Software Engineering

Theory and *Practice* | Fourth Edition

Shari Lawrence Pfleeger
Joanne M. Atlee

Vice President and Editorial Director, ECS: Marcia J. Horton
Executive Editor: Tracy Dunkelberger
Assistant Editor: Melinda Haggerty
Director of Team-Based Project Management: Vince O'Brien
Senior Managing Editor: Scott Disanno
Production Liaison: Jane Bonnell
Production Editor: Pavithra Jayapaul, TexTech
Senior Operations Specialist: Alan Fischer
Operations Specialist: Lisa McDowell
Marketing Manager: Erin Davis
Marketing Assistant: Mack Patterson
Art Director: Kenny Beck
Cover Designer: Kristine Carney
Cover Image: [credit to come]
Art Editor: Greg Dulles
Media Editor: Daniel Sandin
Media Project Manager: John M. Cassar
Composition/Full-Service Project Management: TexTech International Pvt. Ltd.

Library of Congress Cataloging-in-Publication Data
Pfleeger, Shari Lawrence.
Software engineering : theory and practice / Shari Lawrence Pfleeger, Joanne M. Atlee. — 4th ed.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-13-606169-4 (alk. paper)
ISBN-10: 0-13-606169-9 (alk. paper)
1. Software engineering. I. Atlee, Joanne M. II. Title.
QA76.758.P49 2010
005.1—dc22 2008051400

Prentice Hall
is an imprint of

PEARSON

www.pearsonhighered.com

10 9 8 7 6 5 4 3 2 1

ISBN-13: 978-0-13-606169-4
ISBN-10: 0-13-606169-9

"From so much loving and journeying, books emerge."
Pablo Neruda

To Florence Rogart for providing the spark; to Norma Mertz for helping to keep the flame burning.

S.L.P.

To John Gannon, posthumously, for his integrity, inspiration, encouragement, friendship, and the legacy he has left to all of us in software engineering.

J.M.A.

Preface

BRIDGING THE GAP BETWEEN RESEARCH AND PRACTICE

Software engineering has come a long way since 1968, when the term was first used at a NATO conference. And software itself has entered our lives in ways that few had anticipated, even a decade ago. So a firm grounding in software engineering theory and practice is essential for understanding how to build good software and for evaluating the risks and opportunities that software presents in our everyday lives. This text represents the blending of the two current software engineering worlds: that of the practitioner, whose main focus is to build high-quality products that perform useful functions, and that of the researcher, who strives to find ways to improve the quality of products and the productivity of those who build them. Edsgar Dykstra continually reminded us that rigor in research and practice tests our understanding of software engineering and helps us to improve our thinking, our approaches, and ultimately our products.

It is in this spirit that we have enhanced our book, building an underlying framework for this questioning and improvement. In particular, this fourth edition contains extensive material about how to abstract and model a problem, and how to use models, design principles, design patterns, and design strategies to create appropriate solutions. Software engineers are more than programmers following instructions, much as chefs are more than cooks following recipes. There is an art to building good software, and the art is embodied in understanding how to abstract and model the essential elements of a problem and then use those abstractions to design a solution. We often hear good developers talk about "elegant" solutions, meaning that the solution addresses the heart of the problem, such that not only does the software solve the problem in its current form but it can also be modified as the problem evolves over time. In this way, students learn to blend research with practice and art with science, to build solid software.

The science is always grounded in reality. Designed for an undergraduate software engineering curriculum, this book paints a pragmatic picture of software engineering research and practices so that students can apply what they learn directly to the real-world problems they are trying to solve. Examples speak to a student's limited experience but illustrate clearly how large software development projects progress from need to idea to reality. The examples represent the many situations that readers are likely to experience: large projects and small, "agile" methods and highly structured ones, object-oriented and procedural approaches, real-time and transaction processing, development and maintenance situations.

The book is also suitable for a graduate course offering an introduction to software engineering concepts and practices, or for practitioners wishing to expand their

knowledge of the subject. In particular, Chapters 12, 13, and 14 present thought-provoking material designed to interest graduate students in current research topics.

KEY FEATURES

This text has many key features that distinguish it from other books.

- Unlike other software engineering books that consider measurement and modeling as separate issues, this book blends measurement and modeling with the more general discussion of software engineering. That is, measurement and modeling are considered as an integral part of software engineering strategies, rather than as separate disciplines. Thus, students learn how to abstract and model, and how to involve quantitative assessment and improvement in their daily activities. They can use their models to understand the important elements of the problems they are solving as well as the solution alternatives; they can use measurement to evaluate their progress on an individual, team, and project basis.
- Similarly, concepts such as reuse, risk management, and quality engineering are embedded in the software engineering activities that are affected by them, instead of being treated as separate issues.
- The current edition addresses the use of agile methods, including extreme programming. It describes the benefits and risks of giving developers more autonomy and contrasts this agility with more traditional approaches to software development.
- Each chapter applies its concepts to two common examples: one that represents a typical information system, and another that represents a real-time system. Both examples are based on actual projects. The information system example describes the software needed to determine the price of advertising time for a large British television company. The real-time system is the control software for the Ariane-5 rocket; we look at the problems reported, and explore how software engineering techniques could have helped to locate and avoid some of them. Students can follow the progress of two typical projects, seeing how the various practices described in the book are merged into the technologies used to build systems.
- At the end of every chapter, the results are expressed in three ways: what the content of the chapter means for development teams, what it means for individual developers, and what it means for researchers. The student can easily review the highlights of each chapter, and can see the chapter's relevance to both research and practice.
- The Companion Web site can be found at www.prenhall.com/pfleeger. It contains current examples from the literature and examples of real artifacts from real projects. It also includes links to Web pages for relevant tool and method vendors. It is here that students can find real requirements documents, designs, code, test plans, and more. Students seeking additional, in-depth information are pointed to reputable, accessible publications and Web sites. The Web pages are updated regularly to keep the material in the textbook current, and include a facility for feedback to the author and the publisher.
- A Student Study Guide is available from your local Pearson Sales Representative.

- PowerPoint slides and a full solutions manual are available on the Instructor Resource Center. Please contact your local Pearson Sales Representative for access information.
- The book is replete with case studies and examples from the literature. Many of the one-page case studies shown as sidebars in the book are expanded on the Web page. The student can see how the book's theoretical concepts are applied to real-life situations.
- Each chapter ends with thought-provoking questions about policy, legal, and ethical issues in software engineering. Students see software engineering in its social and political contexts. As with other sciences, software engineering decisions must be viewed in terms of the people their consequences will affect.
- Every chapter addresses both procedural and object-oriented development. In addition, Chapter 6 on design explains the steps of an object-oriented development process. We discuss several design principles and use object-oriented examples to show how designs can be improved to incorporate these principles.
- The book has an annotated bibliography that points to many of the seminal papers in software engineering. In addition, the Web page points to annotated bibliographies and discussion groups for specialized areas, such as software reliability, fault tolerance, computer security, and more.
- Each chapter includes a description of a term project, involving development of software for a mortgage processing system. The instructor may use this term project, or a variation of it, in class assignments.
- Each chapter ends with a list of key references for the concepts in the chapter, enabling students to find in-depth information about particular tools and methods discussed in the chapter.
- This edition includes examples highlighting computer security. In particular, we emphasize designing security in, instead of adding it during coding or testing.

CONTENTS AND ORGANIZATION

This text is organized in three parts. The first part motivates the reader, explaining why knowledge of software engineering is important to practitioners and researchers alike. Part I also discusses the need for understanding process issues, for making decisions about the degree of "agility" developers will have, and for doing careful project planning. Part II walks through the major steps of development and maintenance, regardless of the process model used to build the software: eliciting, modeling, and checking the requirements; designing a solution to the problem; writing and testing the code; and turning it over to the customer. Part III focuses on evaluation and improvement. It looks at how we can assess the quality of our processes and products, and how to take steps to improve them.

