the reject mechanism functionality of the visual inspection system, all error paths would need to be tested, including those pathological situations that deal with simultaneous and multiple failures.

White box testing also has the advantage that it can discover those code paths that cannot be executed. This unreachable code is undesirable because it is likely a sign that the logic is incorrect, because it wastes code space memory, and because it might inadvertently be executed in the case of the corruption of the computer's program counter.

7.4.1.2.1 Code Inspections

Group walk-throughs or code inspections are a kind of white box testing in which code is inspected line by line. Walk-throughs have been shown to be much more effective than other types of testing.

In code inspections, the author of some collection of software presents each line of code to a review group, which can detect errors as well as discover ways for improving the implementation. This audit also provides excellent control of the coding standards. Finally, unreachable code can be discovered.

The conduct of audits and reviews was discussed in the previous chapter.

7.4.1.2.2 Formal Program Proving

Formal program proving is a kind of white box testing using formal methods in which the code is treated as a theorem and some form of calculus is used to prove that the program is correct. This form of verification requires a high level of training

and is useful, generally, for only limited purposes because of the intensity of activity required. There are also difficulties dealing with temporal behavior.

7.4.2 TESTING OBJECT-ORIENTED SOFTWARE

A test process that complements object-oriented design and programming can significantly increase reuse, quality, and productivity. There are three issues in testing object-oriented software:

1. Testing the base class
2. Testing external code that uses a base class
3. Dealing with inheritance and dynamic binding

Without inheritance, object-oriented programming is not very different from simply testing abstract data types, such as an image. This image object has some data structure, such as an array, and a set of member functions to operate. There are also member functions to operate on the image. These member functions are tested like any other using black box or white box techniques.

In a good object-oriented design there should be a well-defined inheritance structure. Therefore, most of the tests from the base class can be used for testing the derived class, and only a small amount of retesting of the derived class is required. On the other hand, if the inheritance structure is bad, for example, if there is inheritance of implementation (where code is grabbed from the base class), then additional testing will be necessary. Hence, the price of using inheritance poorly is having to retest all of the inherited code.

Finally, dynamic binding requires that all cases be tested for each binding possibility.

Effective testing is guided by information about likely sources of error. The combination of polymorphism, inheritance, and encapsulation is unique to object-oriented languages, presenting opportunities for error that do not exist in conventional languages. The main rule here is that if a class is used in a new context, then it should be tested as if it were new.

7.4.3 SYSTEM-LEVEL TESTING

Once individual modules have been tested, then subsystems or the entire system needs to be tested. In larger systems, the process can be broken down into a series of subsystem tests and then a test of the overall system.

System testing treats the system as a black box so that one or more of the black box testing techniques can be applied. System-level testing always occurs after all modules pass their unit test. At this point, the coding team hands the software over to the test team for validation.

If an error occurs during system-level testing, it must be repaired. Ideally, every test case involving the changed module must be rerun, and all previous system-level tests must be passed in succession. The collection of system test cases is often called a system test suite.

Burn-in testing is a type of system-level testing that seeks to flush out those failures appearing early in the life of the system, and thus to improve the reliability of the delivered product.

System-level testing is usually followed by alpha testing, which is a type of validation consisting of internal distribution and exercise of the software. This testing is followed by beta testing, where preliminary versions of validated software are distributed to friendly customers who test the software under actual use. Later in the life cycle of the software, if corrections or enhancements are added, then regression testing is performed.

Regression testing (which can also be performed at the module level) is used to validate the updated software against the old set of test cases that have already been passed. Any new test case needed for the enhancements is then added to the test suite, and the software is validated as if it were a new product. Regression testing is also an integral part of integration testing as new modules are added to the tested subsystem.

7.4.3.1 Cleanroom Testing

The principal tenant of cleanroom software development is that given sufficient time and with care, error-free software can be written. Cleanroom software development relies heavily on group walk-throughs, code inspections, code reading by stepwise abstraction, and formal program validation. It is taken for granted that software specifications exist that are sufficient to completely describe the system.

In this approach, the development team is not allowed to test code as it is being developed. Rather, syntax checkers, code walk-through, group inspections, and formal verifications are used to ensure product integrity. Statistically based testing is then applied at various stages of product development by a separate test team. This technique reportedly produces documentation and code that are more reliable and maintainable and easier to test than other development methods.

The program is developed by slowly “growing” features into the code, starting with some baseline of functionality. At each milestone, an independent test team checks the code against a set of randomly generated test cases based on a set of statistics describing the frequency of use for each feature specified in the requirements.

This group tests the code incrementally at predetermined milestones and either accepts it or returns it to the development team for correction. Once a functional milestone has been reached, the development team adds to the “clean” code, using the same techniques as before. Thus, like an onion’s skin, new layers of functionality are added to the software system unit it has completely satisfied the requirements.

Numerous projects have been developed in this way, in both academic and industrial environments. In any case, many of the tenants of cleanroom testing can be incorporated without completely embracing the methodology.

7.4.3.2 Stress Testing

In another type of testing, stress testing, the system is subjected to a large disturbance in the inputs (for example, a large burst of interrupts), followed by smaller distur-

bances spread out over a longer period. One objective of this kind testing is to see how the system fails (gracefully or catastrophically).

Stress testing can also be useful in dealing with cases and conditions where the system is under heavy load, for example, in testing for memory or processor utilization in conjunction with other application and operating system resources to determine if performance is acceptable.

7.5 DESIGN OF TESTING PLANS

The test plan should follow the requirements document item by item, providing criteria that are used to judge whether the required item has been met. A set of test cases are then written that are used to measure the criteria set out in the test plan. Writing such test cases can be extremely difficult when a user interface is part of the requirements.

The test plan includes criteria for testing the software on a module-by-module or unit level, and on a system or subsystem level; both should be incorporated in a good testing scheme. The system-level testing provides criteria for the hardware or software integration process.

Other documentation may be required, particularly in Department of Defense (DOD)-style software development, where preliminary and final documents are required and where additional documentation such as the hardware integration plan, software integration plan, and so on, may be required.

Many software systems that interact directly or indirectly with humans also require some form of user's manual to be developed and tested.

7.6 EXERCISES

- 7.1 Recalculate McCabe's metric for the if, while, and until structures in Figure 7.1.
- 7.2 Research the use of McCabe's metric in imaging systems by searching the literature.
- 7.3 Recalculate the *FP* metric for the visual inspection system set of weightings that assumes that significant off-the-shelf software (say 70%) is to be used. Make assumptions about which factors will be most influenced by the off-the-shelf software. How many lines of C code do you estimate you will need?
- 7.4 Redo Exercise 7.3, except recalculate the feature point metric. How many lines of C code do you estimate will be needed?
- 7.5 For the visual inspection system, which testing approaches would you use? When and why?
- 7.6 If the visual inspection system were written in C++ according to the design fragment described in Chapter 5, describe the testing strategy you would use. If possible, try to design some test cases.
- 7.7 How much can testing and test case/suite generation be automated? What are the roadblocks to automating a test suite? In languages like Java?

8 Hardware–Software Integration and Maintenance

Planned obsolescence is another word for progress.

James Jeffrey Roch

8.1 GOALS OF SYSTEM INTEGRATION

Integration is the process of combining partial functionality to form the overall system functionality. Because imaging systems are embedded, the integration process involves both multiple software units and hardware. Each of these parts has potentially been developed by different teams or individuals within the project organization. Although they have been rigorously tested and verified separately, the overall behavior of the system, and conformance with most of the software requirements, cannot be tested until the system is wholly integrated. Software integration can be further complicated when both hardware and software are new.

The software integration activity has the most uncertain schedule and is typically the cause of project cost overruns. Moreover, the stage has been set for failure or success at this phase by the design and implementation practices used throughout the software project life cycle. Hence, by the time of software integration, it may be very difficult to fix problems. Indeed, many modern programming practices were devised to ensure arrival at this stage with the fewest errors in the source code. For example, Ada has built-in tests for consistency of argument lists, and the C programming language uses an associated utility program called **lint** to do the same. Automation can help avoid the integration troubles mentioned in this chapter.

8.2 SYSTEM UNIFICATION

Fitting the pieces of the system together from its individual components is tricky business, especially for imaging systems. Parameter mismatching, variable name mistyping, and calling sequence errors are some of the problems possibly encountered during system integration. Even the most rigorous unit-level testing cannot eliminate these problems completely.

The system unification process consists of linking together the tested software modules drawn in an orderly fashion from the source code library. During the linking

TABLE 8.1
Sample Test Log for Visual Inspection System

Test Number	Reference Requirements #	Test Name	Pass/Fail	Date	Tester
S121	3.2.2.2	Detect shape 1a	Pass	5/16/03	P.L.
S122	3.2.2.2	Detect shape 1b	Pass	5/16/03	P.L.
S123	3.2.2.2	Detect shape 1c	Fail	5/16/03	P.L.

process, errors are likely to occur that relate to unresolved external symbols, memory assignments violations, page link errors, and the like.

These problems must, of course, be resolved. Once resolved, the loadable code, called a load module, can be downloaded from the development environment to the target machine. This is achieved in a variety of ways depending on the system architecture, but it can include tapes, disks, network connections, modems, or use of an intermediate computer. In any case, once the load module has been created and loaded into the target machine, testing of timing and hardware–software interaction can begin.

8.3 SYSTEM VERIFICATION

Final system testing of embedded systems can be a tedious process, often requiring days or weeks. During system validation, a careful test log must be kept, indicating the test case number, results, and disposition. Table 8.1 is a sample of such a test log for the visual inspection system.

If a system test fails, it is imperative, once the problem has been identified and presumably corrected, that all affected tests be rerun. These include:

- 1. All module-level test cases for any module that has been changed
- 2. All system-level test cases

Even though the module-level test cases and previous system-level test cases have been passed, it is imperative that these be rerun to ensure that no side effects have been introduced during error repair.

8.4 SYSTEM INTEGRATION TOOLS

As mentioned before, it is not always easy to identify sources of error during a system test. A number of hardware and software tools are available to assist in the validation of embedded systems. Remember that test tools make the difference between success and failure — especially in deeply embedded systems.

8.4.1 MULTIMETER

The use of a multimeter in the debugging of imaging systems may seem odd, but it is an important tool in embedded systems where the software controls or reads analog values through hardware. The multimeter measures voltage, current, or power, and can be used to validate the analog input or output into the system.

8.4.2 OSCILLOSCOPE

An oscilloscope, like a multimeter, is not always regarded as a software-debugging tool, but it is useful in embedded software environments. Oscilloscopes range from the basic single-trace variety to storage oscilloscopes with multiple traces. Oscilloscopes can be used for validating interrupt integrity, discrete signal issuance, and receipt, and for monitoring clocks. The more sophisticated storage oscilloscopes with multiple inputs can often be used in lieu of logic analyzers, by using the inputs to track the data and address buses and synchronization with an appropriate clock.

8.4.3 LOGIC ANALYZER

The logic analyzer, an important tool for debugging software, especially in embedded imaging systems, can be used to capture data or events, to measure individual instruction times, or to time sections of code. Moreover, the introduction of programmable logic analyzers with integrated debugging environments has further enhanced the capabilities of the system integrator.

More sophisticated logic analyzers include built-in disassemblers and compilers for source-level debugging and performance analysis. These integrated environments typically are found on more expensive models, but they make the identification of performance bottlenecks particularly easy.

No matter how elaborate, all logic analyzers have the same basic functionality. This is shown in Figure 8.1. The logic analyzer is connected to the system under test via probes that sit directly on the memory and data buses. A clock probe connects to the memory access synchronization clock. Upon each memory access, the data

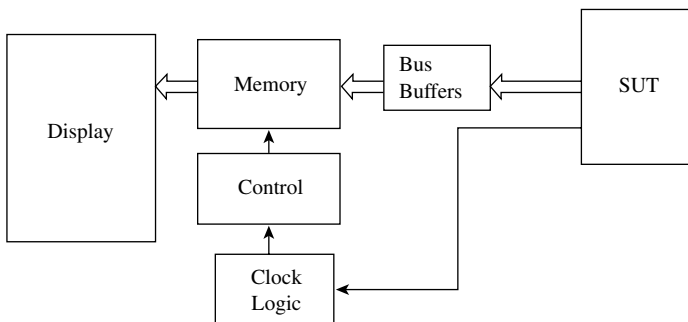


FIGURE 8.1 Basic logic analyzer structure connected to the system under test (SUT).

and address are captured by the logic analyzer and stored in buffers for transfer to the logic analyzer’s main memory, processing for display, and display to the screen.

Using the logic analyzer, the software engineer can capture specific memory locations and data for the purposes of timing or for verifying execution of a specific segment of code.

8.4.3.1 Timing Instructions

The logic analyzer can be used to time an individual macroinstruction, segments of code, or an entire process. To time an individual instruction, the engineer finds a memory location in the code segment of memory containing the desired instruction. Then the logic analyzer is set to trigger on this opcode at the desired location, and on the opcode and location of the next instruction. The trace is set for absolute time. The logic analyzer will then display the difference in time between the fetch of the first instruction (the target) and the next instruction. This is the most accurate means for determining the instruction execution time.

For example, suppose a system contains a 30-msec frame buffering task. It is known from the linker output that instructions and data are found at memory location 4356 through 464B (hexadecimal). Then the memory location, corresponding numerical opcode (in hex), and symbolic equivalent will appear in the display of the logic analyzer as follows:

Location (Hex)	Opcode (Hex)	Instruction
4356	2321	STORE R2, R1
4357	4701 1000	LOAD R1, 1000
4359	2401 FC32	
•		
•		
•		
464B	6300 2000	JUMP 2000

The code represents part of an interrupt handler in which the first instruction is to disable all interrupts and the last instruction is to enable all interrupts. If the logic analyzer is set to trigger on address = 4357 and data = 4701, and to capture only address = 4357 and data = 4701, the time to complete the LOAD (4701) will be displayed. In this case the “data” is the instruction opcode.

8.4.3.2 Timing Code

The logic analyzer also provides an accurate method for measuring time to complete for any periodic task. To measure the total elapsed time for any task in the system, set the logic analyzer to trigger on the starting and ending addresses and opcode for the first instruction of that task. It should be the first instruction of the interrupt handler — usually a disable interrupt instruction. Disable the interrupts for all higher-priority cycles and set the trace for absolute time. The time displayed is the total time of that task cycle.

Consider the code shown in the previous example. If the logic analyzer is set to trigger on address = 4356 and data = 2321, and to capture only address = 464B and data = 6300, the absolute time to execute all instructions in the module will be measured. Suppose the elapsed time is 3, measured as milliseconds for a 10-msec rate. Then the utilization contribution from this code is 33.33% (see Section 8.6.1). This approach can be used to time one or several modules within a cycle, or even sections of code within a module.

8.4.4 IN-CIRCUIT EMULATOR

During module-level debug and system integration in conjunction with embedded systems, the ability to single-step the computer, set the program counter, and deposit and read from memory is extremely important. This capability in conjunction with the symbolic debugger, as in the Unix/Linux environment, was discussed in Chapter 6. In an embedded environment, however, this capability is provided by an in-circuit emulator. In-circuit emulation (ICE) uses special hardware in conjunction with software to emulate the target central processing unit (CPU) while providing the aforementioned features. Typically, the in-circuit emulator plugs into the chip carrier or card slot normally occupied by the CPU. External wires connect to an emulation system. Access to the emulator is provided directly or via a secondary computer.

In-circuit emulators are useful in software patching and for single-stepping through critical portions of code. In-circuit emulators are not typically useful in timing tests, however, because subtle timing changes can be introduced by the emulator.

In certain ICE systems, the symbol table may be too large to load. Privatization of certain global variables can be used to reduce the size of the symbol table. For example, in C, judicious use of the static data type during testing can reduce the number of variables in the global symbol table. This aids in the debugging process.

8.4.5 SOFTWARE SIMULATORS

When integrating and debugging embedded systems, software simulators are often needed to stand in for hardware or inputs that do not exist or that are not readily available, for example, to generate simulated images where real images are unavailable at the time. The author of the simulator code has a task that is by no means easy. The software must be written to mimic exactly the hardware specification, especially in timing characteristics. The simulator must be rigorously tested (unfortunately, this is sometimes not the case). Many systems have been successfully validated and integrated with software simulators only to fail when connected to the actual hardware.

8.4.6 HARDWARE PROTOTYPES

In the absence of the actual hardware system under control, simulation hardware may be preferable to software simulators. These devices might be required when the software is ready before the prototype hardware, or when it would be impossible to test the software on the actual hardware, such as in the control of a large nuclear plant.

Video image generators can simulate real life and can be useful for integration and testing, but they are not suitable for testing the underlying algorithms — real images from live action or tape are needed.

8.5 SOFTWARE INTEGRATION

A deliberate approach must be used when performing system to ensure system integrity. Failure to use one can lead to cost escalation and frustration. Software integration approaches are largely based on experience. The following represents a simple strategy for software integration based on significant experience.

8.5.1 A SIMPLE INTEGRATION STRATEGY

In any embedded operating system, it is important to ensure that all tasks in the system are being scheduled and dispatched properly. Thus, the first goal in integrating the embedded system is to ensure that each task is running at its prescribed rate, and that context is saved and restored. This is done without performing any functions within those tasks; functions are added later.

As discussed before, a logic analyzer is quite useful in verifying cycle rates by setting the triggers on the starting location of each of the tasks involved. During debugging, it is most helpful to establish the fact that cyclic processes are being called at the appropriate rates. Until the system cycles properly, the application code associated with each of the tasks should not be added.

The success of this method depends on the fact that one change at a time is made to the system so that when the system becomes corrupted, the problem can be isolated.

The overall approach is shown in Figure 8.2. It involves establishing a baseline of running kernel components (no applications programs). This ensures that interrupts are being handled properly and that all cycles are running at their prescribed rates, without worry about interference from the applications code. Once the baseline is established, small sections of applications code are added and the cycle rates verified. If an error is detected, it is patched if possible. If the patch succeeds in restoring the cycle rates properly, then more code is added. This ensures that the system is grown incrementally, with an appropriate baseline at each stage of the integration. This approach represents a phased integration with regression testing after each step.

8.5.2 PATCHING

The process of correcting errors in the code directly on the target machine is called patching. Patching allows minor errors detected during the integration process to be corrected directly on the target machine, without undergoing the tedious process of correcting the source code and creating a new load module. Patching requires an expert command of the opcodes for the target machine unless a macroassembly-level patching facility is available. It also requires an accurate memory map, which includes the contents of each address in memory, and a method for depositing directly

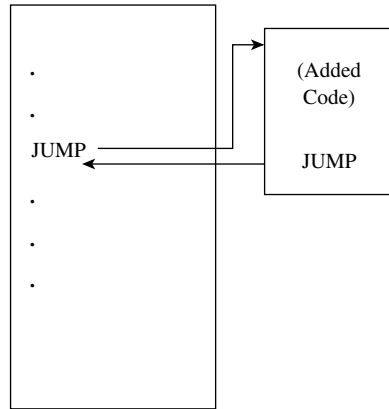


FIGURE 8.4 Placing an oversized patch.

The loading of patches during system integration can often be automated through the use of batch files. However, a large number of patches and patches on top of other patches can become confusing. It is imperative that a careful record be kept of all patches made, that the patches eventually find their way back to the source code, and that a new system be generated before validation testing begins. This is essential from a maintenance standpoint. Final testing should never be performed on a patched system.

Patching of software written in object-oriented languages is very difficult because of the lack of a straightforward mapping from the source code to the object code. Symbolic debuggers are quite helpful in this case, but even so, patching is risky in this situation at best.

8.5.3 THE PROBE EFFECT

The uncertainty principle, originally postulated by Werner Heisenberg in 1927, states essentially that the precise position and momentum of a particle cannot be known simultaneously. An analogy to the Heisenberg uncertainty principle applies in software integration. While software systems do not explicitly deal with electrons (except as ensemble behavior), the uncertainty arises because the more closely a system is examined, the more likely the examination process will affect the system. This fact is especially true for embedded systems where test probes can affect timing.