Chapter 1: Why Software Engineering?

In this chapter we address our track record, motivating the reader and highlighting where in later chapters certain key issues are examined. In particular, we look at

Wasserman's key factors that help define software engineering: abstraction, analysis and design methods and notations, modularity and architecture, software life cycle and process, reuse, measurement, tools and integrated environments, and user interface and prototyping. We discuss the difference between computer science and software engineering, explaining some of the major types of problems that can be encountered, and laying the groundwork for the rest of the book. We also explore the need to take a systems approach to building software, and we introduce the two common examples that will be used in every chapter. It is here that we introduce the context for the term project.

Chapter 2: Modeling the Process and Life Cycle

In this chapter, we present an overview of different types of process and life-cycle models, including the waterfall model, the V-model, the spiral model, and various prototyping models. We address the need for agile methods, where developers are given a great deal of autonomy, and contrast them with more traditional software development processes. We also describe several modeling techniques and tools, including systems dynamics and other commonly-used approaches. Each of the two common examples is modeled in part with some of the techniques introduced here.

Chapter 3: Planning and Managing the Project

Here, we look at project planning and scheduling. We introduce notions such as activities and milestones, work breakdown structure, activity graphs, risk management, and costs and cost estimation. Estimation models are used to estimate the cost and schedule of the two common examples. We focus on actual case studies, including management of software development for the F-16 airplane and for Digital's alpha AXP programs.

Chapter 4: Capturing the Requirements

This chapter emphasizes the critical roles of abstraction and modeling in good software engineering. In particular, we use models to tease out misunderstandings and missing details in provided requirements, as well as to communicate requirements to others. We explore a number of different modeling paradigms, study example notations for each paradigm, discuss when to use each paradigm, and provide advice about how to make particular modeling and abstraction decisions. We discuss different sources and different types of requirements (functional requirements vs. quality requirements vs. design constraints), explain how to write testable requirements, and describe how to resolve conflicts. Other topics discussed include requirements elicitation, requirements documentation, requirements reviews, requirements quality and how to measure it, and an example of how to select a specification method. The chapter ends with application of some of the methods to the two common examples.

Chapter 5: Designing the Architecture

This chapter on software architecture has been completely revised for the fourth edition. It begins by describing the role of architecture in the software design process and in the

larger development process. We examine the steps involved in producing the architecture, including modeling, analysis, documentation, and review, resulting in the creation of a Software Architecture Document that can be used by program designers in describing modules and interfaces. We discuss how to decompose a problem into parts, and how to use different views to examine the several aspects of the problem so as to find a suitable solution. Next, we focus on modeling the solution using one or more architectural styles, including pipe-and-filter, peer-to-peer, client–server, publish–subscribe, repositories, and layering. We look at combining styles and using them to achieve quality goals, such as modifiability, performance, security, reliability, robustness, and usability.

Once we have an initial architecture, we evaluate and refine it. In this chapter, we show how to measure design quality and to use evaluation techniques in safety analysis, security analysis, trade-off analysis, and cost–benefit analysis to select the best architecture for the customer's needs. We stress the importance of documenting the design rationale, validating and verifying that the design matches the requirements, and creating an architecture that suits the customer's product needs. Towards the end of the chapter, we examine how to build a product-line architecture that allows a software provider to reuse the design across a family of similar products. The chapter ends with an architectural analysis of our information system and real-time examples.

Chapter 6: Designing the Modules

Chapter 6, substantially revised in this edition, investigates how to move from a description of the system architecture to descriptions of the design's individual modules. We begin with a discussion of the design process and then introduce six key design principles to guide us in fashioning modules from the architecture: modularity, interfaces, information hiding, incremental development, abstraction, and generality. Next, we take an in-depth look at object-oriented design and how it supports our six principles. Using a variety of notations from the Unified Modeling Language, we show how to represent multiple aspects of module functionality and interaction, so that we can build a robust and maintainable design. We also describe a collection of design patterns, each with a particular purpose, and demonstrate how they can be used to reinforce the design principles. Next, we discuss global issues such as data management, exception handling, user interfaces, and frameworks; we see how consistency and clarity of approach can lead to more effective designs.

Taking a careful look at object-oriented measurement, we apply some of the common object-oriented metrics to a service station example. We note how changes in metrics values, due to changes in the design, can help us decide how to allocate resources and search for faults. Finally, we apply object-oriented concepts to our information systems and real-time examples.

Chapter 7: Writing the Programs

In this chapter, we address code-level design decisions and the issues involved in implementing a design to produce high-quality code. We discuss standards and procedures, and suggest some simple programming guidelines. Examples are provided in a variety of languages, including both object-oriented and procedural. We discuss the need for

program documentation and an error-handling strategy. The chapter ends by applying some of the concepts to the two common examples.

Chapter 8: Testing the Programs

In this chapter, we explore several aspects of testing programs. We distinguish conventional testing approaches from the cleanroom method, and we look at how to test a variety of systems. We present definitions and categories of software problems, and we discuss how orthogonal defect classification can make data collection and analysis more effective. We then explain the difference between unit testing and integration testing. After introducing several automated test tools and techniques, we explain the need for a testing life cycle and how the tools can be integrated into it. Finally, the chapter applies these concepts to the two common examples.

Chapter 9: Testing the System

We begin with principles of system testing, including reuse of test suites and data, and the need for careful configuration management. Concepts introduced include function testing, performance testing, acceptance testing and installation testing. We look at the special needs of testing object-oriented systems. Several test tools are described, and the roles of test team members are discussed. Next, we introduce the reader to software reliability modeling, and we explore issues of reliability, maintainability, and availability. The reader learns how to use the results of testing to estimate the likely characteristics of the delivered product. The several types of test documentation are introduced, too, and the chapter ends by describing test strategies for the two common examples.

Chapter 10: Delivering the System

This chapter discusses the need for training and documentation, and presents several examples of training and documents that could accompany the information system and real-time examples.

Chapter 11: Maintaining the System

In this chapter, we address the results of system change. We explain how changes can occur during the system's life cycle, and how system design, code, test process, and documentation must accommodate them. Typical maintenance problems are discussed, as well as the need for careful configuration management. There is a thorough discussion of the use of measurement to predict likely changes, and to evaluate the effects of change. We look at reengineering and restructuring in the overall context of rejuvenating legacy systems. Finally, the two common examples are evaluated in terms of the likelihood of change.

Chapter 12: Evaluating Products, Processes, and Resources

Since many software engineering decisions involve the incorporation and integration of existing components, this chapter addresses ways to evaluate processes and products. It discusses the need for empirical evaluation and gives several examples to show how measurement can be used to establish a baseline for quality and productivity. We look at

several quality models, how to evaluate systems for reusability, how to perform post-mortems, and how to understand return on investment in information technology. These concepts are applied to the two common examples.

Chapter 13: Improving Predictions, Products, Processes, and Resources

This chapter builds on Chapter 11 by showing how prediction, product, process, and resource improvement can be accomplished. It contains several in-depth case studies to show how prediction models, inspection techniques, and other aspects of software engineering can be understood and improved using a variety of investigative techniques. This chapter ends with a set of guidelines for evaluating current situations and identifying opportunities for improvement.

Chapter 14: The Future of Software Engineering

In this final chapter, we look at several open issues in software engineering. We revisit Wasserman's concepts to see how well we are doing as a discipline. We examine several issues in technology transfer and decision-making, to determine if we do a good job at moving important ideas from research to practice. Finally, we examine controversial issues, such as licensing of software engineers as professional engineers and the trend towards more domain-specific solutions and methods.

ACKNOWLEDGMENTS

Books are written as friends and family provide technical and emotional support. It is impossible to list here all those who helped to sustain us during the writing and revising, and we apologize in advance for any omissions. Many thanks to the readers of earlier editions, whose careful scrutiny of the text generated excellent suggestions for correction and clarification. As far as we know, all such suggestions have been incorporated into this edition. We continue to appreciate feedback from readers, positive or negative.

Carolyn Seaman (University of Maryland–Baltimore Campus) was a terrific reviewer of the first edition, suggesting ways to clarify and simplify, leading to a tighter, more understandable text. She also prepared most of the solutions to the exercises, and helped to set up an early version of the book's Web site. I am grateful for her friendship and assistance. Yiqing Liang and Carla Valle updated the Web site and added substantial new material for the second edition; Patsy Ann Zimmer (University of Waterloo) revised the Web site for the third edition, particularly with respect to modeling notations and agile methods.

We owe a huge thank-you to Forrest Shull (Fraunhofer Center–Maryland) and Roseanne Tesoriero (Washington College), who developed the initial study guide for this book; to Maria Vieira Nelson (Catholic University of Minas Gerais, Brazil), who revised the study guide and the solutions manual for the third edition; and to Eduardo S. Barrenechea (University of Waterloo) for updating the materials for the fourth edition. Thanks, too, to Hossein Saiedian (University of Kansas) for preparing the PowerPoint presentation for the fourth edition. We are also particularly indebted to Guilherme Travassos (Federal University of Rio de Janeiro) for the use of material that he developed

with Pfleeger at the University of Maryland–College Park, and that he enriched and expanded considerably for use in subsequent classes.

Helpful and thoughtful reviewers for all four editions included Barbara Kitchenham (Keele University, UK), Bernard Woolfolk (Lucent Technologies), Ana Regina Cavalcanti da Rocha (Federal University of Rio de Janeiro), Frances Uku (University of California at Berkeley), Lee Scott Ehrhart (MITRE), Laurie Werth (University of Texas), Vickie Almstrum (University of Texas), Lionel Briand (Simula Research, Norway), Steve Thibaut (University of Florida), Lee Wittenberg (Kean College of New Jersey), Philip Johnson (University of Hawaii), Daniel Berry (University of Waterloo, Canada), Nancy Day (University of Waterloo), Jianwei Niu (University of Waterloo), Chris Gorringe (University of East Anglia, UK), Ivan Aaen (Aalborg University), Damla Turget (University of Central Florida), Laurie Williams (North Carolina State University), Ernest Sibert (Syracuse University), Allen Holliday (California State University, Fullerton) David Rine (George Mason University), Anthony Sullivan (University of Texas, Dallas), David Chesney (University of Michigan, Ann Arbor), Ye Duan (Missouri University), Rammohan K. Ragade (Kentucky University), and several anonymous reviewers provided by Prentice Hall. Discussions with Greg Hislop (Drexel University), John Favaro (Intecs Sistemi, Italy), Filippo Lanubile (Università di Bari, Italy), John d'Ambra (University of New South Wales, Australia), Chuck Howell (MITRE), Tim Vieregge (U.S. Army Computer Emergency Response Team) and James and Suzanne Robertson (Atlantic Systems Guild, UK) led to many improvements and enhancements.

Thanks to Toni Holm and Alan Apt, who made the third edition of the book's production interesting and relatively painless. Thanks, too, to James and Suzanne Robertson for the use of the Piccadilly example, and to Norman Fenton for the use of material from our software metrics book. We are grateful to Tracy Dunkelberger for encouraging us in producing this fourth edition; we appreciate both her patience and her professionalism. Thanks, too, to Jane Bonnell and Pavithra Jayapaul for seamless production.

Many thanks to the publishers of several of the figures and examples for granting permission to reproduce them here. The material from *Complete Systems Analysis* (Robertson and Robertson 1994) and *Mastering the Requirements Process* (Robertson and Robertson 1999) is drawn from and used with permission from Dorset House Publishing, at www.dorsethouse.com; all rights reserved. The article in Exercise 1.1 is reproduced from the *Washington Post* with permission from the Associated Press. Figures 2.15 and 2.16 are reproduced from Barghouti et al. (1995) by permission of John Wiley and Sons Limited. Figures 12.14 and 12.15 are reproduced from Rout (1995) by permission of John Wiley and Sons Limited.

Figures and tables in Chapters 2, 3, 4, 5, 9, 11, 12, and 14 that are noted with an IEEE copyright are reprinted with permission of the Institute of Electrical and Electronics Engineers. Similarly, the three tables in Chapter 14 that are noted with an ACM copyright are reprinted with permission of the Association of Computing Machinery. Table 2.1 and Figure 2.11 from Lai (1991) are reproduced with permission from the Software Productivity Consortium. Figures 8.16 and 8.17 from Graham (1996a) are reprinted with permission from Dorothy R. Graham. Figure 12.11 and Table 12.2 are adapted from Liebman (1994) with permission from the Center for Science in the

Public Interest, 1875 Connecticut Avenue NW, Washington DC. Tables 8.2, 8.3, 8.5, and 8.6 are reproduced with permission of The McGraw-Hill Companies. Figures and examples from Shaw and Garlan (1996), Card and Glass (1990), Grady (1997), and Lee and Tepfenhart (1997) are reproduced with permission from Prentice Hall.

Tables 9.3, 9.4, 9.6, 9.7, 13.1, 13.2, 13.3, and 13.4, as well as Figures 1.15, 9.7. 9.8, 9.9, 9.14, 13.1, 13.2, 13.3, 13.4, 13.5, 13.6, and 13.7 are reproduced or adapted from Fenton and Pfleeger (1997) in whole or in part with permission from Norman Fenton. Figures 3.16, 5.19, and 5.20 are reproduced or adapted from Norman Fenton's course notes, with his kind permission.

We especially appreciate our employers, the RAND Corporation and the University of Waterloo, respectively, for their encouragement.[1] And we thank our friends and family, who offered their kindness, support, and patience as the book-writing stole time ordinarily spent with them. In particular, Shari Lawrence Pfleeger is grateful to Manny Lawrence, the manager of the real Royal Service Station, and to his bookkeeper, Bea Lawrence, not only for working with her and her students on the specification of the Royal system, but also for their affection and guidance in their other job: as her parents. Jo Atlee gives special thanks to her parents, Nancy and Gary Atlee, who have supported and encouraged her in everything she has done (and attempted); and to her colleagues and students, who graciously took on more than their share of work during the major writing periods. And, most especially, we thank Charles Pfleeger and Ken Salem, who were constant and much-appreciated sources of support, encouragement, and good humor.

Shari Lawrence Pfleeger
Joanne M. Atlee

[1]Please note that this book is not a product of the RAND Corporation and has not undergone RAND's quality assurance process. The work represents us as authors, not as employees of our respective institutions.

About the Authors

Shari Lawrence Pfleeger (Ph.D., Information Technology and Engineering, George Mason University; M.S., Planning, Pennsylvania State University; M.A., Mathematics, Pennsylvania State University; B.A., Mathematics, Harpur College) is a senior information scientist at the RAND Corporation. Her current research focuses on policy and decision-making issues that help organizations and government agencies understand whether and how information technology supports their missions and goals. Her work at RAND has involved assisting clients in creating software measurement programs, supporting government agencies in defining information assurance policies, and supporting decisions about cyber security and homeland security.

Prior to joining RAND, she was the president of Systems/Software, Inc., a consultancy specializing in software engineering and technology. She has been a visiting professor at City University (London) and the University of Maryland and was the founder and director of Howard University's Center for Research in Evaluating Software Technology. The author of many textbooks on software engineering and computer security, Pfleeger is well known for her work in empirical studies of software engineering and for her multidisciplinary approach to solving information technology problems. She has been associate editor-in-chief of *IEEE Software*, associate editor of *IEEE Transactions on Software Engineering*, associate editor of *IEEE Security and Privacy*, and a member of the IEEE Computer Society Technical Council on Software Engineering. A frequent speaker at conferences and workshops, Pfleeger has been named repeatedly by the *Journal of Systems and Software* as one of the world's top software engineering researchers.