For example, an engineer is debugging an imaging system and discovers that a certain deadline is not being met. Some debugging code is added to print out a preliminary result to a file. But after adding the debugging code, the problem goes away. Declaring success, the engineer removes the debugging code and the problem reappears. In this case, it is clear that the debugging code somehow changed the timing of the system.

The software version of the Heisenberg uncertainty principle should be taken as a warning that testing methods often affect the systems that they test. When this

is the case, nonintrusive testing should be considered (for example, using a logic analyzer). Furthermore, wherever there is an inverse correlation between two variables affecting a system, Heisenberg uncertainty is suggested.

8.6 POSTINTEGRATION SOFTWARE OPTIMIZATION

In general, to achieve optimal performance in image processing algorithms, designers attempt to match the hardware to the problem. Failing this, a variety of techniques used in conjunction with high-level languages are employed to squeeze additional performance from the machine. These techniques include the use of assembly language patches and hand-tuning compiler output. Often, however, use of these practices leads to code that cannot be maintained and is unreliable because it is poorly documented. More desirable, then, is to use coding “tricks” that involve direct interaction with the high-level language and that can be documented. These tricks improve real-time performance, but generally not so at the expense of maintainability and reliability. Note that when optimizing average-case performance, worst-case performance is generally adversely affected.

8.6.1 CPU UTILIZATION ESTIMATION

CPU utilization estimates are measures that are meaningful primarily in cyclic real-time systems. In interrupt-driven systems, calculation of CPU utilization from measured data is easy only for periodic systems, to which this discussion is confined.

For periodic systems, CPU utilization is the sum of task execution times divided by the cycle times. In other fixed-rate, sporadic, or mixed systems, the maximum task execution period can be used in place of the cycle time. The CPU utilization T is given by Equation 8.1:

$$\sum_{i=1}^n A_i / T_i \quad (8.1)$$

where n is the number of tasks, T_i is the cycle time (or minimum time between occurrences), and A_i is the worst-case execution time for task i . A CPU utilization of 100% or higher is considered a time-overloaded condition and will lead to missed deadlines.

For example, consider an imaging system where data are gathered from sensors every 5 msec via an interrupt-driven routine (collecting the data takes 2.1 msec). A second process, which is initiated every 30 msec by an interrupt, processes the images and displays them on a visual display. This process requires 11 msec to complete. Finally, a 1-sec process performs hardware diagnostics and takes 5 msec to complete. The CPU utilization is then

$$2.1/5 + 11/30 + 5/1000 = 79.1\%$$

What is a desirable CPU utilization? Certainly not 100% or too close, or deadlines would be missed. A well-known result is that a system that has a large number of interrupt cycles cannot guarantee meeting all deadlines unless it is at 69% or less CPU utilization. In practice, however, around 80% is acceptable. Even so, the real problem in deadline satisfaction is a function of the predictability of the system, the proper use of locking mechanisms, and avoidance of deadlock — issues that are readily studied in the design of real-time systems (Laplante, 1997).

8.6.2 EXECUTION TIME ESTIMATION

As discussed above, CPU utilization is based on the execution time estimates for each procedure. But how should execution time be estimated or measured? The best method for measuring the execution time of any piece of code is to use a logic analyzer, as previously described. One advantage of this approach is that hardware latencies and other delays not due simply to instruction execution times are taken into account. The drawback in using the logic analyzer is that the system must be completely (or partially) coded and the target hardware available. Hence, the logic analyzer is usually employed only in the late stages of coding, during testing, and especially during system integration.

When a logic analyzer is not available, the code execution time can be estimated by examining the compiler output and counting macroinstructions. This technique also requires that the code be written, an approximation of the final code exists, or similar systems are available for analysis. The approach simply involves tracing the worst-case path through the code, counting the macroinstructions along the way, and adding their execution times. These can be found in the manufacturer's specifications or through measurement with a logic analyzer.

Another accurate method of code execution timing uses the system clock, which is read before and after executing code. The time difference can then be measured to determine the actual time of execution. This technique, however, is only viable when the code to be timed is large relative to the code that reads the clock.

8.6.3 SCALED NUMBERS

In virtually all computers, integer operations are faster than floating-point ones. This fact can be exploited by converting floating-point algorithms into scaled integer algorithms. In these so-called scaled numbers, the least significant bit (LSB) of an integer variable is assigned a real number scale factor. Scaled numbers can be added and subtracted together and multiplied and divided by a constant (but not another scaled number). The results are converted to floating point only at the last step — hence saving considerable time.

For example, suppose an analog-to-digital converter is converting optical information from a camera into electrical impulses corresponding to gray values and storing them in a 16-bit unsigned integer. In this case, floating-point gray values in the range $0 \leq x < 1$ indicate that the pixel is black (1) or white (0). If the LSB of the 16-bit integer has a value of 2^{-16} , then the most significant bit is $1 - 2^{-16} = 0.999984741$.



FIGURE 8.5 A 16-bit BAM word.

A common practice is to quickly convert the integer number into its floating-point equivalent by $xf = x \cdot 0.0000153$ and then proceed to use it in calculations directly with other converted numbers; for example, $diff = xf - zf$ where zf is a similarly converted floating-point number. Instead, the calculation can be performed in integer form first and then converted to floating point: $diff = (x - z) \cdot 0.0000153$.

For applications involving the manipulation, addition, and subtraction of large quantities of floating-point pixel data, scaled numbers can introduce significant savings. Note, however, that multiplication and division (by any number other than 0 or 1) cannot be performed on a scaled number, as those operations change the scale factor. Finally, accuracy is generally sacrificed by excessive use of scaled numbers.

8.6.4 BINARY ANGULAR MEASURE

Another type of scaled number is based on the fact that adding 180° to any angle is analogous to taking its two's complement. This technique, called binary angular measurement (BAM), works as follows. Consider the LSB of an n -bit word to be $2^{n-1} \cdot 180^\circ$ with most significant bit (MSB) = 180° . The range of any angle θ represented this way is $0 \leq \theta \leq 360 - 180 \cdot 2^{-(n-1)^\circ}$. A 16-bit BAM word is shown in Figure 8.5. For more accuracy, BAM can be extended to two more words.

BAM is frequently used in navigation software and imaging systems using line manipulation algorithms such as ray tracing. In addition, it works well in conjunction with digitizing imaging devices.

8.6.5 LOOK-UP TABLES

Another variation of the scaled number concept uses a stored table of function values at fixed intervals. Such a table, called a look-up table, allows for the computation of continuous functions using mostly fixed-point arithmetic.

Let $f(x)$ be a continuous real function and let Δx be the interval size. Suppose it is desired to store n values of f over the range $[x_0, x_0 + (n-1)\Delta x]$ in an array of scaled integers. Values for the derivative, f' , may also be stored in the table. The choice of Δx represents a trade-off between the size of the table and the desired resolution of the function. A generic look-up table is given in Table 8.2.

It is well known that the table can be used for the interpolation of $x < \hat{x} < x + \Delta x$ by the formula

$$f(\hat{x}) = f(x) + (\hat{x} - x) \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{8.2}$$

This calculation is done using integer instructions, except for the final multiplication, by the factor $(\hat{x} - x) / \Delta x$ and conversion to floating point. As a bonus, the look-up

TABLE 8.2
Generic Function Look-Up Table

x	$f(x)$
x_0	$f(x_0)$
$x_0 + \Delta x$	$f(x_0 + \Delta x)$
$x_0 + 2\Delta x$	$f(x_0 + 2\Delta x)$
\dots	\dots
$x_0 + (n - 1)\Delta x$	$f(x_0 + (n - 1)\Delta x)$

table has a faster execution time if \hat{x} happens to be one of the table values. If $f'(x)$ is also stored in the table, then the look-up formula becomes

$$f(\hat{x}) = f(x) + (\hat{x} - x)f'(x) \tag{8.3}$$

This improves the execution time of the interpolation somewhat.

The main advantage in using look-up tables, of course, is speed. If a table value is found and no interpolation is needed, then the algorithm is much faster than the corresponding series expansion. In addition, even if interpolation is necessary, the algorithm is interruptible and hence helps improve performance, compared to a series expansion.

Look-up tables are widely used in the implementation of continuous functions such as the exponential sine, cosine, and tangent functions; their inverses; and so on. For example, consider the combined look-up table for sine and cosine using radian measure, shown in Table 8.3.

Because these trigonometric functions and exponentials are used frequently in conjunction with the discrete Fourier transform (DFT) and discrete cosine transform (DCT), look-up tables can provide considerable savings in imaging applications.

TABLE 8.3
Look-Up Table for Trigonometric Functions

Angle (rads)	Cosine	Sine	Angle (rads)	Cosine	Sine
0.000	1.000	0.000	6.981	0.766	0.643
0.698	0.766	0.643	7.679	0.174	0.985
1.396	0.174	0.985	8.378	-0.500	0.866
2.094	-0.500	0.866	9.076	-0.940	0.342
2.793	-0.940	0.342	9.774	-0.940	-0.342
3.491	-0.940	-0.342	10.472	-0.500	-0.866
4.189	-0.500	-0.866	11.170	0.174	-0.985
4.887	0.174	-0.985	11.868	0.766	-0.643
5.585	0.766	-0.643	12.566	1.000	0.000
6.283	1.000	0.000			

8.6.6 IMPRECISE COMPUTATION

In cases where software routines are needed to provide mathematical support (in the absence of firmware support or digital signal processing (DSP) coprocessors), complex algorithms are often employed to produce the desired calculation. For example, a Taylor series expansion (perhaps using look-up tables for function derivatives) can be terminated early, at a loss of accuracy but with improved performance. Techniques involving early truncation of a series in order to meet deadlines are often called imprecise computation. Imprecise computation (also called approximate reasoning) is often difficult to apply, however, because it is not always easy to determine the processing that can be discarded or its cost.

A variation on imprecise computation occurs when applying an algorithm on a compressed version of the image rather than the image itself in order to save computation time. As an illustration of the approach, suppose a 256-gray-level document image has been captured and it is desired to use a linear matched filter to identify characters within the image. Owing to the binary-like nature of most documents, processing could commence by first thresholding the document to a binary image and then applying a much faster, sparse hit-or-miss matched filter.

8.6.7 OPTIMIZING MEMORY USAGE

In modern computer architectures, memory constraints are not as troublesome as they once were. Nevertheless, in embedded applications or in legacy systems (those that are being reused), often the imaging engineer is faced with restrictions on the amount of memory available for program storage or for scratch-pad calculations, dynamic allocation, and so on. Since there is a fundamental trade-off between memory usage and CPU utilization (with rare exceptions), when it is desired to optimize for memory usage, it is necessary to trade performance to save memory. For example, in the trigonometric function just discussed, using quadrant identities can reduce the need for a large look-up table. The additional logic needed, however, represents a small run-time penalty.

Finally, it is important to match the image processing algorithms to the underlying architecture. In the case of parallel architectures, this is a specialized problem that is out of the scope of this text. But in the case of the von Neumann architecture, for example, it is helpful to recognize the effects of such features as cache size and pipeline characteristics. In the case of cache size, for example, the algorithm should be chosen to optimize the cache hit ratio — the percentage of time that data are found in the cache. In the case of pipelines, increasing the code locality of reference (the tendency of the code to execute sequential instructions) can reduce the amount of deleterious pipeline flushing.

8.7 A SOFTWARE REENGINEERING PROCESS MODEL

Many commercial imaging systems are legacy systems; that is, they constitute the next generation of an existing system. Others borrow code from related systems. In any case, most systems need to have a long shelf life so that development costs can

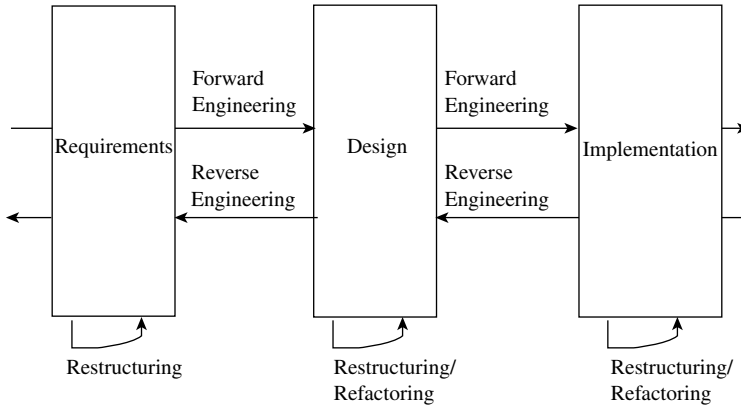


FIGURE 8.6 A reverse engineering process model.

be recouped. Maintaining a system over a long period usually requires some form of reengineering, that is, a reverse flow through the software life cycle.

Generically, reverse engineering is the process of analyzing a subject system to identify its components. Reengineering is sometimes called renovation or reclamation. While there are negative connotations to reverse engineering, as in theft of a design, in some form it is essential for the improvement of the design or implementation, or for recovery of documentation in the case of a system that may have been acquired legitimately from a third party.

Figure 8.6 is a graphical representation of a reengineering process. The forward engineering flow represents a simple, three-phase waterfall model — requirements, design, and implementation.

Documentation recovery or redocumentation is the creation or revision of documentation from a given system, including requirements and design documentation discovery. The need for redocumentation arises when there is poor or missing documentation for any of a number of reasons.

Design recovery is a subset of reverse engineering that recreates the design from code, existing documentation, personal insight, interviews with developers, and general knowledge. Again, the need for this arises in the case of poorly documented design, missing documentation, acquisition of a product from a company with inferior software engineering practices, and so on.

Restructuring is the transformation of one representation to another. In the case of code refactoring, the code is transformed so that the behavior is preserved. In design refactoring, the design is reengineered.

8.8 A MAINTENANCE PROCESS MODEL

Of all the phases, perhaps the maintenance model is the least understood (see Figure 8.7). The maintenance phase generally consists of a series of reengineering processes to prolong the life of the system. There are three types of maintenance:

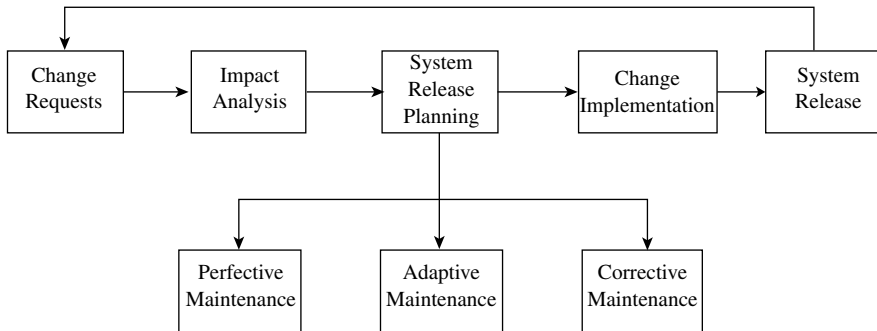


FIGURE 8.7 A maintenance process model. (Adapted from Sommerville, I., *Software Engineering*, 5th ed., Addison-Wesley, New York, 2000.)

1. Adaptive — Changes that result from external changes to which the system must respond
2. Corrective — Changes that involve maintenance to correct errors
3. Perfective — All other maintenance, including enhancements, documentation changes, efficiency improvements, and so on

A widely adopted maintenance model illustrates the relationship between these various forms of maintenance (Sommerville, 2000). The model starts with the generation of change requests by either the customer, management, or engineering team. An impact analysis is performed. This analysis involves studying the effect of the change on related components and systems and includes the development of a budget.

System release planning involves determination of whether the change is perfective, adaptive, or corrective. The nature of the change is crucial in determining whether the release needs to be made to all customers and specific customers, whether the release is going to be immediate or included in the next version, and so on. Finally, the change is implemented (in fact, invoking a mini-software life cycle process from concept to acceptance testing), followed by the official release of the new version.

8.9 SOFTWARE REUSE

Pure software reuse is a highly sought prize in software engineering. It is clearly desirable to have a collection of mix-and-match, validated software components that could easily be pulled off the shelf for customized software applications. However, software reuse is virtually an exploitation of hard-learned experience. Even if software modules are not being explicitly reused, the lessons learned from previous but similar software projects should be carried forward.

Moreover, current practice shows that the benefits of software reuse are very slow to be realized. There are several reasons for this. First, even when a mandate is given to adopt software reuse technology, the effort needed to make it happen far exceeds the benefits in the short run. Second, pressures to deliver projects can quickly

put software reuse efforts on the back burner. Finally, it takes great discipline and a change in thinking on the part of most software engineers to apply software reuse principles effectively.

Unfortunately, even if rigorous reuse policies and practices are adopted, an initial decrease in productivity is to be expected because of the additional activities required for software reuse. For this reason and the previously mentioned drawbacks, a long-term, high-level commitment to the adoption of software reuse policies and practices is needed.

Software components that developers tend to reuse the most are small components, such as abstract data types, utility routines (e.g., memory and I/O handling functions), or class libraries. However, small-component reuse produces minimal savings because such components represent only a small percentage of the final product, perhaps 20% of the entire application (see Pareto's principle below). Another 15% is usually application specific, and therefore not normally reusable; this leaves 65% that is domain specific (Poulin, 1997).

Most of the cost savings can then be expected by reusing domain-specific models; in this case, the domain is image processing. To reuse domain-specific logic, however, developers must clearly separate domain logic from that of the application. They must also clearly distinguish logic that is domain independent.

Developers can achieve this separation by designing applications so that their class structure exhibits high cohesion and low coupling. Unfortunately, doing so is easier said than done, and reusable software design is especially difficult (Neill and Gill, 2003b).

Therefore, the best way to begin a program of software reuse is to start small and learn by doing. Try identifying several small software modules that are good candidates for reuse and focus on preparing these modules for that reuse.

8.9.1 WHEN NOT TO REUSE

It is sometimes desirable to plan not to reuse certain codes. For example, throwaway prototypes are intentionally not to be reused. In other cases, it may not be desirable to try to reuse code that is of limited value. For example, a set of utilities intended for very specific hardware or that serve a very specialized function are probably not worth engineering for reuse when the hardware changes or becomes obsolete.

In any event, reuse of code that was not designed and coded for reuse can create many problems. For example, when a "quick-and-dirty program" becomes a widely used tool, it can present a maintenance nightmare.

8.9.2 ACHIEVING REUSE

One approach to reuse involves building program libraries. Program libraries consist of components that have been validated or certified for "error-free" use.

Many of the beneficial software engineering practices already discussed can lead to a high level of reuse. For example, the benefits of Parnas partitioning in this regard has been mentioned. In addition, the appropriate coding standards can lead to more readable code, which can enhance the potential for reuse. Finally, in object-oriented systems, use of design patterns following the object-oriented design principles pre-

viously described (open–closed principle, once and only once, dependency inversion principle, Liskov substitution principle) and the software engineering principle of generality foster reusability.

8.9.2.1 In Procedural Languages

Another technique that has been used in building program libraries involves domain analysis. Domain analysis views software codes as functions with an input domain and output range, based on the range of their inputs.

The approach is as follows. In a set-theoretic way, define the input and output (I/O) domains for each module to be added to the program library. Then determine the I/O dependencies between each module in the library and any candidate module to be added to the library. The existence of such dependencies determines the compatibility of the candidate module with the existing library modules. Of course, it is assumed that each candidate module has been validated and is fully tested at the module level.

For example, consider a program library that consists of trusted code unit A. Code unit A has an input domain of A_I and an output range of A_O . Now consider a new candidate code unit B, which has already been unit tested. Code unit B has an input domain of B_I and an output range of B_O . “Domains” and “ranges” mean the set of I/O variables of these modules, and their ranges.