Joanne M. Atlee (Ph.D. and M.S., Computer Science, University of Maryland; B.S., Computer Science and Physics, College of William and Mary; P.Eng.) is an Associate Professor in the School of Computer Science at the University of Waterloo. Her research focuses on software modeling, documentation, and analysis. She is best known for her work on model checking software requirements specifications. Other research interests include model-based software engineering, modular software development, feature interactions, and cost-benefit analysis of formal software development techniques. Atlee serves on the editorial boards for *IEEE Transactions on Software Engineering, Software and Systems Modeling*, and the *Requirements Engineering Journal* and is Vice Chair of the International Federation for Information Processing (IFIP) Working Group 2.9, an international group of researchers working on advances in software requirements engineering. She is Program Co-Chair for the 31st International Conference on Software Engineering (ICSE'09).

Atlee also has strong interests in software engineering education. She was the founding Director of Waterloo's Bachelor's program in Software Engineering. She

served as a member of the Steering Committee for the ACM/IEEE-CS Computing Curricula—Software Engineering (CCSE) volume, which provides curricular guidelines for undergraduate programs in software engineering. She also served on a Canadian Engineering Qualifications Board committee whose mandate is to set a software engineering syllabus, to offer guidance to provincial engineering associations on what constitutes acceptable academic qualifications for licensed Professional Engineers who practice software engineering.

SOFTWARE ENGINEERING

1 Why Software Engineering?

In this chapter, we look at

- what we mean by software engineering
- software engineering's track record
- what we mean by good software
- why a systems approach is important
- how software engineering has changed since the 1970s

Software pervades our world, and we sometimes take for granted its role in making our lives more comfortable, efficient, and effective. For example, consider the simple tasks involved in preparing toast for breakfast. The code in the toaster controls how brown the bread will get and when the finished product pops up. Programs control and regulate the delivery of electricity to the house, and software bills us for our energy usage. In fact, we may use automated programs to pay the electricity bill, to order more groceries, and even to buy a new toaster! Today, software is working both explicitly and behind the scenes in virtually all aspects of our lives, including the critical systems that affect our health and well-being. For this reason, software engineering is more important than ever. Good software engineering practices must ensure that software makes a positive contribution to how we lead our lives.

This book highlights the key issues in software engineering, describing what we know about techniques and tools, and how they affect the resulting products we build and use. We will look at both theory and practice: what we know and how it is applied in a typical software development or maintenance project. We will also examine what we do not yet know, but what would be helpful in making our products more reliable, safe, useful, and accessible.

We begin by looking at how we analyze problems and develop solutions. Then we investigate the differences between computer science problems and engineering ones. Our ultimate goal is to produce solutions incorporating high-quality software, and we consider characteristics that contribute to the quality.

We also look at how successful we have been as developers of software systems. By examining several examples of software failure, we see how far we have come and how much farther we must go in mastering the art of quality software development.

Next, we look at the people involved in software development. After describing the roles and responsibilities of customers, users, and developers, we turn to a study of the system itself. We see that a system can be viewed as a group of objects related to a set of activities and enclosed by a boundary. Alternatively, we look at a system with an engineer's eye; a system can be developed much as a house is built. Having defined the steps in building a system, we discuss the roles of the development team at each step.

Finally, we discuss some of the changes that have affected the way we practice software engineering. We present Wasserman's eight ideas to tie together our practices into a coherent whole.

1.1 WHAT IS SOFTWARE ENGINEERING?

As software engineers, we use our knowledge of computers and computing to help solve problems. Often the problem with which we are dealing is related to a computer or an existing computer system, but sometimes the difficulties underlying the problem have nothing to do with computers. Therefore, it is essential that we first understand the nature of the problem. In particular, we must be very careful not to impose computing machinery or techniques on every problem that comes our way. We must solve the problem first. Then, if need be, we can use technology as a tool to implement our solution. For the remainder of this book, we assume that our analysis has shown that some kind of computer system is necessary or desirable to solve a particular problem at hand.

Solving Problems

Most problems are large and sometimes tricky to handle, especially if they represent something new that has never been solved before. So we must begin investigating a problem by **analyzing** it, that is, by breaking it into pieces that we can understand and try to deal with. We can thus describe the larger problem as a collection of small problems and their interrelationships. Figure 1.1 illustrates how analysis works. It is important to remember that the relationships (the arrows in the figure, and the relative positions of the subproblems) are as essential as the subproblems themselves. Sometimes, it is the relationships that hold the clue to how to solve the larger problem, rather than simply the nature of the subproblems.

Once we have analyzed the problem, we must construct our solution from components that address the problem's various aspects. Figure 1.2 illustrates this reverse process: **Synthesis** is the putting together of a large structure from small building blocks. As with analysis, the composition of the individual solutions may be as challenging as the process of finding the solutions. To see why, consider the process of writing a novel. The dictionary contains all the words that you might want to use in your writing. But the most difficult part of writing is deciding how to organize and compose the words into sentences, and likewise the sentences into paragraphs and chapters to form the complete book. Thus, any problem-solving technique must have two parts: analyzing the problem to determine its nature, and then synthesizing a solution based on our analysis.

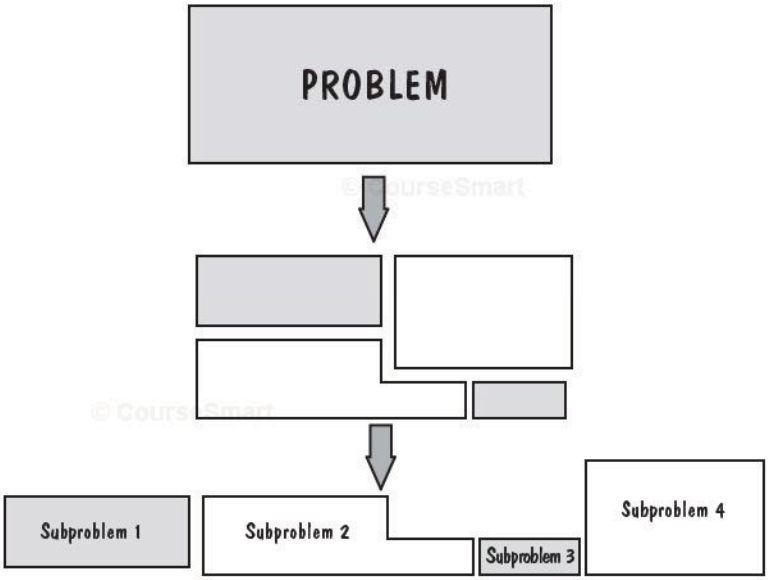


FIGURE 1.1 The process of analysis.

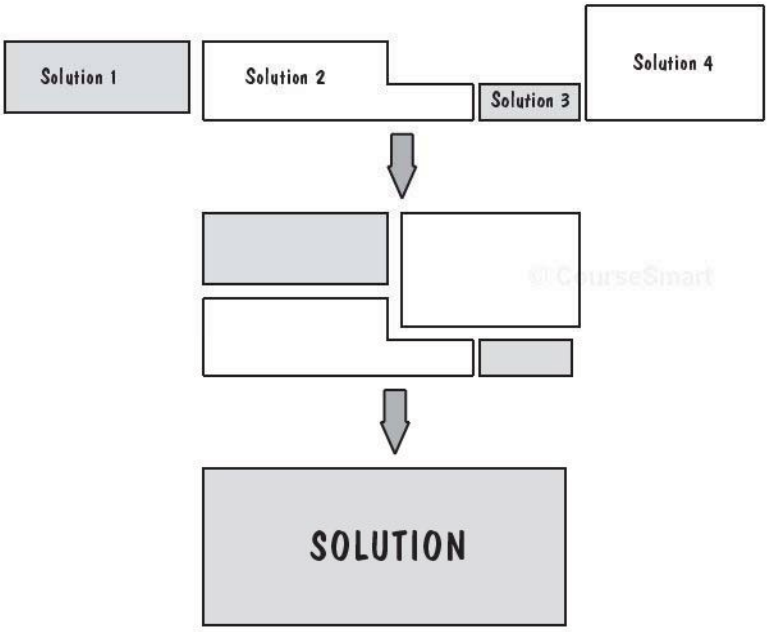


FIGURE 1.2 The process of synthesis.

To help us solve a problem, we employ a variety of methods, tools, procedures, and paradigms. A **method** or **technique** is a formal procedure for producing some result. For example, a chef may prepare a sauce using a sequence of ingredients combined in a carefully timed and ordered way so that the sauce thickens but does not curdle or separate. The procedure for preparing the sauce involves timing and ingredients but may not depend on the type of cooking equipment used.

A **tool** is an instrument or automated system for accomplishing something in a better way. This "better way" can mean that the tool makes us more accurate, more efficient, or more productive or that it enhances the quality of the resulting product. For example, we use a typewriter or a keyboard and printer to write letters because the resulting documents are easier to read than our handwriting. Or we use a pair of scissors as a tool because we can cut faster and straighter than if we were tearing a page. However, a tool is not always necessary for making something well. For example, a cooking technique can make a sauce better, not the pot or spoon used by the chef.

A **procedure** is like a recipe: a combination of tools and techniques that, in concert, produce a particular product. For instance, as we will see in later chapters, our test plans describe our test procedures; they tell us which tools will be used on which data sets under which circumstances so we can determine whether our software meets its requirements.

Finally, a **paradigm** is like a cooking style; it represents a particular approach or philosophy for building software. Just as we can distinguish French cooking from Chinese cooking, so too do we distinguish paradigms like object-oriented development from procedural ones. One is not better than another; each has its advantages and disadvantages, and there may be situations when one is more appropriate than another.

Software engineers use tools, techniques, procedures, and paradigms to enhance the quality of their software products. Their aim is to use efficient and productive approaches to generate effective solutions to problems. In the chapters that follow, we will highlight particular approaches that support the development and maintenance activities we describe. An up-to-date set of pointers to tools and techniques is listed in this book's associated home page on the World Wide Web.

Where Does the Software Engineer Fit In?

To understand how a software engineer fits into the computer science world, let us look to another discipline for an example. Consider the study of chemistry and its use to solve problems. The chemist investigates chemicals: their structure, their interactions, and the theory behind their behavior. Chemical engineers apply the results of the chemist's studies to a variety of problems. Chemistry as viewed by chemists is the object of study. On the other hand, for a chemical engineer, chemistry is a tool to be used to address a general problem (which may not even be "chemical" in nature).

We can view computing in a similar light. We can concentrate on the computers and programming languages, or we can view them as tools to be used in designing and implementing a solution to a problem. Software engineering takes the latter view, as shown in Figure 1.3. Instead of investigating hardware design or proving theorems about how algorithms work, a software engineer focuses on the computer as a problem-solving tool. We will see later in this chapter that a software engineer works with the

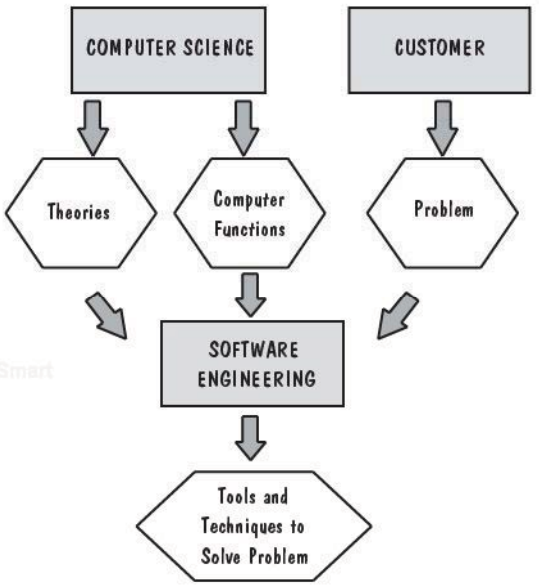


FIGURE 1.3 The relationship between computer science and software engineering.

functions of a computer as part of a general solution, rather than with the structure or theory of the computer itself.

1.2 HOW SUCCESSFUL HAVE WE BEEN?

Writing software is an art as well as a science, and it is important for you as a student of computer science to understand why. Computer scientists and software engineering researchers study computer mechanisms and theorize about how to make them more productive or efficient. However, they also design computer systems and write programs to perform tasks on those systems, a practice that involves a great deal of art, ingenuity, and skill. There may be many ways to perform a particular task on a particular system, but some are better than others. One way may be more efficient, more precise, easier to modify, easier to use, or easier to understand. Any hacker can write code to make something work, but it takes the skill and understanding of a professional software engineer to produce code that is robust, easy to understand and maintain, and does its job in the most efficient and effective way possible. Consequently, software engineering is about designing and developing high-quality software.

Before we examine what is needed to produce quality software systems, let us look back to see how successful we have been. Are users happy with their existing software systems? Yes and no. Software has enabled us to perform tasks more quickly and effectively than ever before. Consider life before word processing, spreadsheets, electronic mail, or sophisticated telephony, for example. And software has supported life-sustaining or life-saving advances in medicine, agriculture, transportation, and most other industries. In addition, software has enabled us to do things that were never imagined in the past: microsurgery, multimedia education, robotics, and more.

However, software is not without its problems. Often systems function, but not exactly as expected. We all have heard stories of systems that just barely work. And we all have written faulty programs: code that contains mistakes, but is good enough for a passing grade or for demonstrating the feasibility of an approach. Clearly, such behavior is not acceptable when developing a system for delivery to a customer.

There is an enormous difference between an error in a class project and one in a large software system. In fact, software faults and the difficulty in producing fault-free software are frequently discussed in the literature and in the hallways. Some faults are merely annoying; others cost a great deal of time and money. Still others are life-threatening. Sidebar 1.1 explains the relationships among faults, errors, and failures. Let us look at a few examples of failures to see what is going wrong and why.

SIDEBAR 1.1 TERMINOLOGY FOR DESCRIBING BUGS

Often, we talk about "bugs" in software, meaning many things that depend on the context. A "bug" can be a mistake in interpreting a requirement, a syntax error in a piece of code, or the (as-yet-unknown) cause of a system crash. The Institute of Electrical and Electronics Engineers (IEEE) has suggested a standard terminology (in IEEE Standard 729) for describing "bugs" in our software products (IEEE 1983).

A **fault** occurs when a human makes a mistake, called an **error**, in performing some software activity. For example, a designer may misunderstand a requirement and create a design that does not match the actual intent of the requirements analyst and the user. This design fault is an encoding of the error, and it can lead to other faults, such as incorrect code and an incorrect description in a user manual. Thus, a single error can generate many faults, and a fault can reside in any development or maintenance product.

A **failure** is a departure from the system's required behavior. It can be discovered before or after system delivery, during testing, or during operation and maintenance. As we will see in Chapter 4, the requirements documents can contain faults. So a failure may indicate that the system is not performing as required, even though it may be performing as specified.

Thus, a fault is an inside view of the system, as seen by the eyes of the developers, whereas a failure is an outside view: a problem that the user sees. Not every fault corresponds to a failure; for example, if faulty code is never executed or a particular state is never entered, then the fault will never cause the code to fail. Figure 1.4 shows the genesis of a failure.

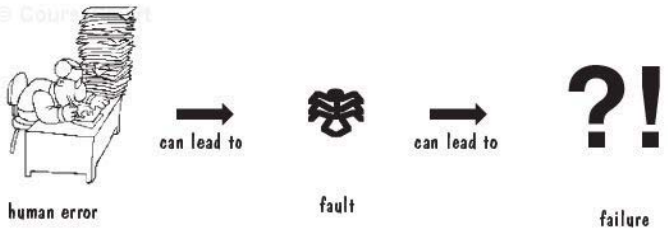


FIGURE 1.4 How human error causes a failure.