Now, if the output range of A, that is, the variables that A could change, does not intersect with the input range of B, and vice versa, then module B may be added to the program library without any further interdependence or compatibility testing. If the input range of B and the output range of A overlap, then interdependencies and compatibility need to be tested before adding A to the library. Formally,

$$\begin{array}{ll} \text{If } A_O \cap B_I = \phi \text{ and } B_O \cap A_I = \phi & \text{then add A to the library} \\ \text{Else} & \text{test further before adding} \end{array} \quad (8.4)$$

As additional modules or code are added to the library, interdependence testing must be done for all modules in the library. For example, if A and B are trusted software in the library and module C is a candidate for the library, it now must be tested against A and B before adding it. Formally,

$$\begin{array}{ll} \text{If } A_O \cap C_I = \phi \text{ and } B_O \cap C_I = \phi \text{ and} \\ C_O \cap A_I = \phi \text{ and } C_O \cap B_I = \phi & \text{then add C to the library} \quad (8.5) \\ \text{Else} & \text{test further before adding} \end{array}$$

It is easy to see that the level of effort grows rapidly as new code is added to the trusted program library.

8.9.2.2 In Object-Oriented Languages

In this situation, the key is to employ the protected variation (Parnas partitioning) principle by identifying those design aspects that are likely to change and build a stable interface around them. Design patterns can loosen the binding between program components, enabling certain types of program evolution to occur with minimal changes to the program itself. However, to make good use of design patterns, the application's design process must undergo a couple of iterations over the project life cycle. Because time to market is often the foremost priority, developers might not have time to create a flexible design. Even if the application does not have the required flexibility, however, introducing that flexibility is possible by refactoring the application (Neill and Gill, 2003b).

8.9.2.3 Pareto's Principle

Pareto was a late-19th- and early-20th-century Italian mathematician and economist who was interested in the laws of chance. His observations can be applied in several ways related to software reuse and engineering. For example, Pareto's principle might suggest that:

- 20% of the code contributes to 80% of the cost of software development.
- 20% of the code contributes 80% of the errors.
- 20% of the errors accounts for 80% of the cost to fix.
- 20% of the modules consumes 80% of the execution time.

The percentages are, of course, arbitrary. But these observations provide insight into how to approach software reuse, testing, and effort planning. For example, it would be helpful to identify the 20% of software that is the most expensive in order to develop and plan to reuse those software. The other 80%, the ones that are relatively easy to develop, might not be prime candidates for reuse. Checkpoints and software black boxes can help to collect code unit execution frequency to identify the high-use code.

8.10 THE SECOND SYSTEM EFFECT

The second system effect, first characterized by Brooks (1995), explains why software maintenance for legacy systems presents such challenges. This phenomenon is discussed in *The Mythical Man-Month*, a series of essays on software project management by Brooks that was first published in the early 1980s. The essays are still relevant today. Brooks notes that "second systems," or the next generation of a delivered system, tend to be overengineered. That is, there is a tendency to carry over and refine techniques whose existence has been made obsolete by changes in basic system assumptions. Doing so tends to make these systems hard to maintain, unwieldy, and unreliable.

Consider, for example, an imaging system that was developed in the 1970s for hardware that is no longer available. In a second system, the underlying hardware may have been modernized. Hence, carrying over old design decisions can be disastrous. Imaging systems tend to be based on carryover software, often originally written in Fortran, C, assembly language, or even BASIC. In some cases, C code is simply “objectified” by wrapping the C code in such a way that it can be compiled as C++ code. This does not realize the benefits of object orientation (see Chapter 6).

Brooks (1995) suggests that the way to avoid this effect is to insist on a project leader who has had experience with at least two systems. In this recommendation, Brooks recognizes that software houses tend to assign new software engineers to maintain old legacy systems, while the more senior engineers are assigned to new software development. While new projects may be more glamorous, younger engineers may not have the confidence or experience to challenge bad design decisions on a legacy system. Hence, it is probably better to have a combination of experience and youth assigned to both new and legacy system software development.

8.11 CODE AND PROGRAM MAINTENANCE

A source code control system places strict control over the access of files related to a software project to prevent conflicts such as multiple engineers editing a file simultaneously. In addition, source code control keeps an audit trail of changes and sets file access permissions. This kind of control is crucial when developing large programs for which many people will have access. Strict version control is quite important when handling programs consisting of a large number of files or when more than one individual is working on the same project.

For example, it would be disastrous if two programmers decided to modify the same source code file simultaneously — one set of changes would be lost. Similarly, suppose that a project consists of dozens of source files along with header and include files. If a header file is changed, every single source file using that header file must be recompiled or very difficult-to-find bugs will be introduced.

8.12 EXERCISES

- 8.1 Why is the maintenance aspect of the software life cycle perhaps the least well understood?
- 8.2 Derive the look-up table for the tangent function in increments of 1° . Be sure to take advantage of symmetry.
- 8.3 Identify the likely library functionality for the visual inspection system. In other words, which modules or objects would be likely to be reused in other, similar systems? Which code units would be likely to be reused in other, dissimilar systems?
- 8.4 What are the advantages and disadvantages of writing a BAM object class in an object-oriented language?

- 8.5 Should software reuse be an *ad hoc*/implicit requirement in a project, or an explicit goal of the project?
- 8.6 When software reuse is not possible, on what other goals can a software project focus?
- 8.7 Can the tools available impact the software project and the code quality? How?

9 Management of Software Projects

Adding manpower to a late project makes it later.

Frederick Brooks

9.1 WHY SOFTWARE PROJECT MANAGEMENT?

It is a commonly held notion that the vast majority of software projects are delivered late and over budget. Surprisingly, however, some recent research seems to suggest that the problem might not be as bad as once thought, at least for one group.

For example, in the previously referenced survey (see Chapter 3), about 42% of the respondents thought that project costs were within budget estimates, and 35% believed that their projects were not. Roughly 45% of respondents thought that projects were finished on time, but an equal number thought that their projects overran their deadlines. About 15% of the respondents thought that “project goals were achieved earlier than predicted,” while 60% disagreed with this statement; 41% of the respondents thought that the project could have been completed faster, but at the expense of quality. About 40% disagreed with that statement (Laplante et al., 2002e).

Whether this apparent mitigation of widely held perceptions about software is due to improved project management is unknown — many other studies still show that cost overruns and project delays are major problems. But it is certainly the case that software development costs can better be controlled through the application of improved management practices.

Many companies have tried to improve their software project management practices through the adoption of one or another more “fashionable” technique. But doing so may ultimately lead to failure and abandonment of that technique. The truth is that there is no magic elixir, no silver bullet for improving project management practices. Bad management habits take a long time to develop and even longer to break. Bad management practices are so pervasive, in fact, that they have been widely described as “antipatterns,” that is, apparent solutions with negative consequences.

There are other reasons to study software project management separately. Although there are many similarities between software project management and other types of project management, software project management has several unique aspects. Moreover, the skills that make a good software engineer are not the skills needed to be a good project manager. But many companies promote software engineers to project management positions without the necessary training. Finally, it is usually too late when it is discovered that a software project is headed for

disaster, in what has been described as a “death march.” Unfortunately, last-minute heroics are a poor substitute for careful planning and execution throughout the software life cycle, and this is particularly true in highly complex imaging systems.

9.2 SOFTWARE PROJECT MANAGEMENT THEMES

This chapter will emphasize several themes. First, the software model or project methodology used is not necessarily as important as having the resolve to select an appropriate one and staying with it.

The second theme is that in project management, of any kind, the use of good interpersonal skills such as negotiating, communicating, and expectation setting is critical.

The final theme for this chapter is that using quantitative tools such as metrics and process monitoring models is essential to keeping a scientific approach to software project management.

9.3 GENERAL PROJECT MANAGEMENT BASICS

A project is a set of tasks with a defined beginning and end. Without a defined beginning, there is no way to begin measuring progress. Without a defined end, there is no way to determine if the project has been completed, and thus progress measurement is again impossible. The simple project definition is recursive in that any project probably consists of more than one subproject.

Generally, a project that involves only a single person is uninteresting. This is the case because without the complexity introduced by the interactions of other human beings, task completion is really just a simple exercise in separation of concerns and self-discipline. Separation of concerns implies that one of the objectives in project management is to subdivide the project in a meaningful way.

Many software engineering project management activities are different from those needed for software project management. These are summarized in Table 9.1.

This framework provides a model for discussion for the rest of this chapter.

9.3.1 WHAT DOES THE PROJECT MANAGER CONTROL?

A project manager may have one of the following three elements under his control:

1. Resources — Such as equipment and team members, and money to buy more of them. Almost always, however, there are financial limitations, and generally these are fixed prior to the start of the project.
2. Schedule — The manager should have some control over the schedule. Even if the delivery date of the product is hard, there should be some flexibility for internal manipulation of the schedule that does not change the delivery date.
3. Functionality — The product functionality may or may not be controllable. Often when negotiating a project, the project manager cannot increase

TABLE 9.1
Software Process Planning vs. Project Planning

Software Engineering Planning Activities	Software Project Management Planning Activities
Determine tasks to be done	Determine skills needed for the task
Establish task precedence	Establish project schedule
Determine level of effort in person-months	Determine cost of effort
Determine technical approach to solving problem	Determine managerial approach to monitoring project status
Select analysis and design tools	Select planning tools
Determine technical risks	Determine management risks
Determine process model	Determine process model
Update plans when the requirements or development environment change	Update plans when the managerial conditions and environment change

Source: Adapted from Thayer, R.H., *Computer*, 35, 4, 68–73, 2002.

costs, but can decrease product functionality in order to meet a customer's budget or deal with an unforeseen problem.

Generally, in terms of controlling the project, the manager must understand the project goals and objectives. Next, the manager needs to understand the constraints imposed on the resources. These include cost and time limitations, performance constraints, and available staff resources. Next, the manager develops a plan that enables her to meet the objectives within the given constraints. Of course, monitoring and control mechanisms must be in place, including metrics. The manager should be prepared to modify the plan as it progresses. These modifications need to be made to the plan, and then team members can make adjustments as necessary and appropriate. Finally, a calm, productive, and positive environment is desirable to maximize the performance of the team and to keep the customer (whether internal, like senior management, or a client) happy and confident that the job is being done right.

9.4 SOFTWARE PROJECT MANAGEMENT

Throughout the text, various properties of software have been discussed. What has been infrequently noted, however, is that the things that make software different from other types of endeavors also make it harder to manage the software process. For example, unlike hardware, to a large extent, software designers build software knowing that it will have to change. Hence the designer has to think about both the design and redesign. That adds a level of complexity.

Of course, software development involves novelty, which introduces uncertainty. It can be argued that there is a higher degree of novelty in software than in other forms of engineering. Why else are there so few reusable and off-the-shelf software components?

The uniqueness of software project management is intensified by a number of specialized activities. These include:

- The process of software development itself
- The complex software maintenance process
- The unique and not well-evolved process of verification and validation
- The interplay of hardware and software that faces the systems engineer

The uniqueness of software and these activities leads to a number of problems, which introduce risk. Many of these have been discussed already, but a more complete list includes:

- Incomplete and imprecise specifications
- Difficulties in modeling highly complex systems
- Uncertainties in allocating functionality to software or hardware and the subsequent turf battles
- Uncertainties in cost and resource estimation
- Difficulties with progress monitoring
- Rapid changes in software technology and the underlying hardware technology
- Measuring and predicting the reliability of the software
- Problems with interface definition
- Problems that are encountered during software–software or hardware–software integration
- Unrealistic schedules and budgets
- Gold plating, which is perfecting functionality that needs no further perfection
- Shortfalls in externally furnished components
- Real-time performance shortfalls
- Trying to strain the limits of computer science capabilities

The following sections describe some approaches to attacking these risk factors.

9.5 MANAGING AND MITIGATING RISKS

The role of the project manager includes managing and mitigating the risks previously introduced. From the list, it can be seen that many of the risks can be managed through close attention to the requirements specification and design processes. Prototyping (especially throwaway) is also an important tool in mitigating risk, as was described in Chapter 3. Code reviews and walk-throughs can play an important role as well. Finally, judicious and vigorous testing can reduce or eliminate many of these risks.

The remainder of the risks need to be managed, that is, closely monitored and controlled where possible. Table 9.2 summarizes the risk factors and possible approaches to risk management and mitigation.

TABLE 9.2
Various Project Risk Sources and Possible Management, Measurement, Elimination, and Mitigation Techniques

Risk Factor	Possible Management/ Mitigation Approach
Incomplete and imprecise specifications	Prototyping Requirements reviews Formal methods
Difficulties in modeling highly complex systems	Prototyping Testing
Uncertainties in allocating functionality to software or hardware and the subsequent turf battles	Prototyping Requirements reviews
Uncertainties in cost and resource estimation	Project management Metrics
Difficulties with progress monitoring	Project management Monitoring tools Metrics
Rapid changes in software technology and the underlying hardware technology	Prototyping Testing
Measuring and predicting the reliability of the software	Metrics Testing
Problems with interface definition	Prototyping
Problems that are encountered during software–software or hardware–software integration	Prototyping Testing
Unrealistic schedules and budgets	Project management Monitoring tools Metrics
Gold plating	Code audits and walk-throughs
Shortfalls in externally furnished components	Testing
Real-time performance shortfalls	Prototyping Testing
Trying to strain the limits of computer science capabilities	Code audits and walk-throughs Testing

9.6 PERSONNEL MANAGEMENT

One of the most important but frequently overlooked aspects of managing software projects is managing people. Of course it is well known that the success of a project is directly related to the quality of talent employed and, more importantly, the manner in which the talent is deployed on the project (MacDonald, 1998). But too frequently project managers* view themselves as technical managers only, forgetting that human nature enters into technical situations.

* *Manager* is a general term for anyone responsible for one or more other persons developing, managing, installing, supporting, or maintaining systems and software. Typical other titles include software project manager, technical lead, senior developer, and so on.

TABLE 9.3
Four Possible Combinations of Good/Bad Management
and Good/Bad Team Chemistry

	Good Team Chemistry	Bad Team Chemistry
Good Management	Likely success	Possible success
Bad Management	Unlikely success	Unlikely success

The special challenges of developing software are imposed on top of the already daunting challenge of managing human teams. Some people might consider the aspect of human resource management insignificant if the project team has enough technical skill. This is not generally true.

The key problem in most cases is that the chemistry of the team makes it impossible for the manager to overcome the constraints — even with good people. Table 9.3 illustrates the four possible cases of good/bad management and good/bad team chemistry. In the case where both are good, the likelihood of project success (which itself must be carefully defined) is high. In the case of bad management, success is unlikely — even with good team chemistry — because bad management will eventually erode morale. But it is where the team chemistry is bad that good management can possibly lead to success.

9.6.1 THE *N*-BODY PROBLEM

One reason why creating a good team chemistry is so hard is that the number of working relationships grows as a polynomial function of *n*, the number of people on the team. This might be whimsically referred to as the *n*-body problem and is depicted in Figure 9.1.

In fact, it can easily be shown by induction that for *n* people on a team, there are $n(n - 1)/2$ possible working relationships, any of which can sour. Furthermore,

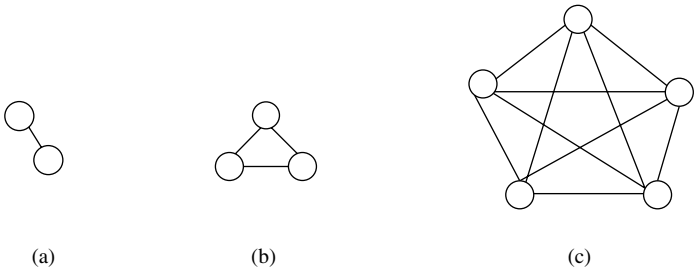


FIGURE 9.1 The *n*-body problem as the growth of working relationships to be managed as a function of number of team members. (a) Two persons with one relationship. (b) Three persons with three relationships. (c) Five persons with 10 relationships. (From Laplante, P., *IT Prof.*, Jan./Feb., 46–50, 2003. With permission from the Institute of Electrical and Electronics Engineers.)

a working relationship is not transitive. So, for example, Roger may work well with Mary, and Mary with Sue, but Roger and Sue may not work well together. Finally, complicating these interactions are intercultural differences and outsourcing of project components. All these aspects must be considered when building and managing teams, planning projects, and dealing with difficult personal situations (MacDonald, 1998). “Too many chefs spoil the broth, indeed,” or to paraphrase Brooks, “Adding manpower to a late software project makes it later” (Brooks, 1995).

9.6.2 SOME APPROACHES TO LEADING TEAMS

There are almost as many management styles as there are people. But traditionally, a small collection of paradigms can be used to more or less describe the management style of an individual. Understanding these basic approaches can be helpful in understanding the motivations of customer, supervisors, and subordinates. Much of the foregoing discussion is adapted from Laplante (2003a).

9.6.2.1 Theory X

Theory X is perhaps the oldest management style, and it is closely related to the hierarchical, command-and-control model used by military organizations. According to Theory X, this approach is necessary because most people inherently dislike work and will avoid it if they can. Hence, managers should coerce, control, direct, and threaten their workers in order to get the most out of them. Generally, the theory holds that most people prefer to be told what to do than to have to decide for themselves. A typical statement by Theory X managers might be “People only do what you audit.”

9.6.2.2 Theory Y

As opposed to Theory X, Theory Y holds that work is a natural and desirable activity. Hence, external control and threats are not needed to guide the organization. In fact, the level of commitment is based on the clarity and desirability of the goals set for the group. Finally, most individuals actually seek responsibility and do not shirk it, as Theory X proposes.

A Theory Y manager simply needs to provide the resources, articulate the goals, and leave the team alone. This does not always work, of course, because some individuals do need more supervision than others.

9.6.2.3 Theory Z

Theory Z is based on the philosophy that employees will stay for life with a single employer, which results in strong bonding to the corporation and subordination of individual identity to that of the company. Theory Z organizations have implicit, not explicit, control mechanisms such as peer and group pressure. The norms of the particular corporate culture also provide additional implicit controls. Japanese companies are well known for their collective decision making and responsibility at all levels.

Theory Z management emphasizes a high degree of cross-functionality for all of its workers. Specialization is discouraged. Most top Japanese managers have worked in all aspects of their business from the production floor to sales and marketing. This is also true within functional groups. For example, assemblers will be cross-trained to operate any machine on the assembly floor. Theory Z employers are notoriously slow in giving promotions, and most Japanese CEOs are over 50.

9.6.2.4 Theory W

Theory W is a software project management paradigm developed by Boehm and Ross (1989) that focuses on the following for each project:

- Establishing a set of win–win preconditions
- Structuring a win–win software process
- Structuring a win–win software product

Establishing a set of win–win preconditions recognizes that the best working relationships are those in which everyone “wins.” Zero-sum or win–lose or lose–win situations can leave one or both parties bitter. Win–win solutions can be sought as follows.

First, recognize that everyone wants to win. Then understand what constitutes a winning situation for each individual. Money, power, and recognition contribute to winning conditions for most people, but there are other, more subtle conditions, such as job satisfaction, a feeling of belonging, and moral fulfillment.

Next, establish reasonable expectations. The importance of setting reasonable and mutually fulfilling expectations in every aspect of human relations cannot be overemphasized. Then ensure that people’s task assignments will match their win conditions.

Finally, provide an environment that supports the fulfillment of people’s win conditions. This can take a variety of forms but might include such things as financial incentives, group activities, and communication sessions to head off problems.

Structuring a win–win software process means setting up a system that will lead to success. This includes establishing a realistic process plan based on some standard methodology. This methodology may be internal and company-wide or off-the-shelf.

It is also important to use the project/management plan to control the project. It has been said, “The plan is nothing, planning is everything.” Too often, managers develop a project plan to sell the job to senior management or the customer, and then throw away the plan once it has been sold. Use and maintain the project plan throughout the life of the project.

Project managers need to monitor the risks that have been described, as they can lead to win–lose or lose–lose situations. Thus, risks should be identified and eliminated at the earliest opportunity.

Keeping people involved is essential. It helps team members feel a part of the project and improves communications. Besides, listening to team members can highlight great ideas.

Structuring a win–win software product refers to the process of specification writing. Matching the users’ and maintainers’ win conditions is the key. This process includes careful expectation setting.

9.6.3 PRINCIPLE-CENTERED LEADERSHIP

All of the management approaches discussed thus far focus on organizational frameworks for management. Principle-centered leadership focuses on the behavior of the manager as an agent for change (Covey, 1992). Some management theorists hold that motivating team members by example and leadership, and not through hierarchical application of authority, is much more effective (manage things, but lead people). A key concept in principle-centered leadership is that the best managers are leaders and that the only way to affect change is by the managers changing themselves first.

Principle-centered leadership recognizes that principles are more important than values. Values are society based and can change over time and differ from culture to culture. Principles are more universal, more lasting. Think of some of the old principles, like the golden rule. These are timeless and transcend culture.

Another example of a timeless principle is “you reap what you sow.” This holds true when dealing with people. Treat people with respect, and you will be respected. Failure to respect people leads to disrespect.

In fact, there is a great deal of similarity in principle-centered leadership and Theory W, with principle-centered leadership being much more generic.

9.6.3.1 Management by Sight

Management by sight, also known as management by walking around, is not really a full-bodied management approach, but rather a substrategy for the approaches already discussed. This approach is people oriented because it requires the manager to be very visible and to interact with staff. Interacting with staff at all levels is a good way for managers to collect important information about the project and people in their care.

Really, management by sight is obvious. The manager uses observation and visibility to provide leadership, to monitor the situation, and even to control when necessary. In general, it is advisable to incorporate this strategy into any management approach.

9.6.3.2 Management by Objectives

Management by objectives (MBO) is another substrategy that can be used in conjunction with any other approach to management. MBO involves managers and subordinates jointly setting carefully structured objectives with measurable outcomes and rewards.

Coupled with periodic reviews to measure progress, MBO has the effect of a “carrot-and-stick” reinforcement of desired performance.

For example, a manager may contract with a team member responsible for writing a section of the software design description that she completes her task by a certain date (provisos can be made for various, inevitable distractions that will appear). In return, time off might be granted for meeting the goal — more time off for early completion. The scenario becomes somewhat more complex when the other 10 things that the team member is responsible for are factored in (for example, producing other reports, attending meetings, working on another project simultaneously). The process tracking tools to be discussed shortly are very helpful in this case.

The keys to MBO are setting reasonable but aggressive goals and having a clear means of measuring success.

9.6.4 DEALING WITH DIFFICULT PEOPLE

One challenge facing every manager is dealing with difficult people, whether they are subordinates, peers, or superiors. Dealing with difficult people is largely personality based. In any case, it is important that the manager does not form an opinion about a person or situation too soon. Never attribute some behavior to malice when a misunderstanding could be the reason. Almost without exception, taking the time to investigate an issue and to think about it calmly is superior to reacting spontaneously or emotionally.

Whatever management style is employed, the manager should make sure that the focus is on issues and not people. Managers should avoid the use of accusatory phrases such as telling someone that he is incompetent. The manager should instead focus on her feelings about the situation. Make sure that all sides of the story are listened to when arbitrating a dispute — before forming a plan of resolution. It is often said that there are three sides to an issue, the sides of the two opponents and the truth, which is somewhere in between. While this is just a cliché, there is much truth to it.

The manager should always work to set or clarify expectations. Management failures, parental failures, marital failures, and the like are generally caused by a lack of clear expectations. The manager should set expectations early in the process, making sure that everyone understands them. She should continue to monitor the expectations and refine them if necessary.

Good team chemistry can be fostered through mentoring, and most of the best managers you probably know also fit the description of a mentor. The behaviors already described are generally those of someone who has a mentoring personality.

Finally, the manager should be an optimist. No one chooses to fail. In fact, MacDonald (1998) notes that most programmers are optimists. The manager should always give people the benefit of the doubt and work with them.

9.7 ASSESSMENT OF PROJECT PERSONNEL

Managers use a variety of testing instruments to evaluate job applicants for skills, knowledge, and even personality. Candidate exams range widely from programming tests, exam-like tests, situation analysis, and “trivia” tests that purport to test critical thinking skills. Some companies do not test at all.

9.7.1 SKILLS TESTING

Skills tests can be used to assess a variety of proficiencies. For those who might work on imaging software systems, the proficiencies should include:

- Knowledge of imaging algorithms and underlying science
- Knowledge of commercial imaging hardware

- Proficiency in one or more relevant programming languages, such as C, C++, Java, assembly language, or Fortran
- Proficiency with commercial computer-aided software engineering and programming environment tools

In some companies, the test is written, administered, and graded by the HR department. In other companies, the technical line managers administer the test. Elsewhere, the tests are organized and delivered by “peer” staff — those who would potentially work with the candidate and who seem to be in the best position to determine which technical skills are relevant. Finally, some companies outsource this kind of testing.

During the dot.com era, a number of websites emerged that provided on-line testing. These companies wrote and managed a test data bank and also provided incentives for candidates to visit the site and take the test. For a fee, subscribing companies could access the database and interview those candidates who had achieved a certain score or answered certain questions correctly on the test.

The real question, however, is “Are any of these tests a true predictor of job performance?” There seems to be no consensus, even among those who study human performance testing.

One thing a company could do to test the efficacy of a knowledge/skills exam is to give it to a group of current employees who are known to be successful and to another group who are less successful. Correlating the results could lead to a set of questions that “good” employees are more likely to get right (or wrong). Alternatively, a company could survey those employees who were fired for their poor skills to identify any common trends. Clearly though, while such studies might be interesting, they would be nearly impossible to conduct, and in any case, it is unclear if these tests measure what a hiring manager really wants to know.

Perhaps skill or knowledge testing is not what is needed. In many cases, a failure in attitude leads to performance shortfalls. Perhaps then some sort of assessment of potential to get along with others might be needed. Some organizations will measure a candidate’s compatibility with the rest of the team by using tests that measure emotional intelligence or some variation of that (for example, the Kersey’s Temperament Sorter Test). The idea is to establish the “style” of existing team members based on their personalities, and then look to add someone whose style is compatible. This practice, however, can be used to illegally discriminate.

In any case, it is questionable as to whether these kinds of tests really do lead to better hires. Unfortunately, it is also hard to gauge a person’s fit with the team based on a series of interviews with team members and the obligatory group lunch interview. Anyone can put on airs for a few hours in order to get hired, and most interviewers are not specifically trained to see through the act.

9.7.2 RECOMMENDED PRACTICES

If the company must use an assessment of “intelligence,” college grades or graduate record examination scores may be used. If the real goal is to determine programming prowess, checking grades in programming courses might be helpful. Better still, the

manager can ask the candidate to bring in a sample of some code he or she developed and discuss it. If the code sample is too trivial or the candidate struggles in explaining it, it is likely that he or she did not write it or understand it that well. Other skills can be tested this way too; for example, if the job entails writing software specifications or design or test plans, the candidate can be asked to provide a writing sample. It is possible that candidates do not have samples from their current or past jobs for proprietary or security reasons, but they should still be able to talk about what they did without notes. If they can recount the project in great detail, then they probably know what they are talking about.

As for compatibility with the rest of the team, there is no magic here. Managers have to do their homework. They must spend time with candidates in a variety of settings, even having lunch with them (a lot can be learned about a person from his or her manners, for example). Make sure that whoever will be working closely with the applicant is on the hiring or interview team and gets to meet the candidate. But make sure that everyone has been trained on what to look for during an interview. Then get together and compare notes right after the candidate leaves.

Background checking is important too, and most people do not do a good job on the phone dealing with references. Here are some simple guidelines for background checking:

1. First, ask legal questions. There are many questions that cannot be asked; a human resources or legal advisor should be consulted before writing the interview questions.
2. Be sure you check at least three references. It is harder to hide any problems this way.
3. Be sure to talk to supervisors, peers, and subordinates. A team member has to be able to lead and be led.
4. Take good notes and ask follow-up questions. Many references are reluctant to say bad things about people even if they do not believe the person is a strong candidate. By listening carefully to what references say — and do not say — the real message will come through.
5. Be sure to ask a broad range of questions, especially questions that encourage elaboration (as opposed to yes/no answers). For example, some of the following might be helpful:
 - a. “Describe the candidate’s technical skills.”
 - b. “Describe a difficult situation that the candidate encountered and how he dealt with it.”
 - c. “Describe the candidate’s interpersonal skills.”
 - d. “Why do you think the candidate is leaving the company?”
 - e. “Describe the kind of work environment in which the candidate would thrive.”
 - f. “Describe the current work environment in which the candidate works.”
 - g. “Describe the contributions of the candidate in the capacities with which you are familiar.”
 - h. “Describe the kind of manager that you think would be best for the candidate.”

6. Evaluate the references that candidates give you. If they have difficulty providing references, or if none are for a direct supervisor (or subordinate), then this may indicate some problem. If the reference barely knows the candidate or simply worked in the same building, then regardless of the person's opinions, they cannot be weighed heavily.

In summary, it is crucial that the hiring manager and hiring or interview team learn the art of interviewing and background checking. This is more likely to lead to the right fit than a series of tests. Relying on "trivia" tests is risky at best and might cost a good employee.

9.8 TRACKING AND REPORTING PROGRESS

Tracking progress is important for identifying problems early, for reporting purposes, and for performing appropriate resource allocation and reallocation as required. Three tools that can help the software project manager to measure the progress of a project are:

1. Gantt chart
2. Critical path method (CPM)
3. Program evaluation and review technique (PERT)

There are numerous commercial implementations of these tools, which typically can convert from one to the other and integrate with many popular word processing, spreadsheet, and presentation software.

9.8.1 GANTT CHART

The ubiquitous Gantt chart was developed by Henry Gantt during the First World War as a planning tool. The approach is simple in that it lists project tasks in a sequential or parallel fashion.

Project tasks are listed along the left-hand side of the chart in a hierarchical fashion. If a work breakdown structure was used in the software design description, then it can be transferred to the chart.

The chart provides a visual aid in assessing progress for each task and its subordinate and dependent tasks. So long as the chart is a realistic reflection of the situation, the manager can use it to track progress, adjust the schedule, and perhaps, most importantly, communicate about the status of the project.

A time line is drawn along the bottom edge of the chart. Each project subtask activity is represented by a directed arrow. The starting point of the arc is placed at the point in the time line where the task would commence. Project durations are represented by the length of the arcs. Personnel are listed next to the project activity on the left-hand side. Milestones can be marked, and task slippage can be denoted by dashed lines in the activity arcs. The chart is redrawn as necessary.

As an example, consider the simple work breakdown structure for the visual inspection system (VIS). A corresponding Gantt chart is shown in Figure 9.2. It can

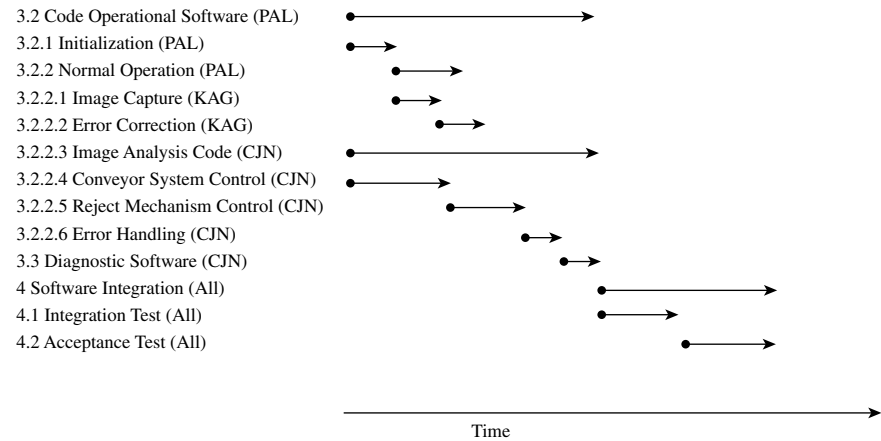


FIGURE 9.2 Partial Gantt chart for the VIS.

be seen from the chart that parallel tasks can be identified and sequencing easily depicted. Task assignments can be made by writing the name of the person responsible next to each task. PAL, CJN, and KAG are the initials of the persons assigned to the tasks; “All” represents all team members.

Here the time units are omitted, but would usually be represented by tick marks in units of days, weeks, or months. Furthermore, personnel would be assigned to each task. Commercial products are available for building these charts and for depicting the activities as a CPM or PERT chart.

9.8.2 CRITICAL PATH METHOD

The CPM is an improvement on the Gantt chart in that task dependencies can be more easily depicted and task times are represented numerically rather than visually. The method was developed in the 1950s by researchers at DuPont and Remington Rand.

The CPM chart is essentially a precedence graph (also known as a Hasse diagram) connecting tasks and illustrating their dependencies, as well as the budgeted completion time and maximum cumulative completion time, along the path from the origin to the current task (Figure 9.3). Because one of the paths will be longer than the others, the project manager can identify what is known as the critical path. The critical path is the one that can be improved to reduce overall project completion time.

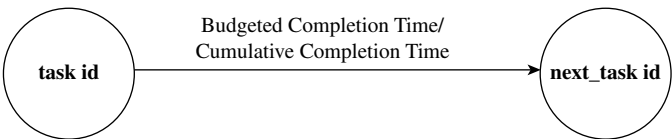


FIGURE 9.3 Basic element of CPM showing first task and next dependent task. The arc is labeled with best, average, and worst-case completion times.

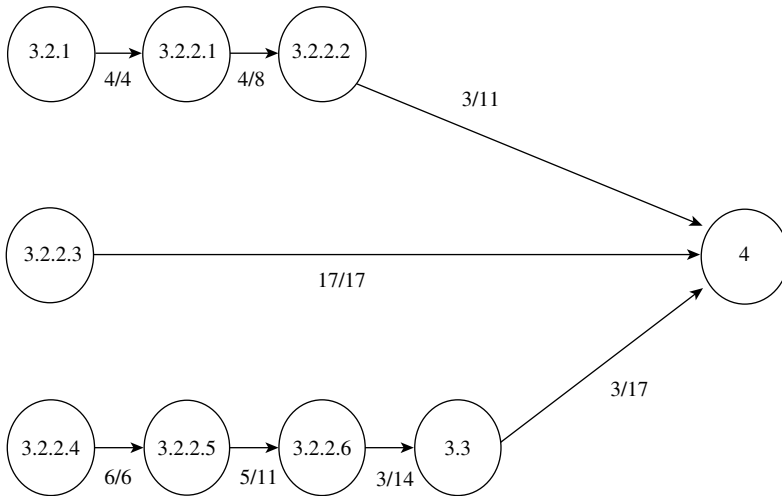


FIGURE 9.4 Partial CPM corresponding to VIS Gantt chart shown in Figure 9.2.

Returning again to the VIS example, consider the tasks in the work breakdown structure by their numerical coding shown in the Gantt chart. These tasks are depicted in Figure 9.4. Here tasks 3.2.1, 3.2.2.3, and 3.2.2.4 may all begin simultaneously. Assume that task 3.2.1 is expected to take four time units (days). Notice that, for example, the arc from task 3.2.1 is labeled with “4/4” because the estimated time for that task is four, and the cumulative time along that path up to that node is four. Looking at task 3.2.2.1, which succeeds task 3.2.1, we see that the edge is labeled with “4/8.” This is because a completion time for task 3.2.2.1 is estimated at 4 days, but the cumulative time for that path (from tasks 3.2.1 through 3.2.2.1) is estimated to take a total of 8 days.

Finally, task 3.2.2.2 is expected to take 3 days, and hence the cumulative completion time is 11 days. On the other hand, task 3.2.2.3 is expected to take 17 days. The task durations are based on either estimation or using a tool such as COCOMO, which will be discussed later. If a Gantt chart accompanies the CPM diagram, the task durations represented by the length of the arrows there should correspond to those labeled on the CPM chart.

Moving along the last path at the bottom of Figure 9.4, it can be seen that the cumulative completion time is 17. Therefore, in this case, the two lower task paths represent critical paths. Hence, only by reducing the completion time of both can the project completion be accelerated.

9.8.3 PROGRAM EVALUATION AND REVIEW TECHNIQUE

PERT was developed by the Navy and Lockheed (now Lockheed Martin) in the 1950s, around the same time as CPM. PERT is the same as CPM topologically, except that PERT depicts optimistic, likely, and pessimistic completion times along each arc between tasks (Figure 9.5). Figure 9.6 is a partial PERT chart for the VIS.

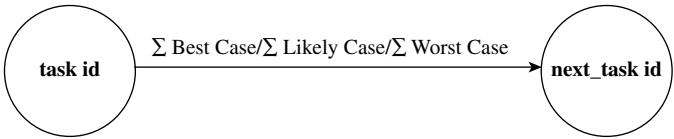


FIGURE 9.5 Partial PERT corresponding to VIS Gantt chart shown in Figure 9.2.

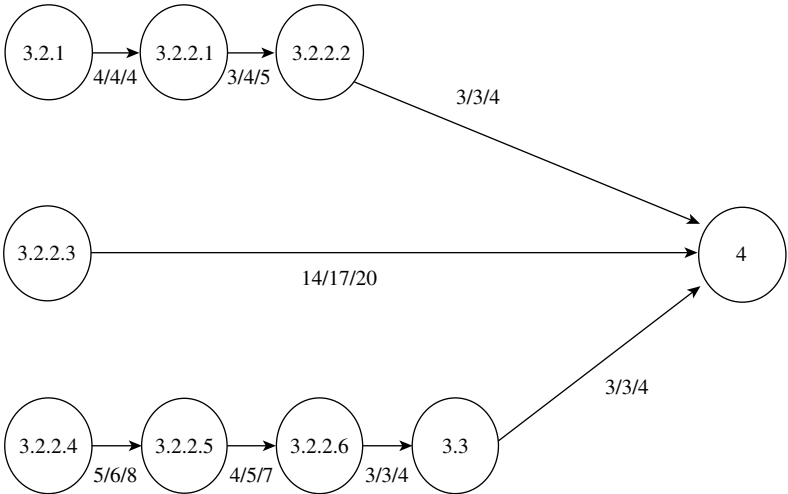


FIGURE 9.6 Partial PERT chart for VIS showing best-case, likely, and worst-case completion times for each task.

In this depiction, it can be seen that the topology is the same as that for CPM. Here the triples indicate the best, likely, and worst-case completion times for each task. These times are estimated, as in CPM, either through best engineering judgment or through the use of a tool like COCOMO.

Adding these triples vectorially yields the PERT chart in Figure 9.7. The aggregated times can now be seen along the arcs, providing cumulative best, likely, and worst-case scenarios. This provides even more control information than the Gantt or CPM project representations.

9.9 COST ESTIMATION USING COCOMO

One of the most widely used software modeling tools is Boehm’s COCOMO model, first introduced in 1981. COCOMO is an acronym for constructive cost model. That is, it is a predictive model. There are three versions of COCOMO: basic, intermediate, and detailed.

9.9.1 BASIC COCOMO

The basic COCOMO model is based on thousands of lines of deliverable source instructions. In short, for a given piece of software, the time to complete is a function

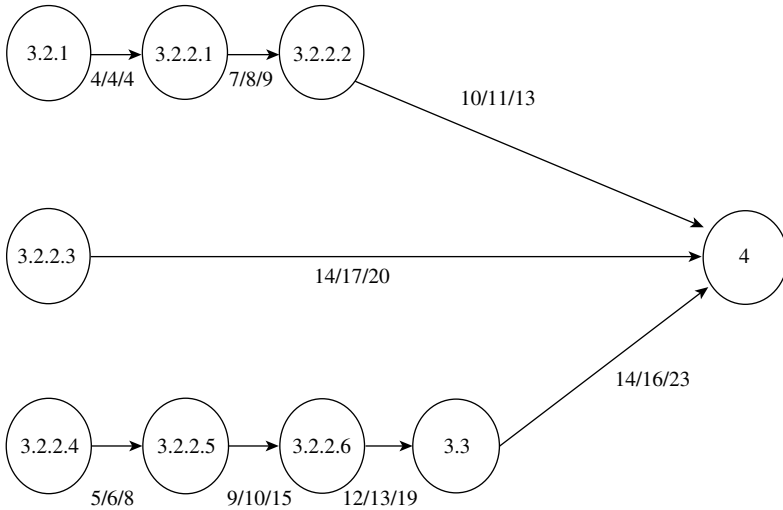


FIGURE 9.7 Partial PERT chart for VIS showing cumulative best-case, likely, and worst-case completion times for each task.

of L , the number of lines of delivered source instructions (KDSIs), and two additional parameters, a and b , which will be explained shortly. This is the effort equation for the basic COCOMO model.

$$T = aL^b \quad (9.1)$$

Dividing T by a known productivity factor, in KLOC (thousands of lines of code) per person-month, yields the number of person-months estimated to complete the project. The parameters a and b are a function of the type of software system to be constructed.