In the early 1980s, the United States Internal Revenue Service (IRS) hired Sperry Corporation to build an automated federal income tax form processing system. According to the *Washington Post,* the "system . . . proved inadequate to the workload, cost nearly twice what was expected and must be replaced soon" (Sawyer 1985). In 1985, an extra $90 million was needed to enhance the original $103 million worth of Sperry equipment. In addition, because the problem prevented the IRS from returning refunds to taxpayers by the deadline, the IRS was forced to pay $40.2 million in interest and $22.3 million in overtime wages for its employees who were trying to catch up. In 1996, the situation had not improved. The *Los Angeles Times* reported on March 29 that there was still no master plan for the modernization of IRS computers, only a 6000-page technical document. Congressman Jim Lightfoot called the project "a $4-billion fiasco that is floundering because of inadequate planning" (Vartabedian 1996).

Situations such as these still occur. In the United States, the Federal Bureau of Investigation's (FBI's) Trilogy project attempted to upgrade the FBI's computer systems. The results were devastating: "After more than four years of hard work and half a billion dollars spent, however, Trilogy has had little impact on the FBI's antiquated case-management system, which today remains a morass of mainframe green screens and vast stores of paper records" (Knorr 2005). Similarly, in the United Kingdom, the cost of overhauling the National Health Service's information systems was double the original estimate (Ballard 2006). We will see in Chapter 2 why project planning is essential to the production of quality software.

For many years, the public accepted the infusion of software in their daily lives with little question. But President Reagan's proposed Strategic Defense Initiative (SDI) heightened the public's awareness of the difficulty of producing a fault-free software system. Popular newspaper and magazine reports (such as Jacky 1985; Parnas 1985, Rensburger 1985) expressed skepticism in the computer science community. And now, years later, when the U.S. Congress is asked to allocate funds to build a similar system, many computer scientists and software engineers continue to believe there is no way to write and test the software to guarantee adequate reliability.

For example, many software engineers think that an antiballistic-missile system would require at least 10 million lines of code; some estimates range as high as one hundred million. By comparison, the software supporting the American space shuttle consists of 3 million lines of code, including computers on the ground controlling the launch and the flight; there were 100,000 lines of code in the shuttle itself in 1985 (Rensburger 1985). Thus, an antimissile software system would require the testing of an enormous amount of code. Moreover, the reliability constraints would be impossible to test. To see why, consider the notion of safety-critical software. Typically, we say that something that is **safety-critical** (i.e., something whose failure poses a threat to life or health) should have a reliability of at least 10^{-9}. As we shall see in Chapter 9, this terminology means that the system can fail no more often than once in 10^9 hours of operation. To observe this degree of reliability, we would have to run the system for at least 10^9 hours to verify that it does not fail. But 10^9 hours is over 114,000 years—far too long as a testing interval!

We will also see in Chapter 9 that helpful technology can become deadly when software is improperly designed or programmed. For example, the medical community was aghast when the Therac-25, a radiation therapy and X-ray machine, malfunctioned and killed several patients. The software designers had not anticipated the use of several

arrow keys in nonstandard ways; as a consequence, the software retained its high settings and issued a highly concentrated dose of radiation when low levels were intended (Leveson and Turner 1993).

Similar examples of unanticipated use and its dangerous consequences are easy to find. For example, recent efforts to use off-the-shelf components (as a cost savings measure instead of custom-crafting of software) result in designs that use components in ways not intended by the original developers. Many licensing agreements explicitly point to the risks of unanticipated use: "Because each end-user system is customized and differs from utilized testing platforms and because a user or application designer may use the software in combination with other products in a manner not evaluated or contemplated by [the vendor] or its suppliers, the user or application designer is ultimately responsible for verifying and validating the [software]" (*Lookout Direct* n.d.).

Unanticipated use of the system should be considered throughout software design activities. These uses can be handled in at least two ways: by stretching your imagination to think of how the system can be abused (as well as used properly), and by assuming that the system will be abused and designing the software to handle the abuses. We discuss these approaches in Chapter 8.

Although many vendors strive for zero-defect software, in fact most software products are not fault-free. Market forces encourage software developers to deliver products quickly, with little time to test thoroughly. Typically, the test team will be able to test only those functions most likely to be used, or those that are most likely to endanger or irritate users. For this reason, many users are wary of installing the first version of code, knowing that the bugs will not be worked out until the second version. Furthermore, the modifications needed to fix known faults are sometimes so difficult to make that it is easier to rewrite a whole system than to change existing code. We will investigate the issues involved in software maintenance in Chapter 11.

In spite of some spectacular successes and the overall acceptance of software as a fact of life, there is still much room for improvement in the quality of the software we produce. For example, lack of quality can be costly; the longer a fault goes undetected, the more expensive it is to correct. In particular, the cost of correcting an error made during the initial analysis of a project is estimated to be only one-tenth the cost of correcting a similar error after the system has been turned over to the customer. Unfortunately, we do not catch most of the errors early on. Half of the cost of correcting faults found during testing and maintenance comes from errors made much earlier in the life of a system. In Chapters 12 and 13, we will look at ways to evaluate the effectiveness of our development activities and improve the processes to catch mistakes as early as possible.

One of the simple but powerful techniques we will propose is the use of review and inspection. Many students are accustomed to developing and testing software on their own. But their testing may be less effective than they think. For example, Fagan studied the way faults were detected. He discovered that testing a program by running it with test data revealed only about a fifth of the faults located during systems development. However, peer review, the process whereby colleagues examine and comment on each other's designs and code, uncovered the remaining four out of five faults found (Fagan 1986). Thus, the quality of your software can be increased dramatically just by having your colleagues review your work. We will learn more in later chapters about how the review and inspection processes can be used after each major development

step to find and fix faults as early as possible. And we will see in Chapter 13 how to improve the inspection process itself.

1.3 WHAT IS GOOD SOFTWARE?

Just as manufacturers look for ways to ensure the quality of the products they produce, so too must software engineers find methods to ensure that their products are of acceptable quality and utility. Thus, good software engineering must always include a strategy for producing quality software. But before we can devise a strategy, we must understand what we mean by quality software. Sidebar 1.2 shows us how perspective influences what we mean by "quality." In this section, we examine what distinguishes good software from bad.

SIDEBAR 1.2 PERSPECTIVES ON QUALITY

Garvin (1984) discusses about how different people perceive quality. He describes quality from five different perspectives:

- the *transcendental view*, where quality is something we can recognize but not define
- the *user view*, where quality is fitness for purpose
- the *manufacturing view*, where quality is conformance to specification
- the *product view*, where quality is tied to inherent product characteristics
- the *value-based view*, where quality depends on the amount the customer is willing to pay for it

The transcendental view is much like Plato's description of the ideal or Aristotle's concept of form. In other words, just as every actual table is an approximation of an ideal table, we can think of software quality as an ideal toward which we strive; however, we may never be able to implement it completely.

The transcendental view is ethereal, in contrast to the more concrete view of the user. We take a user view when we measure product characteristics, such as defect density or reliability, in order to understand the overall product quality.

The manufacturing view looks at quality during production and after delivery. In particular, it examines whether the product was built right the first time, avoiding costly rework to fix delivered faults. Thus, the manufacturing view is a process view, advocating conformance to good process. However, there is little evidence that conformance to process actually results in products with fewer faults and failures; process may indeed lead to high-quality products, but it may possibly institutionalize the production of mediocre products. We examine some of these issues in Chapter 12.

The user and manufacturing views look at the product from the outside, but the product view peers inside and evaluates a product's inherent characteristics. This view is the one often advocated by software metrics experts; they assume that good internal quality indicators will lead to good external ones, such as reliability and maintainability. However, more research is

needed to verify these assumptions and to determine which aspects of quality affect the product's actual use. We may have to develop models that link the product view to the user view.

Customers or marketers often take a user view of quality. Researchers sometimes hold a product view, and the development team has a manufacturing view. If the differences in viewpoints are not made explicit, then confusion and misunderstanding can lead to bad decisions and poor products. The value-based view can link these disparate pictures of quality. By equating quality to what the customer is willing to pay, we can look at trade-offs between cost and quality, and we can manage conflicts when they arise. Similarly, purchasers compare product costs with potential benefits, thinking of quality as value for money.

Kitchenham and Pfleeger (1996) investigated the answer to this question in their introduction to a special issue of *IEEE Software* on quality. They note that the context helps to determine the answer. Faults tolerated in word processing software may not be acceptable in safety-critical or mission-critical systems. Thus, we must consider quality in at least three ways: the quality of the product, the quality of the process that results in the product, and the quality of the product in the context of the business environment in which the product will be used.

The Quality of the Product

We can ask people to name the characteristics of software that contribute to its overall quality, but we are likely to get different answers from each person we ask. This difference occurs because the importance of the characteristics depends on who is analyzing the software. Users judge software to be of high quality if it does what they want in a way that is easy to learn and easy to use. However, sometimes quality and functionality are intertwined; if something is hard to learn or use but its functionality is worth the trouble, then it is still considered to have high quality.

We try to measure software quality so that we can compare one product with another. To do so, we identify those aspects of the system that contribute to its overall quality. Thus, when measuring software quality, users assess such external characteristics as the number of failures and type of failures. For example, they may define failures as minor, major, and catastrophic, and hope that any failures that occur are only minor ones.

The software must also be judged by those who are designing and writing the code and by those who must maintain the programs after they are written. These practitioners tend to look at internal characteristics of the products, sometimes even before the product is delivered to the user. In particular, practitioners often look at numbers and types of faults for evidence of a product's quality (or lack of it). For example, developers track the number of faults found in requirements, design, and code inspections and use them as indicators of the likely quality of the final product.

For this reason, we often build models to relate the user's external view to the developer's internal view of the software. Figure 1.5 is an example of an early quality model built by McCall and his colleagues to show how external quality factors (on the left-hand side) relate to product quality criteria (on the right-hand side). McCall associated each right-hand criterion with a measurement to indicate the degree to which an

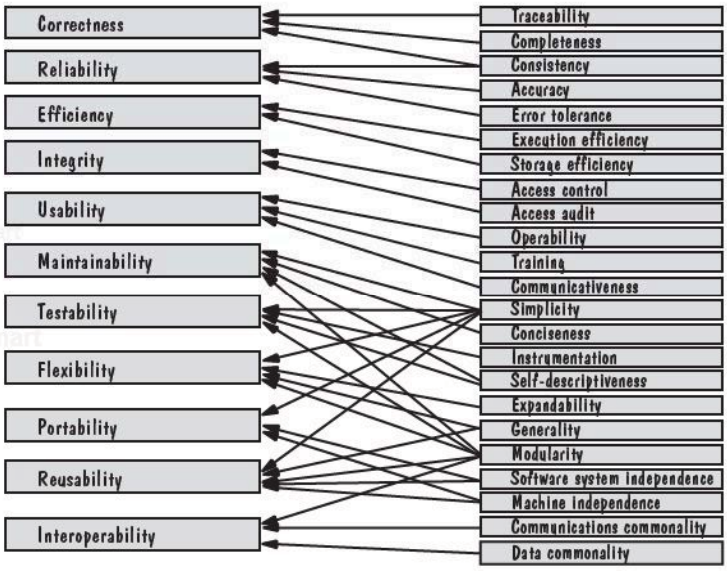


FIGURE 1.5 McCall's quality model.

element of quality was addressed (McCall, Richards, and Walters 1977). We will examine several product quality models in Chapter 12.

The Quality of the Process

There are many activities that affect the ultimate product quality; if any of the activities go awry, the product quality may suffer. For this reason, many software engineers feel that the quality of the development and maintenance process is as important as product quality. One of the advantages of modeling the process is that we can examine it and look for ways to improve it. For example, we can ask questions such as:

- Where and when are we likely to find a particular kind of fault?
- How can we find faults earlier in the development process?
- How can we build in fault tolerance so that we minimize the likelihood that a fault will become a failure?
- How can we design secure, high-quality systems?
- Are there alternative activities that can make our process more effective or efficient at ensuring quality?

These questions can be applied to the whole development process or to a subprocess, such as configuration management, reuse, or testing; we will investigate these processes in later chapters.

In the 1990s, there was a well-publicized focus on process modeling and process improvement in software engineering. Inspired by the work of Deming and Juran, and implemented by companies such as IBM, process guidelines such as the Capability

Maturity Model (CMM), ISO 9000, and Software Process Improvement and Capability dEtermination (SPICE) suggested that by improving the software development process, we can improve the quality of the resulting products. In Chapter 2, we will see how to identify relevant process activities and model their effects on intermediate and final products. Chapters 12 and 13 will examine process models and improvement frameworks in depth.

Quality in the Context of the Business Environment

When the focus of quality assessment is on products and processes, we usually measure quality with mathematical expressions involving faults, failures, and timing. Rarely is the scope broadened to include a business perspective, where quality is viewed in terms of the products and services being provided by the business in which the software is embedded. That is, we look at the technical value of our products, rather than more broadly at their business value, and we make decisions based only on the resulting products' technical quality. In other words, we assume that improving technical quality will automatically translate into business value.

Several researchers have taken a close look at the relationships between business value and technical value. For example, Simmons interviewed many Australian businesses to determine how they make their information technology-related business decisions. She proposes a framework for understanding what companies mean by "business value" (Simmons 1996). In a report by Favaro and Pfleeger (1997), Steve Andriole, chief information officer for Cigna Corporation, a large U.S. insurance company, described how his company distinguishes technical value from business value:

> We measure the quality [of our software] by the obvious metrics: up versus down time, maintenance costs, costs connected with modifications, and the like. In other words, we manage development based on operational performance within cost parameters. HOW the vendor provides cost-effective performance is less of a concern than the results of the effort. . . . The issue of business versus technical value is near and dear to our heart . . . and one [on] which we focus a great deal of attention. I guess I am surprised to learn that companies would contract with companies for their technical value, at the relative expense of business value. If anything, we err on the other side! If there is not clear (expected) business value (expressed quantitatively: number of claims processed, etc.) then we can't launch a systems project. We take very seriously the "purposeful" requirement phase of the project, when we ask: "why do we want this system?" and "why do we care?"

There have been several attempts to relate technical value and business value in a quantitative and meaningful way. For example, Humphrey, Snyder, and Willis (1991) note that by improving its development process according to the CMM "maturity" scale (to be discussed in Chapter 12), Hughes Aircraft improved its productivity by 4 to 1 and saved millions of dollars. Similarly, Dion (1993) reported that Raytheon's twofold increase in productivity was accompanied by a $7.70 return on every dollar invested in process improvement. And personnel at Tinker Air Force Base in Oklahoma noted a productivity improvement of 6.35 to 1 (Lipke and Butler 1992).

However, Brodman and Johnson (1995) took a closer look at the business value of process improvement. They surveyed 33 companies that performed some kind of process improvement activities, and examined several key issues. Among other things, Brodman

and Johnson asked companies how they defined return on investment (ROI), a concept that is clearly defined in the business community. They note that the textbook definition of **return on investment**, derived from the financial community, describes the investment in terms of what is given up for other purposes. That is, the "investment must not only return the original capital but enough more to at least equal what the funds would have earned elsewhere, plus an allowance for risk" (Putnam and Myers 1992). Usually, the business community uses one of three models to assess ROI: a payback model, an accounting rate-of-return model, and a discounted cash flow model.