For example, if the system is organic, that is, one that is not heavily embedded in the hardware, then the following parameters are used: $a = 3.2$, $b = 1.05$. If the system is semidetached, that is, partially embedded, then these parameters are used: $a = 3.0$, $b = 1.12$.

Finally, if the system is embedded, that is, closely tied to the underlying hardware like the VIS, then the following parameters are used: $a = 2.8$, $b = 1.20$. Note that the exponent for the embedded system is the highest, leading to the longest time to complete for an equivalent number of delivered source instructions.

Recall from Chapter 7 that for the VIS, using feature points, 40,000 lines of C code were estimated. Hence, an effort level estimate is obtained using COCOMO of

$$T = 2.8 \cdot (40K)^{1.2} = 234K$$

Suppose, then, it is known that an efficient software engineer using computer-aided software engineering and other tools can generate 2000 lines of code per month. Then superficially, at least, it might be estimated that the project would take

about 117 person-months to complete. Not counting dependencies in the task graph, this implies that a five-person team would take about 20 months to complete the project. It would be expected, however, that more time would be needed because of task dependencies (identified, e.g., using PERT).

9.9.2 INTERMEDIATE AND DETAILED COCOMO

The intermediate and detailed COCOMO models dictate the kinds of adjustments used. Consider the intermediate model, for example. Once the effort level for the basic model is computed based on the appropriate parameters and number of source instructions, other adjustments can be made based on additional factors.

In this case, for example, if the lines of code to be produced consist of design modified, code modified, and integration modified code rather than straight code, a linear combination of these relative percentages is used to create an adaptation adjustment factor as follows.

Adjustments are then made to T based on two sets of factors: the adaptation adjustment factor, A , and the effort adjustment factor, E .

The adaptation adjustment factor is a measure of the kind and proportion of code that is to be used in the system, namely, design modified, code modified, and integration modified. The adaptation factor, A , is given by Equation 9.2:

$$A = 0.4 (\% \text{ design modified}) + 0.03 (\% \text{ code modified}) + 0.3 (\% \text{ integration modified}) \quad (9.2)$$

For new components, $A = 100$. On the other hand, if all of the code is design modified, then $A = 40$, and so on. Then the new estimation for delivered source instructions, E , is given as

$$E = L \cdot A / 100 \quad (9.3)$$

An additional adjustment, the effort adjustment factor, can be made to the number of delivered source instructions based on a variety of other factors, including:

- Product attributes
- Computer attributes
- Personnel attributes
- Project attributes

Each of these attributes is assigned a number based on an assessment that rates them on a relative scale. Then a simple linear combination of the attribute numbers is formed based on project type. This gives a new adjustment factor, E' .

The second effort adjustment factor, E'' , is then made based on the formula

$$E'' = E' \cdot E \quad (9.4)$$

Then the delivered source instructions are adjusted, yielding the new effort equation:

$$T = aE''^b \quad (9.5)$$

The detailed model differs from the intermediate model in that different effort multipliers are used for each phase of the software life cycle.

COCOMO is widely recognized and respected as a software project management tool. It is useful even if the underlying model is not really understood. COCOMO software is commercially available and can even be found on the Web for free use. One drawback, however, is that the model does not take into account the leveraging effect of productivity tools.

Finally, the model bases its estimation almost entirely on lines of code, not on program attributes, which is something that function points do. Function points, however, can be converted to lines of code using standard conversion formulas, as was shown.

9.9.3 COCOMO II

COCOMO II is a major revision of COCOMO that is evolving to deal with some of the previously described shortcomings. For example, the original COCOMO 81 model was defined in terms of delivered source instructions. COCOMO II uses the metric source lines of code instead of delivered source instructions. The new model helps better accommodate more expressive modern languages as well as software generation tools that tend to produce more code with essentially the same effort.

In addition, in COCOMO II some of the more important factors that contribute to a project's expected duration and cost are included as new scale drivers. These five scale drivers are used to modify the exponent used in the effort equation:

- Precedentedness (that is, novelty of the project)
- Development flexibility
- Architectural/risk resolution
- Team cohesion
- Process maturity

The first two drivers, precededness and development flexibility, for example, describe many of the same influences found in the adjustment factors of COCOMO 81.

It is beyond the scope of this text to study COCOMO in detail. As with any metric and model, it must be used carefully and based on practice and experience. Nevertheless, using proven models is better than none at all or guessing.

9.10 EXERCISES

- 9.1 Compare FP and \overline{FP} examples shown in the previous chapter to the COCOMO basic, intermediate, and detained models. Make appropriate assumptions where necessary.

9.2 Research commercially available tools that can be used to implement:

- Gantt charts
- PERT
- CPM

What free versions are available?

9.3 Find a demo version of COCOMO 81 or COCOMO II on the Web and experiment with the VIS example.

9.4 How can COCOMO be used to measure progress? When should it be used?

9.5 Are there any inherent problems in using project management tools like Gantt charts, PERT, CPM, and COCOMO, which were developed in a “procedural-oriented world” with object-oriented techniques?

9.6 Does self-auditing of projects by managers/team leaders and reviews of goals by team members prove fruitful? Or is outside oversight more impartial to a project team that might overrank itself?

Glossary

Many of these terms are taken, with permission, from the *CRC Comprehensive Dictionary of Computer Science, Engineering and Technology* (Laplante, 2001).

Term	Definition
Abstract class	A superclass that has no direct instances.
Adaptive programming	A lightweight programming methodology that offers a series of frameworks to apply adaptive principles and encourage collaboration.
Agile programming	A lightweight programming methodology that is divided into four activities: planning, designing, coding, and testing, all performed iteratively.
Algorithm	A systematic and precise step-by-step procedure for solving certain kinds of problems or accomplishing a task, for instance, converting a particular kind of input data to a particular kind of output data, or controlling a machine tool. An algorithm can be executed by a machine.
Approximate reasoning	See <i>imprecise computation</i> .
Argument	(1) An address or value that is passed to a procedure or function call, as a way of communicating cleanly across procedure and function boundaries. (2) Data given to a hardware operator block.
Arithmetic operation	Any of the following operations and combination thereof: addition, subtraction, multiplication, division.
Artifact	Any by-product of the software production process, including code and documentation.
Assembler	A computer program that translates an assembly-code text file to an object file suitable for linking.
Attribute	A named property of a class that describes a value held by each object of the class.
Benchmark	Standard tests that are used to compare the performance of computers, processors, circuits, or algorithms.

Term	Definition
Branch instruction	An instruction is used to modify the instruction execution sequence of the CPU. The transfer of control to another sequence of instructions may be unconditional or conditional based on the result of a previous instruction. In the latter case, if the condition is not satisfied, the transfer of control will be to the next instruction in sequence. It is equivalent to a jump instruction, although the range of the transfer may be limited in a branch instruction compared to the jump.
Branch prediction	A mechanism used to predict the outcome of branch instructions prior to their execution.
Break point	(1) An instruction address at which a debugger is instructed to suspend the execution of a program. (2) A critical point in a program at which execution can be conditionally stopped to allow examination (if the program variables contain the correct values) or other manipulation of data. Break-point techniques are often used in modern debuggers, which provide nice user interfaces to deal with them.
Break-point instruction	A debugging instruction provided through hardware support in most microprocessors. When a program hits a break point, specified actions occur that save the state of the program, and then switch to another program that allows the user to examine the stored state. The user can suspend the execution of a program; examine the registers, stack, and memory; and then resume the program's execution, which is very helpful in a program's debugging.
Built-in self-test (BIST)	Special software used to perform self-testing. On-line BIST assures testing concurrently with normal operation (e.g., accomplished with coding or duplication techniques). Off-line BIST suspends normal operation and is carried out using a built-in test pattern generator and a test response analyzer (e.g., Signature analyzer).
Call-by-address	See <i>call-by-reference</i> .
Call-by-reference	Parameter passing mechanism in which the address of the parameter is passed by the calling routine to the called procedure so that it can be altered there. Also known as call-by-address.
Call-by-value parameter passing	Parameter passing mechanism in which the value of the actual parameter in the subroutine or function call is copied into the procedure's formal parameter.

Term	Definition
Capability	An object that contains both a pointer to another object and a set of access permissions that specify the modes of access permitted to the associated object from a process that holds the capability.
Checkpoint	Time in the history of execution at which a consistent version of the system's state is saved so that if a later event causes potential difficulties, the system can be restarted from the state that had been saved at the checkpoint. Checkpoints are important for the reliability of a distributed system, since timing problems or message loss can create a need to "back up" to a previous state, which has to be consistent in order for the overall system to operate functionally.
Checkpointing	Method used in rollback techniques in which some subset of the system states (data, program, etc.) is saved at specific points (checkpoints), during the process execution, to be used for recovery if a fault is detected.
Checksum	A value used to determine if a block of data has changed. The checksum is formed by adding all of the data values in the block together and then finding the two's complement of the sum. The checksum value is added to the end of the data block. When the data block is examined (possibly after being received over a serial line), the sum of the data values and checksum should be zero.
Class	A group of objects with similar attributes, behavior, and relationships to other objects.
Code	(1) A technique for representing information in a form suitable for storage or transmission. (2) A mapping from a set of messages into binary strings.
Coding	The process of programming, generating code in a specific language, and translating data from a representation form into a different one by using a set of rules or tables.
Command	(1) Directives in natural language or symbolic notations entered by users to select computer programs or functions. (2) Instructions from the central processor unit (CPU) to controllers and other devices for execution. (3) A CPU command or a single instruction, add, load, etc.
Compiler	A program that translates a high-level language program into an executable machine instruction program or other lower-level form, such as assembly language.

Term	Definition
Complement	(1) To swap 1 for 0 and 0 for 1 in a binary number. (2) Opposite form of a number system.
Computer-aided software engineering (CASE)	A computer application automating the development of graphics and documentation of application design.
Computer simulation	A set of computer programs that allows one to model the important aspects of the behavior of the specific system under study. Simulation can aid the design process (e.g., by allowing one to determine appropriate system design parameters) or the analysis process (e.g., by allowing one to estimate the end-to-end performance of the system under study).
Conditional instruction	An instruction that performs its function only if a certain condition is met.
Configuration	Operation in which a set of parameters is imposed for defining the operating conditions.
Correctness	A property in which the software does not deviate from the requirements specification. Often used synonymously with reliability, correctness requires a stricter adherence to the requirements.
Crystal	A lightweight programming methodology that empowers the development team to define the development process and refine it in subsequent iterations until it is stable.
Data dependency	The normal situation in which the data that an instruction uses or produces depends upon the data used or produced by other instructions such that the instructions must be executed in a specific order to obtain the desired results.
Data-oriented methodology	An application development methodology that considers data the focus of activities because they are more stable than processes.
Data structure	A particular way of organizing a group of data, usually optimized for efficient storage, fast search, fast retrieval, and fast modification.
Debug	To remove errors from hardware or software.
Debug port	The facility to switch the processor from run mode into probe mode to access its debug and general registers.
Debugger	(1) A program that allows interactive analysis of a running program, by allowing the user to pause execution of the running program and examine its variables and path of execution at any point. (2) Program that aids in debugging.

Term	Definition
Debugging	(1) Locating and correcting errors in a circuit or a computer program. (2) Determining the exact nature and location of a program error and fixing the error.
Default	The value or status that is assumed unless otherwise specified.
Dependability	System feature that combines such concepts as reliability, safety, maintainability, performance, and testability.
Disassembler	A computer program that can take an executable image and convert it back into assembly code.
Distributed computing	An environment in which multiple computers are networked together and the resources from more than one computer are available to a user.
DSI	Delivered source instruction. See <i>KLOC</i> .
Dynamic systems development method (DSDM)	A lightweight programming methodology conceived as a methodology for rapid application development, DSDM relies on a set of principles that include empowered teams, frequent deliverables, incremental development, and integrated testing.
Embedded software	Software that is part of an embedded system.
Embedded system	A computing machine contained in a device whose purpose is not to be a computer. For example, the computers in automobiles and household appliances are embedded computers. An embedded computer uses embedded software, which integrates an operating system with specific drivers and application software. Their design often requires special software–hardware co-design methods for speed, low power, low cost, high testability, or other special requirements.
Emulate	Executing a program compiled to one instruction set on a microprocessor that uses an incompatible instruction set, by translating the incompatible instructions while the program is running.
Emulation	(1) A model that accepts the same inputs and produces the same outputs as a given system. (2) To imitate one system with another.
Emulation mode	State describing the time during which a microprocessor is performing emulation.
Emulator	(1) The firmware that simulates a given machine architecture. (2) A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system.

Term	Definition
Encapsulation	Property of a program that describes the complete integration of data with legal process relating to the data.
Entity relationship diagram	A diagram that describes the important entities in a system and the ways in which they are interrelated.
Environment	A set of objects outside the system, a change in whose attributes affects, and is affected by, the behavior of the system.
Error extension	The multiplication of errors that might occur during the decoding of a line coded sequence, or during the decoding of a forward error control coded sequence when the number of symbol errors exceeds the error correction capability of the code.
Extreme programming (XP)	A lightweight programming methodology based on 12 practices, including pair programming (all code developed jointly by two developers), test-first coding, having the customer on-site, and frequent refactoring. Extreme programming is perhaps the most prescriptive of the lightweight methodologies.
Failure	Manifestation of any defect. It relates to execution of wrong actions, nonexecution of correct actions, performance degradation, etc.
Fault prevention	Any technique or process that attempts to eliminate the possibility of having a failure occur in a hardware device or software routine.
Fault tolerance	Correct execution of a specified function in a circuit (system), provided by redundancy, despite faults. The redundancy provides the information needed to negate the effects of faults.
Feature-driven development	A lightweight model-driven, short-iteration process built around the feature, a unit of work that has meaning for the client and developer and is small enough to be completed quickly.
Finite state automaton (FSA)	See <i>finite state machine</i> .
Finite state machine (FSM)	A mathematical model of a machine consisting of a set of inputs, a set of states, and a transition function that describes the next state given the current state and an input. Also known as finite state automaton and state transition diagram.
Firm real-time system	A real-time system that can fail to meet one or more deadlines without system failure.

Term	Definition
Flowchart	A traditional graphic representation of an algorithm or a program that uses named functional blocks (rectangles), decision evaluators (diamonds), and I/O symbols (paper, disk) interconnected by directional arrows that indicate the flow of processing. Synonym: flow diagram.
Forward error recovery	A technique (also called roll-forward) of continuing processing by skipping faulty states (applicable to some real-time systems in which occasional missed or wrong responses are tolerable).
Framework	A skeletal structure of a program that requires further elaboration.
FSA	Finite state automaton. See <i>finite state machine</i> .
FSM	See <i>finite state machine</i> .
Function test	A check for correct device operation generally by truth table verification.
Functional decomposition	The division of processes into modules.
Garbage	An object or a set of objects that can no longer be accessed, typically because all pointers that direct accesses to the object or set have been eliminated.
Garbage collector	A software run-time system component that periodically scans dynamically allocated storage and reclaims allocated storage that is no longer in use (garbage).
Generalization	The relationship between a class and one or more variations of that class.
Global variable	Any variable that is within the scope of all modules of the software system.
Hard real-time system	A real-time system in which missing even one deadline results in system failure.
Hazard	A momentary output error that occurs in a logic circuit because of input signal propagation along different delay paths in the circuit.
Heterogeneous	Having dissimilar components in a system; in the context of computers, having different types or classes of machines in a multiprocessor or multicomputer system.
Host	A computer that is the one responsible for performing a certain computation or function.
ICE	See <i>in-circuit emulator</i> .
Imprecise computation	Techniques involving early truncation of a series in order to meet deadlines. Sometimes called approximate reasoning.

Term	Definition
In-circuit emulator (ICE)	A device that replaces the processor and provides the functions of the processor plus testing and debugging functions.
Incrementality	A software approach in which progressively larger increments of the desired product are developed.
Information hiding	A program design principle that makes available to a function only the data it needs.
Inheritance	In object orientation, the possibility for different data types to share the same methods.
Initialize	(1) To place a hardware system in a known state, for example, at power up. (2) To store the correct beginning data in a data item, for example, filling an array with zero values before it is used.
Instance	An occurrence of a class.
Instruction issue	The sending of an instruction to functional units for execution.
Instruction set	The instruction set of a processor is the collection of all the machine-language instructions available to the programmer.
Interoperability	Software quality that refers to the ability of the software system to coexist and cooperate with other systems.
Interpreter	A computer program that translates and immediately performs intended operations of the source statements of a high-level language program.
Interrupt	An input to a processor that signals the occurrence of an asynchronous event. The processor's response to an interrupt is to save the current machine state and execute a predefined subprogram. The subprogram restores the machine state on exit, and the processor continues in the original program.
Interrupt handler	A predefined subprogram executed when an interrupt occurs. The handler may perform input or output, save data, update pointers, or notify other processes of the event. The handler must return to the interrupted program with the machine state unchanged.
KLOC	A software metric measuring thousands of lines of code (not counting comments and nonexecutable statements). Called the clock metric. Also known as thousands of delivered source instructions (KDSIs) and noncommented source statements (NCSSs).
Legacy system	Applications that are in a maintenance phase but are not ready for retirement.

Term	Definition
Library	A set of precompiled routines that may be linked with a program at compile time or loaded at load time or dynamically at run time.
Lightweight programming methodology	Any programming methodology that is adaptive rather than predictive and emphasizes people rather than process.
Link	The portion of the compilation process in which separate modules are placed together and cross-module references resolved.
Linker	A computer program that takes one or more object files, assembles them into blocks that are to fit into particular regions in memory, and resolves all external (and possibly internal) references to other segments of a program and to libraries of precompiled program units.
Logic analyzer	A machine that can be used to send signals to, and read output signals from, individual chips or circuit boards.
Logical operation	The machine-level instruction that performs Boolean operations such as AND, OR, and COMPLEMENT.
Machine code	The machine format of a compiled executable, in which individual instructions are represented in binary notation.
Machine language	The set of legal instructions to a machine's processor, expressed in binary notation.
Macro	A short code-like text, defined by the programmer, which the assembler or compiler will recognize and which will result in an in-line insertion of a predefined block of code into the source code.
Maintainability	A software quality that is a measure of how easy the system can be evolved to accommodate new features, or changed to repair errors.
Maintenance	The changes made on a system to fix errors, to support new requirements, or to make it more efficient.
Mealy finite state machine	A finite state machine with outputs.
Memory reference instruction	An instruction that communicates with virtual memory, writing to it (store) or reading from it (load).
Message-passing system	A multiprocessor system that uses messages passed among the processors to coordinate and synchronize the activities in the processors.
Microcode	A collection of low-level operations executed as a result of a single instruction being issued.

Term	Definition
Modularity	Design principle that calls for the design of small, self-contained code units.
Moore finite state machine	See <i>finite state machine</i> .
Multiprocessor	A computer system that has more than one internal processor capable of operating collectively on a computation. Normally associated with those systems where the processors can access a common main memory.
NCSS	Noncommented source statement. See <i>KLOC</i> .
Nested subroutine	A subroutine called by another subroutine. The programming technique of a subroutine calling another subroutine is called nesting.
Object	An instance of a class definition.
Object code	A file comprising an intermediate description of a program segment.
Object type	The type of an object determines the set of allowable operations that can be performed on the object. This information can be encoded in a “tag” associated with the object, can be found along an access path reaching to the object, or can be determined by the compiler that inserts “correct” instructions to manipulate the object in a manner consistent with its type.
Object oriented	The organization of software into discrete objects that encapsulate both data structure and behavior.
Object-oriented analysis	A method of analysis that examines requirements from the perspective of the classes and objects found in the problem domain.
Object-oriented design	A design methodology viewing a system as a collection of objects with messages passed from object to object.
Object-oriented methodology	An application development methodology that uses a top-down approach based on a decomposition of a system in a collection of objects communicating via messages.
Open source code	Source code that is made available to the user community for moderate improvement and correction.
Open system	An extensible collection of independently written applications that cooperate to function as an integrated system.
Operating system	A set of programs that manages the operations of a computer. It oversees the interaction between the hardware and the software and provides a set of services to system users.

Term	Definition
Operation	Specification of one or a set of computations on the specified source operands, placing the results in the specified destination operands.
Output dependency	The situation when two sequential instructions in a program write to the same location. To obtain the desired result, the second instruction must write to the location after the first instruction.
Overloading	Principle according to which operations bearing the same name apply to arguments of different data type.
Pair programming	A technique in which two persons write code together.
Pattern	A named problem–solution pair that can be applied in new contexts, with advice on how to apply it in novel situations.
Performance	A measure of the software’s capability of meeting certain functional constraints such as timing or output precision.
Portability	A quality in which the software can easily run in different environments.
Power-on self-test (POST)	A series of diagnostic tests performed by a machine (such as the personal computer) when it powers on.
Preprocessing	A series of image enhancements and transformations performed to ease the subsequent image analysis process through, e.g., noise removal or feature extraction/enhancement.
Procedure	A self-contained code sequence designed to be reexecuted from different places in a main program or another procedure.
Procedure call	In program execution, the execution of a machine-language routine, after which execution of the program continues at the location following the location of the procedure call.
Process	The context, consisting of allocated memory, open files, network connections, etc., in which an operating system places a running program.
Process control block (PCB)	An area of memory containing information about the context of an executing program. Although the PCB is primarily a software mechanism used by the operating system for the control of system resources, some computers use a fixed set of PCBs as a mechanism to hold the context of an interrupted process.
Programmed I/O	Transferring data to or from a peripheral device by running a program that executes individual computer instruction or commands to control the transfer. An alternative is to transfer data using direct memory address (DMA).

Term	Definition
Protection	Control access to information in a computer's memory, consistent with a particular policy or mechanism. The term <i>security</i> is used when the constraints and policies are very restrictive.
Protection fault	An error condition detected by the address mapper when the type of request is not permitted by the object's access code.
Prototyping	Building an engineering model of all or part of a system to prove that the concept works.
Pseudo-code	A technique for specifying the logic of a program in an English-like language. Pseudo-code does not have to follow any syntax rules and can be read by anyone who understands programming logic.
Pseudo-exhaustive testing	A testing technique that relies on various forms of code segmentation and application of exhaustive test patterns to these segments.
Pseudo-operation	In assembly language, an operation code that is an instruction to the assembler rather than a machine-language instruction.
Pseudo-random testing	A testing technique based on pseudo-randomly generated test patterns. Test length is adapted to the required level of fault coverage.
Pure procedure	A procedure that does not modify itself during its own execution. The instructions of a pure procedure can be stored in a read-only portion of the memory and can be accessed by many processes simultaneously.
Race condition	A situation where multiple processes access and manipulate shared data with the outcome dependent on the relative timing of these processes.
Random testing	The process of testing using a set of pseudo-randomly generated patterns.
Real-time	Refers to systems whose correctness depends not only on outputs, but also on the timeliness of those outputs. Failure to meet one or more of the deadlines can result in system failure.
Real-time computing	Support for environments in which response time to an event must occur within a predetermined amount of time. Real-time systems may be categorized into hard, firm, and soft real-time.

Term	Definition
Reentrant	Term describing a program that uses concurrently, exactly the same executable code in memory for more than one invocation of the program (each with its own data), rather than separate copies of a program for each invocation. The read and write operations must be timed, so that the correct results are always available and the results produced by an invocation are not over-written by another one.
Recovery	Action that restores the state of a process to an earlier configuration after it has been determined that the system has entered a state that does not correspond to functional behavior. For overall functional behavior, the states of all processes should be restored in a manner consistent with each other, and with the conditions within communication links or message channels.
Recursion	The situation whereby a program calls itself.
Recursive procedure	A procedure that can be called by itself or by another program that it has called; effectively, a single process can have several executions of the same program alive at the same time. Recursion provides one means of defining functions.
Redundancy	The use of parallel or series components in a system to reduce the possibility of failure. Similarly, referring to an increase in the number of components that can interchangeably perform the same function in a system. Sometimes it is referred to as hardware redundancy in the literature to differentiate it from so-called analytical redundancy in the field of FDI (fault detection and isolation/identification). Redundancy can increase the system reliability.
Reentrancy	The characteristic of a block of software code that if present, allows the code in the block to be executed by more than one process at a time.
Refactoring	To perform a behavior preserving code transformation.
Register indirect addressing	An instruction-addressing method in which the register field contains a pointer to a memory location that contains the memory address of the data to be accessed or stored.
Reliability	The probability a component or system will function without failure over a specified period, under stated conditions.

Term	Definition
Requirements analysis	A phase of software development life cycle in which the business requirements for a software product are defined and documented.
Reusability	The possibility to use or easily adapt the hardware or software developed for a system to build other systems.
Reuse	Programming modules are reused when they are copied from one application program and used in another. Reusability is a property of module design that permits reuse.
Reverse engineering	The reverse analysis of an old application to conform to a new methodology.
Robustness	A software quality that measures the software's tolerance to exceptional situations, for example, an input out of range.
Safety	The probability a system will either perform its functions correctly or discontinue its functions in a well-defined, safe manner.
Safety-critical system	A system intended to handle rare unexpected, dangerous events.
Scrum	A lightweight programming methodology based on the empirical process control model, the name is a reference to the point in a rugby match where the opposing teams line up in a tight and contentious formation. Scrum programming relies on self-directed teams and dispenses with much advanced planning, task definition, and management reporting.
Self-modifying code	A program using a machine instruction that changes the stored binary pattern of (usually) another machine instruction in order to create a different instruction, which will be executed subsequently. Definitely not a recommended practice and not supported on all processors.
Self-test	A test that a module, either hardware or software, runs upon itself.
Self-test and repair	A fault-tolerant technique based on functional unit active redundancy, spare switching, and reconfiguration.
Sequential fault	A fault that causes a combinational circuit to behave like a sequential one.
SLOC	See <i>source line of code</i> .
Soft computing	An association of computing methodologies centering on fuzzy logic, artificial neural networks, and evolutionary computing. Each of these methodologies provides us with complementary and synergistic reasoning and searching methods to solve complex, real-word problems.

Term	Definition
Soft real-time system	A real-time system in which failure to meet deadlines results in performance degradation but not necessarily failure.
Software design	A phase of software development life cycle that maps what the system is supposed to do into how the system will do it in a particular hardware or software configuration.
Software development life cycle	A way to divide the work that takes place in the development of an application.
Software engineering	Systematic development, operation, maintenance, and retirement of software.
Software evolution	The process that adapts the software to changes of the environment where it is used.
Software interrupt	A machine instruction that initiates an interrupt function. Software interrupts are often used for system calls because they can be executed from anywhere in memory and the processor provides the necessary return address handling.
Software reengineering	The reverse analysis of an old application to conform to a new methodology.
Source code	Software code written in a form or language meant to be understood by programmers. Must be translated to object code in order to run on a computer.
Source line of code (SLOC)	A metric that measures the number of executable program instructions — one SLOC may span several lines, for example, as in an if-then-else statement.
Specification	A statement of the design or development requirements to be satisfied by a system or product.
Speculative execution	A CPU instruction execution technique in which instructions are executed without regard to data dependencies.
State diagram	A form of diagram showing the conditions (states) that can exist in a logic system and what signals are required to go from one state to another state. Also called finite state machine (FSM) or finite state automaton (FSA).
Subclass	A class that adds specific attributes, behaviors, and relationships for a generalization.
Subroutine	A group of instructions written to perform a task, independent of a main program; can be accessed by a program or another subroutine to perform the task.
Superclass	A class that holds common attributes, behaviors, and relationships for generalization.

Term	Definition
Synchronous	An operation or operations that are controlled or synchronized by a clocking signal.
Syntax	The part of a formal definition of a language that specifies legal combinations of symbols that make up statements in the language.
System implementation	A phase of software development life cycle during which a software product is integrated into its operational environment.
Systems engineering	An approach to the overall life cycle evolution of a product or system. Generally, the systems engineering process is composed of a number of phases. There are three essential phases in any systems engineering life cycle: formulation of requirements and specifications, design and development of the system or product, and deployment of the system. Each of these three basic phases may be further expanded into a larger number. For example, deployment is generally composed of operational test and evaluation, maintenance over an extended operational life of the system, and modification and retrofit (or replacement) to meet new and evolving user needs.
Test-first coding	A software engineering technique in which the code unit test cases are written by the programmer before the actual code is written. Also called test-driven development.
Test pattern	Input vector such that the faulty output is different from the fault-free output.
Testability	The measure of the ease with which a system can be tested.
Testing	A phase of software development life cycle during which the application is exercised for the purpose to find errors.
Timing error	An error in a system due to faulty time relationships between its constituents.
Traceability	A software property concerned with the relationships between requirements, their sources, and the system design.
Tracing	In software engineering, the process of capturing the stream of instructions, referred to as the trace, for later analysis.
UML	See <i>unified modeling language</i> .
Unconditional branch	An instruction that causes a transfer of control to another address without regard to the state of any condition flags.

Term	Definition
Unified modeling language (UML)	A collection of modeling tools for object-oriented representation of software and other enterprises.
Unified process model (UPM)	Process model that uses an object-oriented approach by modeling a family of related software processes using the unified modeling language (UML) as a notation.
UPM	See <i>unified process model</i> .
Usability	A property of software detailing the ease with which it can be used.
Validation	A review to establish the quality of a software product for its operational purpose.
Verifiability	Software property in which its other properties (e.g., portability, usability) can be verified easily.
Virtual machine	A process on a multitasking computer that behaves as if it were a stand-alone computer and not part of a larger system.
WBS	See <i>work breakdown structure</i> .
Work breakdown structure (WBS)	A hierarchically decomposed listing of tasks.

References

- Abdel-Hamid, T.K. and Madnick, S.E., (1986), Impact of schedule estimation on software project behavior, *IEEE Software*, 70–75.
- Akao, Y., Ed., (1990), *Quality Function Deployment*, Productivity Press, Cambridge, MA.
- Andersen, O., (1990), The use of software engineering data in support of project management, *Software Eng. J.*, 5, 350–356.
- Armour, F.J. and Gupta, M., (1999), Mentoring for success, *IT Prof.*, 64–66.
- Bach, J., (1998), The highs and lows of change control, *Computer*, 31, 113–115.
- Banker, R.D. et al., (1993), Repository evaluation of software reuse, *IEEE Trans. Software Eng.*, 19, 4, 379–389.
- Beck, K., (1999), *Extreme Programming Explained: Embrace Change*, Addison-Wesley, New York.
- Beizer, B., (1990), *Software Testing Techniques*, 2nd ed., Van Nostrand Reinhold, New York.
- Bentley, J.L., (1982), *Writing Efficient Programs*, Prentice Hall, Englewood Cliffs, NJ.
- Beyer, H. and Holtzblatt, K., (1995), Apprenticing with the customer: a collaborative approach to requirements definition, *Commun. ACM*.
- Blum, B.I., (1992), *Software Engineering: A Holistic View*, Oxford University Press, New York.
- Boehm, B., (1988), A spiral model of software development and enhancement, *IEEE Comput.*, 21, 61–72.
- Boehm, B.W. and Ross, R., (1989), Theory-W software project management: principles and examples, *IEEE Trans. Software Eng.*, SE-15, 902–916.
- Booch, G., (1991), *Object-Oriented Design with Applications*, Benjamin Cummings, Menlo Park, CA.
- Brooks, F., (1995), *The Mythical Man-Month*, 2nd ed., Addison-Wesley, New York.
- Bucci, G., Campanai, M., and Nesi, P., (1995), Tools for specifying real-time systems, in *Real-Time Systems: The International Journal of Time Critical Systems*, Vol. 8, Kluwer Academic Press, Netherlands, pp. 117–172.
- Budgen, D., (1994), *Software Design*, Addison-Wesley, New York.
- Caldiera, G. and Basili, V.R., (1991), Identifying and qualifying reusable software components, *IEEE Comput.*, 24, 2, 61–70.
- Chikofsky, E.J. and Cross, J.H., II, (1990), Reverse engineering and design recovery: a taxonomy, *IEEE Software*, 7, 1, 13–17.
- Covey, S., (1992), *Principle Centered Leadership*, Simon & Schuster, New York.
- DeMarco, T., (1978), *Structured Analysis and System Specification*, Prentice Hall, Englewood Cliffs, NJ.
- DeMarco, T., (1982), *Controlling Software Projects*, Prentice Hall, Englewood Cliffs, NJ.
- Dougherty, E.R. and Laplante, P.A., (1995a), *Introduction to Real-Time Image Processing*, SPIE Press/IEEE Press, Bellingham, WA.
- Dougherty, E.R. and Laplante, P.A., (1995b), *Real-Time Image Processing*, SPIE Press/IEEE Press, Bellingham, WA.
- Douglass, B.P., (1998), *Real Time UML: Developing Efficient Objects for Embedded Systems*, Addison-Wesley, Reading, MA.

- Everett, W.W., (1990), Software reliability measurement, *IEEE J. Selected Areas Commun.*, 8, 257–252.
- Everett, W.W. and Musa, J.D., (1993), A software reliability engineering practice, *Computer*, 26, 77–79.
- Fenton, N., (1994), Software measurement: a necessary scientific basis, *IEEE Trans. Software Eng.*, 20, 199–206.
- Fenton, N., (1996), *Software Metrics: A Rigorous Approach*, Chapman & Hall, New York.
- Flynn, M.J., (1966), Very high-speed computing systems, *Proc. IEEE*, 54, 12, 1901–1909.
- Frakes, W.B. and Fox, C.J., (1996), Quality improvement using a software reuse failure modes model, *IEEE Trans. Software Eng.*, 22, 4, 274–179.
- Frakes, W.B. and Isoda, S., (1994), Success factors of systematic reuse, *IEEE Software*, 11, 14–19.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J., (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, New York.
- Gane, C. and Saron, T., (1979), *Structured Systems Analysis*, Prentice Hall, Englewood Cliffs, NJ.
- Ghezzi, C., Jazayeri, M., and Mandriolo, D., (1991), *Fundamentals of Software Engineering*, Prentice Hall, Englewood Cliffs, NJ.
- Grady, R., (1992), *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, Englewood Cliffs, NJ.
- Gries, D., (1981), *The Science of Programming*, Springer-Verlag, Heidelberg.
- Haag, S., Raja, M.K., and Schkade, L.L., (1996), Quality function deployment usage in software development, *Commun. ACM*, 41–49.
- Hall, T. and Fenton, N., (1997), Implementing effective software metrics programs, *IEEE Software*, 14, 2, 55–65.
- Hatley, D.J. and Pirbhaj, I.A., (1987), *Strategies for Real-Time System Specification*, Dorest House Publishing, New York.
- Hauser, J.R. and Clausing, D., (1988), The house of quality, *Harv. Bus. Rev.*, 3, May/June, 63–73.
- Henderson-Sellers, B. and Edwards, J.M., (1990), The object-oriented systems life cycle, *Comm. of the ACM*, 33, 9, 152.
- Institute of Electrical and Electronics Engineers, (1998), *IEEE Standard 830-1998: Recommended Practice for Software Requirements Specifications*, IEEE, New York.
- Jacobson, I. et al., (1997), Making the reuse business work, *Computer*, 30, 10, 36–42.
- Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G., (1992), *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, MA.
- Jones, C., (1994), Software metrics: good, bad and missing, *Computer*, 27, 98–100.
- Jones, C., (1995), Determining software schedules, *Computer*, 28, 73–75.
- Jones, C., (1996a), Software change management, *Computer*, 29, 80–82.
- Jones, C., (1996b), Activity based software costing, *Computer*, 27, 98–100.
- Jones, C., (1998), *Estimating Software Costs*, McGraw-Hill, New York.
- Jorgensen, P., (2002), *Software Testing: A Craftsman's Approach*, 2nd ed., CRC Press, Boca Raton, FL.
- Kanoun, K. et al., (1997), Qualitative and quantitative reliability assessment, *IEEE Software*, 14, 77–87.
- Kremer, R., (2000), *Programming Patterns Overview*, University of Calgary, Canada.
- Laplane, P., (2003a), Remember the human element in IT project management, *IT Prof.*, Jan./Feb., 46–50.
- Laplane, P. and Neill, C., (January 2003c), A Class of Kalman Filters for Real-Time Image Processing, paper presented at Proceedings of the Real-Time Imaging Conference, SPIE, Santa Clara, CA, 22–29.

- Laplante, P., Stoyenko, A., and Sinha, D., (1996b), *Image Processing Methods: Real-Time Imaging*, SPIE Press, Bellingham, WA (conference proceedings).
- Laplante, P.A., (1997), *Real-Time Systems Design and Analysis: An Engineer's Handbook*, 2nd ed., IEEE Press/IEEE CS Press, Piscataway, NJ.
- Laplante, P.A., Ed., (2001), *Comprehensive Dictionary of Computer Science, Engineering and Technology*, CRC Press, Boca Raton, FL.
- Laplante, P.A., (2002b), Software engineering: the seven deadly sins, *Opt. Eng.*, 37.
- Laplante, P.A., (2002d), A retrospective on real-time imaging, a new taxonomy and a roadmap for the future, *Real-Time Imaging*, 8, 5, 413–425.
- Laplante, P.A. and Neill, C.J., (January 2002a), An overview of software specification techniques for real-time imaging, in *Real-Time Imaging VI*, vol. 4666, SPIE Press, Bellingham, WA, pp. 55–64.
- Laplante, P.A. and Neill, C.J., (2003b), Software specification and design for imaging systems, *J. Electron. Imaging*, 12, 252–262.
- Laplante, P.A., Neill, C.J., and Jacobs, C., (December 2002e), Requirements Specification Practices: Some Real Data, paper presented at 27th NASA/IEEE Software Engineering Workshop, Greenbelt, MD, IEEE Computer Society Press, Los Alamitos, CA. Proceedings on disk.
- Laplante, P.A., Neill, C.J., and Russell, D.W., (July 2002c), Object-Oriented Requirements Specification for Imaging Systems, paper presented at Proceedings of the Real-Time Imaging Conference, SPIE, Seattle.
- Laplante, P.A. and Stoyenko, A., Eds., (1996a), *Real-Time Image Processing: Theory, Techniques, and Applications*, IEEE Press, New York.
- Larman, C., (2002), *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd ed., Prentice Hall, New York.
- Levine, D. and Schmidt, D., (2000), *Introduction to Patterns and Frameworks*, Washington University, St. Louis.
- Liskov, B., (1988), Data abstraction and hierarchy, *SIGPLAN Notices*, 23, 17–34.
- Lorenz, M. and Kidd, J., (1994), *Object-Oriented Software Metrics*, Prentice Hall, Englewood Cliffs, NJ.
- Louden, K.C., (1993), *Programming Languages Principles and Practice*, PWS-KENT Publishing Company, Boston.
- Low, G.C. and Jeffery, D.R., (1990), Function points in the estimation and evaluation of the software process, *IEEE Trans. Software Eng.*, 16, 64–71.
- Macaulay, L.A., (1996), *Requirements Engineering*, Springer-Verlag, London.
- MacDonald, J.S., (1998), Systems Engineering: Art and Science in an International Context, keynote speech presented at the 1998 INCOSE Symposium.
- Mann, C.C., (2002), Why software is so bad, *Technol. Rev.*, July/Aug., 33–38.
- Mariatos, E.P., Birbas, M.K., Birbas, A.N., and Petrellis, N., (1996) Object-oriented prototyping at the system level: an image reconstruction application example, in *Proceedings of the Seventh IEEE International Workshop on Rapid System Prototyping*, June 19–21, 1996, IEEE Computer Society Press, Los Alamitos, CA, 1996, 90–95.
- Martin, R.C., (May 1996), The Dependency Inversion Principle, *C++ Report*, WordenWare, www.brent.worden.org.
- Matson, J.E. et al., (1994), Software Development Cost Estimation Using Function Points, *IEEE Trans. Software Eng.*, 20, 275–287.
- McCabe, T., (1976), A software complexity measure, *IEEE Trans. Software Eng.*, SE-2, 308–320.
- Metzger, P. and Boddie, J., (1996), *Managing a Programming Project: Processes and People*, 3rd ed., Prentice Hall, Upper Saddle River, NJ.

- Meyer, B., (1998), The role of object-oriented metrics, *Computer*, 31, 123–127.
- Meyer, B., (2000), *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ.
- Mili, H., Mili, F., and Mili, A., (1995), Reusing software: issues and research directions, *IEEE Trans. Software Eng.*, 21, 528–562.
- Möeller, K.H. and Paulish, D.J., (1993), *Software Metrics: A Practitioner's Guide to Improved Product Development*, Chapman & Hall, London.
- Möller, T. and Haines, E., (1999), *Real-Time Rendering*, A. K. Peters, Natick, MA.
- Motus, L. and Rodd, M.G., (1994), *Timing Analysis of Real-Time Software*, Pergamon, Oxford.
- Myers, W., (1997), Software reuse: ostriches beware, *Computer*, 30, 119–120.
- Naks, T. and Motus, L., (2001), Handling timing in a time-critical reasoning system: a case study, *Annu. Rev. Control*, 25, 157–168.
- Neill, C.J. and Gill, B., (2003b), Refactoring reusable business components, *IT Prof.*, Jan./Feb., 33–38.
- Neill, C.J. and Holt, J.D., (2002a), Adding temporal modeling to the UML to support systems design, *Syst. Eng.*, 5, 213–222.
- Neill, C.J. and Laplante, P.A., (July 2002b), Modeling Time in Object-Oriented Specifications of Real-Time Imaging Systems, paper presented at Proceedings of the Real-Time Imaging Conference, SPIE, Seattle.
- Neill, C.J. and Laplante, P.A., (2003a), Specification of real-time imaging systems using UML, *Real-Time Imaging*, 9, 2, 125–137.
- NIST, (2002), The Economic Impact of Inadequate Infrastructure for Software Testing, Planning Report 02-3, www.nist.gov/director/prog-ofc/report02-3.pdf.
- Page-Jones, M., (1980), *The Practical Guide to Structured Systems Design*, Prentice Hall, Englewood Cliffs, NJ.
- Parnas, D.L., (1972), On the criteria to be used in decomposing systems into modules, *Commun. ACM*, 15, 1053–1058.
- Parnas, D.L., (1979), Designing software for ease of extension and contraction, *IEEE Trans. Software Eng.*, SE-5, 128–138.
- Parnas, D.L., (1996), Introduction to Chapter 6.5, in *Great Papers in Computer Science*, Laplante, P.A., Ed., West Publishing, New York.
- Parnas, D.L. and Clements, P.C., (1986), A rational design process: how and why to fake it, *IEEE Trans. Software Eng.*, 12, 251–257.
- Pfleeger, S.L., (1992), Measuring software reliability, *IEEE Spectrum*, 29, 8, 55–60.
- Poling, C., (2002), Designing a machine-vision system, *Opt. Eng.*, May, 34–36.
- Poulin, J., (1997), *Measuring Software Reuse Principles, Practices and Economic Models*, Addison-Wesley, New York.
- Pressman, R.S., (2000), *Software Engineering: A Practitioner's Approach*, 5th ed., McGraw-Hill, New York.
- Putnam, L.H. and Myers, W., (1997), *Industrial Strength Software: Effective Management Using Measurement*, IEEE Computer Society Press, Los Alamitos, CA.
- Quirk, W.J. and Gilbert, R., (1977), *The Formal Specification of the Requirements of Complex Real-Time Systems*, No. 8602, Atomic Energy Research Establishment, Harwell, U.K.
- Rajeswari, M. and Rodd, M.G., (1999), Real-time analysis of an IC wire-bonding inspection system, *Real-Time Imaging*, 5, 409–421.
- Roman, D., Fisher, M., and Cubillo, J., (1998), Digital image processing: an object-oriented approach, *IEEE Trans. Educ.*, 41, 331–333.
- Rosenfeld, A., (2000), Classifying the literature related to computer vision and image analysis, *Comput. Vision Understanding*, 79, 308–323.

- Rzucidlo, M., (2002), Fault tolerance and reliability in software, in *Proceedings of the Research Institute*, February 2002, Penn State Great Valley School of Graduate and Professional Studies, Malvern, PA.
- Sadr, B. and Dousette, P.J., (1996), An OO project management strategy, *Computer*, 29, 33–38.
- Selic, B., Gullekson, G., and Ward, P.T., (1994), *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York.
- Selic, B. and Rumbaugh, J., (March 1998), Using UML for Modeling Complex Real-Time Systems, ObjecTime Limited/Rational Software Corp. white paper, available at www.rational.com.
- Sengupta, K. and Abdel-Hamid, T.K., (1996), The Impact of unreliable information on the management of software projects: a dynamic decision perspective, *IEEE Trans. Syst. Man Cybern.*, A26, 177–189.
- Shi, Y.Q. and Sun, H., (2000), *Image and Video Compression for Multimedia Engineering*, CRC Press, Boca Raton FL.
- Sigried, S., (1996), *Understanding Object-Oriented Software Engineering*, IEEE Press, New York.
- Sinha, P., Gorinsky, S., Laplante, P.A., and Stoyenko, A.D., (1996), A survey of real-time imaging, *J. Electron. Imaging*, 5, 466–478.
- Sommerville, I., (2000), *Software Engineering*, 5th ed., Addison-Wesley, New York.
- Stoyenko, A., Marlowe, T., and Laplante, P.A., (1996), A description language for engineering of complex real-time systems, *Real-Time Syst. J.*, 11, 223–244.
- Svoboda, C.P., (1997), Structured analysis, in *IEEE Software Requirements Engineering*, 2nd ed., IEEE Computer Society Press, Los Alamitos, CA.
- Thayer, R.H., (2002), Software system engineering: a tutorial, *Computer*, 35, 4, 68–73.
- Thomas, A.D.H., Rodd, M.G., Holt, J.D., and Neill, C.J., (1995), Real-time industrial inspection: a review, *Real-Time Imaging J.*, 1, 139–158.
- Tran, T. and Sherif, J.S., (1995), QFD: an effective technique for requirements acquisition and reuse, in *Proceedings of the 2nd IEEE International Software Engineering Standards Symposium*, IEEE Computer Society Press, Los Alamitos, CA, 191–200.
- Tucker, A.B., Jr., Ed., (1996), *The Computer Science and Engineering Handbook*, CRC Press, Boca Raton, FL.
- Voas, J., (2001), The pitfalls of managing a superstar, *IT Prof.*, 3, 2, 65–67.
- Wang, Y. and King, G., (2000), *Software Engineering Processes: Principles and Applications*, CRC Press, Boca Raton, FL.
- Ward, P.T. and Mellor, S.J., (1985), *Structured Development for Real-Time Systems*, Vol. 1, *Introduction & Tools*; Vol. 2, *Essential Modeling Techniques*; Vol. 3, *Implementation Modeling Techniques*, Yourdon Press, New York.
- Wood, A., (1996), Predicting software reliability, *Computer*, 29, 69–77.
- Yourdon, E., (1991), *Modern Structured Analysis*, Prentice Hall, Englewood Cliffs, NJ.

Index

Note: Italicized pages refer to notes, tables and illustrations

A

- abstract data types, 112, 114, 160
- abstract interface, 72
- acceptance testing, 27
- ACM (Association for Computing Machinery), 36
- Ada 95 programming language, 109, 114–115
 - See also* programming languages
 - call-by-value-result parameter passing in, *110*
 - consistency checking in, 145
 - dynamic memory allocation in, 111
 - modularity in, 113
 - pragma pseudo-op in, 115
- adaptive adjustment factor (*A*), 182
- adaptive maintenance, 159
- adaptive programming, 33
- ADC (analog-to-digital converters), 107, 154
- agile programming, 33
- Algol-60 programming language, *110*
- algorithms, 2
- analog-to-digital converters (ADC), 107, 154
- AND states, 50–51
- ANSI-C standard, 113, 122
- anticipation of change, in software engineering, 18
- application programming interface (API), 18, *19*
- approximate reasoning, 157
- AR (autoregressive) model, 100
- arrays, 118
- artifacts, 23
- as** program, 119
- assemblers, 3, 118
- assembly language, 109, 115
 - See also* programming languages
 - lines of code per function point, *136*
 - translation high-level language into, 118
- assessment of project personnel, 174
 - background checking in, 176–177
 - recommended practices in, 175–177
 - skills testing, 174–175
- Association for Computing Machinery (ACM), 36
- Atomic Energy Research Establishment (U.K.), 85
- attributes, 15, 81, 113
- autoregressive (AR) model, 100

B

- B programming language, 115
- background checking, 176–177
- backtracking transitions, 27–28
- BAM (binary angular measure), 155
- baseline method, 132
- basic COCOMO (constructive cost model), 180–182
- BASIC programming language, 109
- bathtub curve, 10–11
- BCPL programming language, 115
- behavior of objects, 80–81
- behavioral model, *60*
- binary angular measure (BAM), 155
- black boxes, 104–105, 162
- black box testing, 139–141
- block diagrams, 55
- block matching, 55–56
- Boolean types, 112
- boundary value testing, 140
- boundary violation errors, 112
- breakpoints, 121
- broadcast communication, 49, 51
- bugs, 46, 138–139
- built-in test software, 105
- burn-in testing, 143
- bus, 94
- bus multiplexers, 107

C

- C beautifier (cb), 121
- C programming language, 109, 115–116
 - See also* programming languages
 - consistency checking in, 145
 - dynamic memory allocation in, 111
 - exception handling in, 113
 - functionality in, 136
 - information hiding in, 113
 - lines of code per function point, *136*
 - modularity in, 113
 - and SASD (structured analysis and structured design), 59

- typing, 112
- C++ programming language, 109, 116
 - See also* programming languages
 - functionality in, 136
 - lines of code per function point, 136
- c2** program, 119
- cache hit ratio, 157
- cache memory, 98
- calibration mode, 47
- call-by-address parameter passing, 110
- call-by-constant parameter passing, 110
- call-by-name parameter passing, 110
- call-by-reference parameter passing, 110, 117
- call-by-value parameter passing, 110, 117
- call-by-value-result parameter passing, 110
- camera input, 4
- CameraProxy element, 88
- capability maturity model (CMM), 34–35
- capability maturity model integration (CMMI), 35
- Carnegie-Mellon University, 34
- CASE (computer-aided software engineering), 27
- CASE tools, 77
- cb** program, 121
- cc** program, 119
- ccom** program, 119
- central processing unit (CPU), 94, 149
- CFDs (control flow diagrams), 78
- chain reactions, 51
- channels, 85
- character types, 112
- charge-couple device (CCD) arrays, 5
- checkpoints, 102, 162
- checksums, 106
- circular message-waiting conditions, 88
- class libraries, 160
- class structure, 19
- Classifier element, 88
- ClassifyStrategy element, 89
- cleanroom software development, 143
- cleanroom testing, 143
- clock probes, 147–148
- CMM (capability maturity model), 34–35
- CMMI (capability maturity model integration), 35
- COCOMO (constructive cost model), 180
 - See also* software project management
 - basic, 180–182
 - COCOMO II, 183
 - intermediate and detailed, 182–183
- COCOMO II, 183
- code unit testing, 27
- codes, 2
 - control and maintenance of, 163
 - debugging, 121–122
 - documenting, 2
 - excessive, 2
 - inspection of, 141
 - link with documentation, 20–21
 - loadable, 146
 - locality of reference, 157
 - reuse of, 2
 - reviews and audits of, 124–126
 - standards, 123–124
 - writing and testing, 118–123
- cohesion, 15–18
- coincident level of cohesion, 16
- collision testing, 56
- communicational level of cohesion, 17
- compilers, 3
 - converting Java to machine or object codes, 117
 - handling errors in, 120
 - in logic analyzers, 147
 - tests on language constructs, 122–123
 - translating high-level language with, 118
 - Unix/Linux C, 119–120
- composite color, 5
- compression, 5
- computer-aided software engineering (CASE), 27
- configuration management software, 26–27
- consistency checking, 45
- constant types, 115
- constructive cost model. *See* COCOMO
- content coupling, 18
- control coupling, 18
- control flow diagrams (CFDs), 78
- control models, 79
- control signals, 4
- control specifications (C-SPECs), 78
- control stores, 78
- conventional model, 24
- conveyors, 6
- coprocessors, 94–95
- core dump, 121
- corrective maintenance, 159
- correctness of software, 12
- cosine function, 156
- coupling, 15–18
- CPM (critical path method), 178–179
- cpp** program, 119
- CPU (central processing unit), 94, 149
- CPU testing, 106
- CPU utilization, 153–154, 157
- CRC (cyclic redundancy code), 106
- critical path method (CPM), 178–179
- Crystal programming, 33
- customers, 43–44
- cycle time (T), 153
- cyclic redundancy code (CRC), 106

D

data abstraction, 80
 data consumption time, 86
 data coupling, 18
 data dictionaries, 59, 76–77
 data flow diagrams (DFDs), 49, 59
 data flow processors, 96
 data types, 112
 DCT (discrete cosine transform), 156
 debugging, 120
 See also errors; software testing
 of imaging systems, 152
 symbolic, 121
 and system cycles, 150
 decision tables, 75
 decision trees, 75
 decompression, 5
 decorators, 88
 delivered source instructions. *See* DSIs
 Department of Defense (DOD), 36, 114
 dependency inversion principle (DIP), 82, 161
 design activity, 71–72
 design constraint requirements, 43
 design patterns, 83, 84
 design recovery, 158
 developers, 43–44
 DFDs (data flow diagrams), 49
 in structured analysis and structured design
 (SASD), 59
 in structured design (SD), 74–76
 DFT (discrete Fourier transform), 156
 diagnostic mode, 47, 79
 digital signal processing (DSP), 94–95, 157
 digital-to-analog converters (DAC), 107
 digitizing imaging devices, 155
 DIP (dependency inversion principle), 82, 161
 direct memory address (DMA), 97–98
 discrete cosine transform (DCT), 156
 discrete Fourier transform (DFT), 156
 disassemblers, 147
 divide-and-conquer methods, 111
 divide-by-zero errors, 97, 112
 DMA (direct memory address), 97–98
 documentation, 126–127
 documentation recovery, 158
 DOD (Department of Defense), 36, 114
 DOD-STD-2167A software standard, 37–38
 DOD-STD-498 software standard, 37
 DOD-STD-7935A, 37
 domain analysis, 161
 domains, 161
 domain-specific logic, 160
 dot coms, 175
 downtime, 10

DRAM (dynamic random access memory), 98
 DSdM (dynamic systems development methods),
 33
 DSIs (delivered source instructions), 129–130
 See also metrics
 adjustment factors, 182
 in basic COCOMO model, 181
 DSP (digital signal processing), 94–95, 157
 DuPont, 178
 dynamic memory allocation, 111
 dynamic random access memory (DRAM), 98
 dynamic requirements, 43
 dynamic systems development methods (DSdM),
 33

E

effort adjustment factor (*E*), 182
 embedded imaging systems, 1
 risk factors, 35–36
 state charts in, 51
 end-product liability, 69
 entity relationship diagrams (ERDs), 59
 environmental model, 60
 epilogues, 115
 equivalence interval, 86
 ERDs (entity relationship diagrams), 59
 errors, 138
 economic cost of, 1
 handling and recovery, 42, 112–113
 hard, 106
 percentage of cost in software development,
 162
 pixel overflow, 112
 as side effect of software development, 8
 soft, 106
 synchronization, 12
 syntax, 120
 evolutionary delivery cycle, 30
 evolutionary prototyping, 30
 evolvability of software, 13
 exception handling, 112–113
 execution time, 154, 162
 exhaustive testing, 140
 exponential functions, 156
 external interface requirements, 43
 extreme programming (XP), 33, 122

F

failure function, 10, 11
 failures, 138–139
 See also errors

- exponential model of, 11
- intensity of, 10
- probability of, 10
- fault tolerance, 99
- faults, 138–139
- fault-tolerant design, 99
 - checkpoints, 102
 - n-version programming, 105–107
 - recovery blocks, 102–104
 - software black boxes, 104–105
 - spatial fault tolerance, 99
 - using Kalman filter, 99–102
- feasibility studies, 41
- feature points, 137
- feature-driven development, 33
- feedback control, 99
- filter operation, 80
- FilterDecorator element, 89
- FilterImp instance, 102
- filtration, 5
- finite state automation (FSA), 46
- finite state machine. *See* FSM
- firing rule, 52
- firm real-time systems, 12
- firmware, 157
- flash memory, 98
- floating-point overflow errors, 112–113
- floating-point types, 112, 155
- flowcharts, 45
 - of multigrid block matching, 56
 - in multiresolution block matching, 55
- formal methods, 44–45
 - FSM (finite state machine), 46–49
 - limitations of, 45–46
 - Petri nets, 51–53
 - state charts, 49–51
 - survey of, 68
 - uses for, 45
 - Z language, 46
- formal program proving, 141–142
- formality in software engineering, 15
- Fortran programming language, 109
 - information hiding in, 113
 - lines of code per function point, 136
 - modularity in, 113
 - and SASD (structured analysis and structured design), 59
- fountain model, 31–32
- fractal compression, 111
- frame grabbers, 5, 6, 107
- FrameGrabber element, 88
- FSA (finite state automation), 46
- FSM (finite state machine), 46–49
 - See also* SRS (software requirements specification)

- design in procedural form, 78–80
- modes of operation, 47
- for transition matrix of visual inspection
 - system, 48
- function points, 134–136
- function polymorphism, 80
- functional level of cohesion, 17
- functional requirements, 42–43
- functionality, control of, 166

G

- Gantt chart, 177–178
- Gantt, Henry, 177
- garbage collection, 113, 118
- Gaussian distribution, 101
- generality in software engineering, 20
- generic function look-up table, 156
- global variables, 110–111, 149
- GoF (gang of four) patterns, 83, 84
- gold plating, 2
- GOTO statement, 118
- graphical objects, 56
- graphical user interface (GUI), 121
- graphics-rendering software, 73
- grid computers, 96
- group walkthroughs, 141
- GUI (graphical user interface), 121

H

- Hadamard matrices, 111
- Halstead's metrics, 132–133
- hard errors, 106
- hard real-time systems, 12
- hardware interrupts, 96–97
- hardware prototypes, 149–150
- Hasse diagram, 178
- Heisenberg uncertainty principle, 152–153
- Heisenberg, Werner, 152
- hierarchical collision testing algorithm, 57
- high-level language, 118
- Hungarian notation standard, 123

I

- ICE (in-circuit emulation), 149
- IEEE (Institute of Electrical and Electronics Engineers), 36
- IEEE Standard 830, 26
 - requirements, 42–43
 - template for, 64–66
- image capture elements, 88

image reconstruction systems, 57–58
 ImageProcessor element, 88, 89
 imaging dilemma, real-time, 12
 imaging systems, 1
 See also visual inspection system
 debugging, 152
 hardware considerations in, 93–99
 line manipulation algorithms in, 155
 prototypes, 36
 real-time, 12
 reliability of, 9–10
 risk factors, 35–36
 second system effect in, 163
 software engineering approach to, 2
 software for, 4–5
 software reengineering in, 157–158
 specifications, 44–45, 53–54
 typical setup, 4
 using Kalman filter in, 99–102
 implementation model, 60
 imprecise computation, 157
 in-circuit emulators, 149
 incremental model, 31
 incrementality in software engineering, 20
 Industry Standard Architecture (ISA), 6
 inertial navigation data, 99
 informal methods, in requirements specification, 45
 information hiding, 18
 See also Parnas partitioning
 in object-oriented programming languages, 80, 113
 in procedural languages, 113
 in software partitioning, 72
 in Z language, 46
 inheritance, 114
 and late binding, 113
 in object-oriented programming language, 80
 in Z language, 46
 in-line patch, 151
 input and output (I/O), 97–99, 161
 input devices, 94
 input source, 5
 InputStream class, 102
 inspection software, 5
 instantiation, 46
 Institute of Electrical and Electronics Engineers (IEEE), 36
 integers, 112, 155
 integrated circuit boards, 6
 intelligence, assessment of, 175–176
 intermediate COCOMO (constructive cost model), 182–183
 International Function Point Users Group, 136
 International Standards Organization (ISO), 36

interoperability of software, 13
 interrupt controllers, 115
 interrupt handling, 96–97, 112, 148
 I/O (input and output), 97–99, 161
 ISA (Industry Standard Architecture), 6
 ISO 9000-9003 standard, 38–39
 ISO (International Standards Organization), 36
 ISO/IEC 12207 standard, 40

J

JAD (joint application design), 124–126
 Java programming language, 109, 113, 117–118
 jitters, 12
 joint application design (JAD), 124–126
 jumper wires, 151

K

Kalman filters, 99–102, 103
 KalmanFilter class, 102
 KalmanMediator class, 102
 kernel components, 150
 KLOC (thousands of lines of code), 129–130, 181

L

language constructs, 122–123
 late binding, 113
 ld program, 119
 leadership, principle-centered, 173–174
 least-significant bit (LSB), 154–155
 legacy systems, 157
 life cycle models, 23–24
 capability maturity model (CMM), 34–35
 evolutionary model, 30
 fountain model, 31–32
 incremental model, 31
 lightweight methodologies, 32–33
 spiral model, 28–30
 unified process model, 34
 V model, 28
 waterfall model, 24–28
 light barriers, 6
 lighting, 5
 lightweight methodologies, 32–33
 line manipulation algorithms, 155
 line scan, 5
 linear sequential model, 24
 lines of code, 129–130
 link loaders, 118
 linkers, 118
 lint, 121, 145

- Linux C compiler, 119–120
- Linux operating system, 115, 121
- Liskov substitution principle, 82–83, 161
- LISP programming language, 114
- load modules, 146
- logic analyzers, 147–148
 - See also* system integration tools
 - in measuring performance of imaging systems, 13
 - timing code, 148–149
 - timing instructions, 148
- logic errors, 120
- logical database requirements, 43
- logical level of cohesion, 16
- longjmp procedure call, 116
- look-up tables, 155–156
- lose-win management paradigm, 172
- lossy compression, 58
- LSB (least-significant bit), 154

M

- machine codes, 118
- machine vision system, 56–57
- macroinstructions, 96
- mainframe computers, 94
- maintainability of software, 13
- maintenance engineers, 43–44
- maintenance process model, 158–159
- management by objectives (MBO), 173–174
- management by sight, 173
- managers, 43–44, 169
- Markov random fields, 57–58
- master processors, 105
- MBO (management by objectives), 173–174
- McCabe's metric, 130–132
- Mealy machine, 48, 49
- mean time between failures (MTBF), 11
- mean time to first failure (MTFF), 11
- memory, 94, 97
- memory maps, 150–151
- memory testing, 106
- memory usage, 157
- memory-mapped I/O, 97
- metrics, 129
 - feature points, 137
 - function points, 134–136
 - Halstead's metrics, 132–133
 - lines of code, 129–130
 - McCabe's metric, 130–132
 - objections to, 138
 - for object-oriented software, 137–138

- software complexity, 130–132
- microcontrollers, 94
- microinstructions, 96
- MIL-STD-2167A software standard, 37
- MIL-STD-498 software standard, 37
- MIL-STD-498 standard, 26
- MIMD (multiple instruction multiple data), 96
- minicomputers, 94
- MISD (multiple instruction single data), 96
- ML programming language, 114
- model checking, 45
- modeling, 3
- Modula-2 programming language
 - dynamic memory allocation in, 111
 - exception handling in, 113
- modularity, 15–18, 113
- monochrome cameras, 5
- Moore machine, 48
- most significant bit (MSB), 155
- motion estimation, 55
- MSB (most significant bit), 155
- MTBF (mean time between failures), 11
- MTFF (mean time to first failure), 11
- multimeters, 147
- multiple instruction multiple data (MIMD), 96
- multiple instruction single data (MISD), 96
- multiprocessing systems, 52
- multiprocessors, 96
- multiprogramming systems, 52
- multiresolution block matching, 55–56
- Mythical Man-Month, The*, 162

N

- National Institute of Standards Technology (NIST), 1, 138
- navigation software, 155
- n-body problem, 170–171
- NCSSs (noncommented source code statements), 129–130
- N-FeatureComparison element, 89, 93
- NIST (National Institute of Standards Technology), 1, 138
- NoiseModel instance, 102
- noncommented source code statements (NCSSs), 129–130
- nonfunctional requirements, 43
- nonvolatile memory, 106
- non-von Neumann architectures, 95
- notation evolution, 46
- null channel delay, 86
- n-version programming, 105

O

OAOO (once and only once) principle, 82, 161
 object attributes, 15
 object codes, 118
 object-oriented analysis. *See* OOA
 object-oriented design. *See* OOD
 object-oriented programming language, 15, 80
 coding standard for, 124
 information hiding in, 113
 metrics for, 137–138
 patching in, 152
 software reuse in, 162
 testing, 142
 objects, 72
 behavior of, 80–81
 state, 80
 Object-Z, 46
 occurrences, 132
 OCP (open-closed principle), 81–82, 161
 once and only once (OAOO) principle, 161
 on-line testing, 175
 OOA (object oriented analysis), 61–62
 low usage of, 68
 state charts in, 51
 vs. structured analysis (SA), 62–64
 survey of, 68–69
 OOD (object-oriented design), 72
 benefits of, 81
 dependency inversion principle (DIP) in, 82
 design patterns, 83
 Liskov substitution principle, 82–83
 once and only once (OAOO) principle in, 82
 open-closed principle in, 81–82
 Q-model in, 84–88
 using UML (unified modeling language), 84
 in visual inspection system, 88–93
 opcodes, 150
 open-closed principle (OCP), 81–82, 161
 operational mode, 47, 79
 operators, 132
 optics, 5
 orthogonality, 49–51
 oscilloscopes, 147
 output devices, 94
 overengineering of algorithms, 2
 overflow conditions, 97
 oversized patch, 152

P

parallel architectures, 95, 96
 parameter passing, 110
 Pareto's principle, 162

Pareto, Vilfredo, 162
 Parnas information hiding, 113
 Parnas partitioning, 18, 72–74
 See also information hiding
 of graphics-rendering software, 73
 in module design, 140
 in software reuse, 160, 162
 partitioning, 72
 Pascal programming language
 See also programming languages
 dynamic memory allocation in, 111
 exception handling in, 113
 lines of code per function point, 136
 pragma pseudo-op in, 115
 patching, 150–152
 pattern languages, 83
 PCI (Peripheral Component Interconnect), 6
 PDLs (program description languages), 54, 56
 perfective maintenance, 159
 performance of software, 12–13
 performance requirements, 43
 Peripheral Component Interconnect (PCI), 6
 personnel management, 169–170
 dealing with difficult people in, 174
 n-body problem, 170–171
 principle-centered leadership in, 173–174
 team management theories, 171–172
 PERT (program evaluation and review technique),
 179–180
 Petri nets, 51–53
 pipelined architectures, 96
 pixel overflow errors, 112
 pixels, 80–81
 plug-and-play systems, 7
 Poisson distribution, 101
 polymorphism, 114
 and late binding, 113
 in object-oriented programming language, 80
 in Z language, 46
 portability of software, 14
 positioning systems, 5
 postintegration software optimization, 153–157
 binary angular measure (BAM), 155
 CPU utilization estimation, 153–154
 execution time estimation, 154
 imprecise computation, 157
 look-up tables, 155–156
 optimizing memory usage in, 157
 scaled numbers, 154–155
 potential volume (V^*), 133
 pragma pseudo-op, 115
 precedence graph, 178
 predicate calculus, 44, 46
 primitives, 117
 probability of failure, 10

- probe effect, 152–153
- probes, 147
- procedural level of cohesion, 16
- procedurally-oriented programming language, 15, 161
- procedural-oriented design, 72
 - data dictionaries, 76–77
 - Parnas partitioning, 72–74
 - structured design (SD), 74–78
- procedures, 15, 113
- process execution time, 86
- process group behavior, 88
- process pair interaction, 87
- process specifications (P-SPECs), 75, 79, 87
- process start period, 86
- process time set, 86
- ProcessDecorator element, 89
- processors, 94–95
- program description languages (PDLs), 54, 56
- program evaluation and review technique (PERT), 179–180
- program length (N), 132
- program level (L), 133
- program vocabulary (n), 132
- program volume (V), 132–133
- programmed I/O, 97
- programmers, 7
- programming, 7
- programming languages, 109–110
 - Ada 95
 - . See Ada 95 programming language
 - assembly language, 115
 - C
 - . See C programming language
 - C++
 - . See C++ programming language
 - call-by-reference parameter passing, 110
 - call-by-value parameter passing, 110
 - dynamic memory allocation, 111
 - exception handling in, 112–113
 - Fortran
 - . See Fortran programming language
 - global variables, 110–111
 - Java, 109, 113, 117–118
 - modularity in, 113
 - object-oriented
 - . See object-oriented programming language
 - parameter passing, 110
 - procedurally-oriented, 15, 161
 - recursion, 111
 - skills testing in, 175
 - typing, 112
- progressive scan, 5
- project managers, 166–167

- projects, 166
- prologues, 115
- propositional logic, 44
- protected variation principle, 162
- prototype hardware, 149–150
- prototyping, 35–36
 - evolutionary, 30
 - of imaging systems, 2
 - in requirements analysis, 41
 - in spiral software model, 29–30
- proximity sensors, 6
- pseudo-codes, 54
 - in collision testing of graphical objects, 56
 - in process specifications, 75
- P-SPECs (process specifications), 75, 79, 87
- Python programming language, 114

Q

- Q-model, 56
 - of compareImage for N-FeatureComparison, 93
 - development of, 85
 - generating, 90
 - levels of analysis in, 87
 - process parameters, 85–87
 - of product classification, 92
- quad trees, 111
- “quick-and-dirty” prototypes, 160

R

- RAD (rapid application delivery), 30
- random access memory (RAM), 106–107
- random test case generation, 140–141
- ranges, 161
- rapid application delivery (RAD), 30
- rapid delivery, 30
- ray tracing, 155
- read-only memory (ROM), 106
- real-time imaging dilemma, 12
- real-time imaging systems, 12
- real-time interval logic (RTIL), 46
- real-valued failure function, 10
- reclamation, 158
- recovery blocks, 102–104
- recursion, 111
- redirecting standard error, 120
- redocumentation, 158
- reduced instruction set computer (RISC), 94
- reengineering, 158
- register types, 115
- regression testing, 143

RejectController element, 89
 reliability function, 10
 reliability of software, 9–10
 Remington Rand, 178
 renovation, 158
 repairability of software, 13
 requirements engineering, 41
 See also software engineering; SRS (software requirements specification)
 JAD (joint application design), 124–126
 survey of, 68–69
 types of requirements, 42–43
 requirements users, 43–44
 resources, control of, 166
 reuse of codes, 2
 reverse engineering, 158
 RGB color, 5
 rigor in software engineering, 15
 RISC (reduced instruction set computer), 94
 risk analysis, 29
 risk factors, 35–36
 risk management and mitigation, 168–169
 ROM (read-only memory), 106
 RS-170/CCIR, 5
 RTIL (real-time interval logic), 46
 Ruby programming language, 114
 run-time codes, 121

S

SA (structured analysis), 59–61
 vs. object oriented analysis (OOA), 62–64
 state charts in, 51
 transitioning to structured design, 74–76
 sabotage, 3
 safety-critical systems, 45
 sampling speed, 6
 SASD (structured analysis and structured design), 59
 problems in imaging applications, 77–78
 real-time extensions of, 78
 satellite positioning systems, 99
 scaled numbers, 154–155
 schedules, control of, 166
 schema calculus, 46
 schemes, 46
 scripting languages, 114
 scrum programming, 33
 SD (structured design), 74
 transitioning from structured analysis to, 74–76
sdb program, 121–122
 SDD (software design description), 26, 71
 SDP (software development plan), 30

second system effect, 162–163
 segmentation, 111
 semiformal methods, in requirements specification, 45
 senior developer, 169
 sensors, 4, 6
 separation of concerns, 15
 sequential level of cohesion, 17
 set theory, 44, 46
 setjmp procedure call, 116
 SIMD (single instruction multiple data), 95–96
 Simonyi, Charles, 123
 simultaneity interval, 86
 sine function, 156
 single instruction multiple data (SIMD), 95–96
 single instruction single data (SISD), 95
 SISD (single instruction single data), 95
 skills tests, 174–175
 slave processors, 105
 soft errors, 106
 software black boxes, 104–105
 software complexity, 130–132
 software conception, 25
 software design, 26
 design activity, 71–72
 fault-tolerant design, 99–107
 hardware considerations in, 93–99
 object-oriented design (OOD), 80–81
 procedural-oriented design, 72
 software design description (SDD), 26, 71
 software development, 26–27, 162
 software development plan (SDP), 30
 software engineering, 1
 anticipation of change, 18
 in building imaging systems, 2
 generality, 20
 incrementality, 20
 misconceptions on, 7–8
 modularity, 15–18
 poor practices in, 2–3
 rigor and formality, 15
 separation of concerns, 15
 vs. software project management, 167
 tools, 7
 traceability, 20–22
 Software Engineering Institute, 34
 software engineers, 1
 responsibilities to stakeholders, 44
 roles of, 3–4
 software integration, 150
 patching, 150–152
 simple strategy in, 150
 software life cycle, 2, 126
 software maintenance, 27
 software methodologies, 23

- software partitioning, 72
- software processes, 23
- software production process
 - coding standards, 123–124
 - documentation, 126–127
 - programming languages in, 109–118
 - reviews and audits, 124–126
 - writing and testing codes in, 118–123
- software productivity, 7
- Software Productivity Research, Inc., 137
- software project management, 165–166
 - assessment of project personnel in, 174–177
 - cost estimation with COCOMO, 180–183
 - managing and mitigating risks in, 168–169
 - personnel management in, 169–174
 - risks in, 168
 - vs. software engineering, 167
 - specialized activities in, 168
 - themes in, 166
 - tracking and reporting progress in, 177–180
- software project manager, 169
- software qualities, 9
 - correctness, 12
 - interoperability, 13
 - maintainability, 13
 - measurement approach, 14
 - performance, 12–13
 - portability, 14
 - reliability, 9–10
 - system-level, 142–144
 - usability, 13
 - verifiability, 14
- software reengineering process model, 157–158
- software requirements specification. *See* SRS
- software reuse, 159–160
 - achieving, 160–161
 - avoiding, 160
 - in object-oriented languages, 162
 - Pareto's principle, 162
 - in procedural languages, 161
- software simulators, 149
- software specifications, formal methods in, 44–46
- software standards, 36–39
 - DOD-STD-2167A, 37–38
 - DOD-STD-498, 37
 - ISO 9000-9003, 38–39
 - ISO/IEC 12207, 40
- software testing, 1
 - design plans for, 143–144
 - extended syntax and semantic checking in, 120–121
 - failures in, 2
 - of object-oriented software, 142
 - role of, 139
 - symbolic debugging in, 121–122
 - unit-level, 120, 139–142
 - in waterfall software life cycle, 27
- software test requirements specification (STRS), 27
- source codes. *See* codes
- spatial fault tolerance, 99
- specification of imaging systems, 53–54, 56–57
 - collision testing of graphical objects, 56
 - Markov random fields image reconstruction, 57–58
 - multiresolution block matching, 55–56
 - recommendations on, 64
- spiral model, 28–30
- square root of negative errors, 112
- SRAM (static random access memory), 98
- SRS (software requirements specification), 25–26
 - See also* requirements engineering
 - current practices in, 68–69
 - design activity in, 71–72
 - FSM (finite state machine) in, 78
 - goal of, 41
 - and lightweight methodologies, 32–33
 - organizing, 64–66
 - requirements validation and review, 67–68
 - writing good requirements in, 66–67
- stakeholders, 43–44
- stamp coupling, 18
- state, 80
- state charts, 49–51
- state transition diagrams (STDs), 46, 59
- statements, 15, 132
- static random access memory (SRAM), 98
- static types, 115
- STDs (state transition diagrams), 46, 59
- storage oscilloscopes, 147
- stress testing, 143–144
- string types, 112, 118
- STRS (software test requirements specification), 27
- structured analysis. *See* SA
- structured analysis and structured design. *See* SASD
- subroutines, 113
- symbolic debugging, 121–122, 152
- synchronization error, 12
- synchronous clusters, 87
- syntax errors, 120
- system integration, 145
- system integration tools, 146–150
 - hardware prototypes, 149–150
 - in-circuit emulators, 149
 - logic analyzers, 147–149
 - multimeters, 147
 - oscilloscopes, 147
 - software simulators, 149

system unification, 145–146
 system verification, 146
 system-level testing, 142–144
 systolic processors, 96

T

tangent function, 156
 Taylor series expansion, 157
 team management, theories on, 171–172
 technical lead, 169
 temporal level of cohesion, 16
 terminal states, 47
 terminators, 74
 test cases, 146
 test cases, limit of number of, 132
 test engineers, 43–44
 test-first coding, 122
 testing, 139
 theorem proving, 45
 Theory W management, 172
 Theory X management, 171
 Theory Y management, 171
 Theory Z management, 171–172
 thinning, 111
 thousands of lines of code (KLOC), 129–130, 181
 throwaway prototypes, 160
 timing code, 148–149
 timing instructions, 148
 traceability, 20–22
 transition function, 47
 transitions, 105
 translation, 46
 transmission errors, 58
 transputers, 96
 traps, 97
 trigonometric functions, look-up table for, 156
 “try, throw, catch, finally” approach, 113
 typing, 112

U

UML (unified modeling language), 34
 in notation evolution, 46
 in object-oriented design, 81, 84
 and Q model, 88
 in requirements specification, 45
 uncertainty principle, 152–153
 unified process model (UPM), 34
 unit-level testing, 120
 See also software testing
 black box testing, 139–141
 white box testing, 141–142

universal quantification, 42
 Universal Serial Bus (USB), 6
 Unix C compiler, 119–120
 Unix operating system, 115
 symbolic debugging in, 121
 UPM (unified process model), 34
 usability of software, 13
 USB (Universal Serial Bus), 6
 use cases, 62, 91
 utility routines, 160

V

V model, 28
 validation, 139
 variable declarations, 15
 variable scan, 5
 variables, 112
 verifiability of software, 14
 version control software, 26–27
 very long instruction word (VLIW) architectures, 96
 video capture cards, 6
 virtual machine, 117
 Visual BASIC programming language, 109
 visual inspection system, 5–7
 See also imaging system
 applications of, 6–7
 behavior of, 47
 context diagram for, 61
 functional representation of, 56–57
 noise reduction code for, 131
 object-oriented design in, 88–93
 partial CPM diagram for, 179
 partial Gantt chart for, 178
 partial PERT chart for, 180
 problems in using SASD (structured analysis and structured design), 77–78
 real-time, 12
 sample test log for, 146
 use case diagram of, 62
 using Kalman filter in, 99–102
 VLIW (very long instruction word) architectures, 172
 volatile types, 115
 von Neumann architecture, 157
 voting schemes, 99

W

walkthroughs, 141
 waterfall software life cycle, 24
 See also life cycle models

- backtracking transitions in, 27–28
- phases in, 25
- requirements specification, 25–26
- software conception, 25
- software design, 26
- software development, 26–27
- software maintenance, 27
- summary of, 28
- testing, 27
- wave-front processors, 96
- WBS (work breakdown structure), 65
- win-lose management paradigm, 172
- win-win management paradigm, 172
- work breakdown structure (WBS), 65
- worst-case execution time (A^t), 153
- worst-case testing, 141
- wraparound errors, 112

X

- XP (extreme programming), 33, 122
- X-Y positioning table, 6

Y

- Yourdon's modern structured analysis, 59

Z

- Z language, 46
- zero-sum management paradigm, 172