However, Brodman and Johnson (1995) found that the U.S. government and U.S. industry interpret ROI in very different ways, each different from the other, and both different from the standard business school approaches. The government views ROI in terms of dollars, looking at reducing operating costs, predicting dollar savings, and calculating the cost of employing new technologies. Government investments are also expressed in dollars, such as the cost of introducing new technologies or process improvement initiatives.

On the other hand, industry viewed investment in terms of effort, rather than cost or dollars. That is, companies were interested in saving time or using fewer people, and their definition of return on investment reflected this focus on decreasing effort. Among the companies surveyed, return on investment included such items as

- training
- schedule
- risk
- quality
- productivity
- process
- customer
- costs
- business

The cost issues included in the definition involve meeting cost predictions, improving cost performance, and staying within budget, rather than reducing operating costs or streamlining the project or organization. Figure 1.6 summarizes the frequency with which many organizations included an investment item in their definition of ROI. For example, about 5 percent of those interviewed included a quality group's effort in the ROI effort calculation, and approximately 35 percent included software costs when considering number of dollars invested.

The difference in views is disturbing, because it means that calculations of ROI cannot be compared across organizations. But there are good reasons for these differing views. Dollar savings from reduced schedule, higher quality, and increased productivity are returned to the government rather than the contractor. On the other hand, contractors are usually looking for a competitive edge and increased work capacity as well as greater profit; thus, the contractor's ROI is more effort- than cost-based. In particular, more accurate cost and schedule estimation can mean customer satisfaction and repeat business. And decreased time to market as well as improved product quality are perceived as offering business value, too.

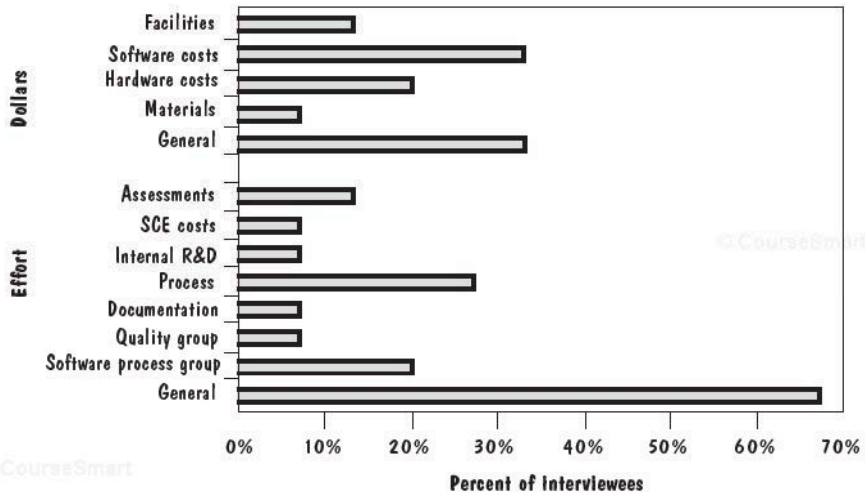


FIGURE 1.6 Terms included in industry definition of return on investment.

Even though the different ROI calculations can be justified for each organization, it is worrying that software technology return on investment is not the same as financial ROI. At some point, program success must be reported to higher levels of management, many of which are related not to software but to the main company business, such as telecommunications or banking. Much confusion will result from the use of the same terminology to mean vastly different things. Thus, our success criteria must make sense not only for software projects and processes, but also for the more general business practices they support. We will examine this issue in more detail in Chapter 12 and look at using several common measures of business value to choose among technology options.

1.4 WHO DOES SOFTWARE ENGINEERING?

A key component of software development is communication between customer and developer; if that fails, so too will the system. We must understand what the customer wants and needs before we can build a system to help solve the customer's problem. To do this, let us turn our attention to the people involved in software development.

The number of people working on software development depends on the project's size and degree of difficulty. However, no matter how many people are involved, the roles played throughout the life of the project can be distinguished. Thus, for a large project, one person or a group may be assigned to each of the roles identified; on a small project, one person or group may take on several roles at once.

Usually, the participants in a project fall into one of three categories: customer, user, or developer. The **customer** is the company, organization, or person who is paying for the software system to be developed. The **developer** is the company, organization, or person who is building the software system for the customer. This category includes any

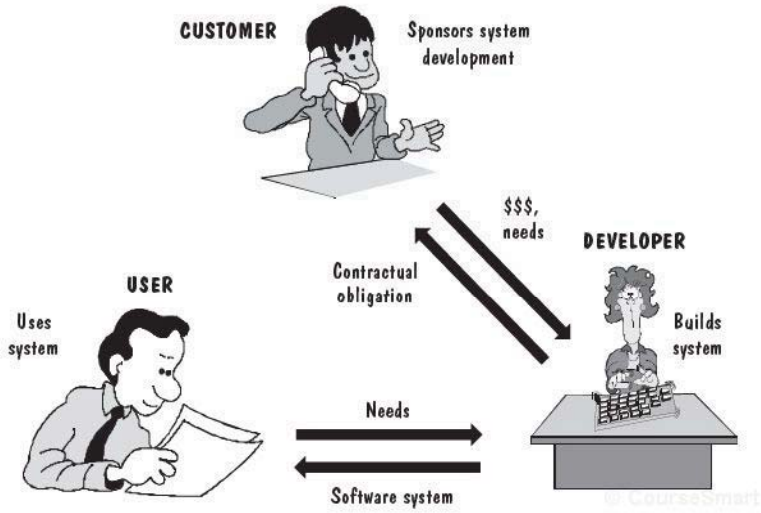


FIGURE 1.7 Participants in software development.

managers needed to coordinate and guide the programmers and testers. The **user** is the person or people who will actually use the system: the ones who sit at the terminal or submit the data or read the output. Although for some projects the customer, user, and developer are the same person or group, often these are different sets of people. Figure 1.7 shows the basic relationships among the three types of participants.

The customer, being in control of the funds, usually negotiates the contract and signs the acceptance papers. However, sometimes the customer is not a user. For example, suppose Wittenberg Water Works signs a contract with Gentle Systems, Inc., to build a computerized accounting system for the company. The president of Wittenberg may describe to the representatives of Gentle Systems exactly what is needed, and she will sign the contract. However, the president will not use the accounting system directly; the users will be the bookkeepers and accounting clerks. Thus, it is important that the developers understand exactly what both the customer and users want and need.

On the other hand, suppose Wittenberg Water Works is so large that it has its own computer systems development division. The division may decide that it needs an automated tool to keep track of its own project costs and schedules. By building the tool itself, the division is at the same time the user, customer, and developer.

In recent years, the simple distinctions among customer, user, and developer have become more complex. Customers and users have been involved in the development process in a variety of ways. The customer may decide to purchase Commercial Off-The-Shelf (**COTS**) software to be incorporated in the final product that the developer will supply and support. When this happens, the customer is involved in system architecture decisions, and there are many more constraints on development. Similarly, the developer may choose to use additional developers, called **subcontractors**, who build a subsystem and deliver it to the developers to be included in the final product. The subcontractors may work side by side with the primary developers, or they may work at a different site,

coordinating their work with the primary developers and delivering the subsystem late in the development process. The subsystem may be a **turnkey system**, where the code is incorporated whole (without additional code for integration), or it may require a separate integration process for building the links from the major system to the subsystem(s).

Thus, the notion of "system" is important in software engineering, not only for understanding the problem analysis and solution synthesis, but also for organizing the development process and for assigning appropriate roles to the participants. In the next section, we look at the role of a systems approach in good software engineering practice.

1.5 A SYSTEMS APPROACH

The projects we develop do not exist in a vacuum. Often, the hardware and software we put together must interact with users, with other software tasks, with other pieces of hardware, with existing databases (i.e., with carefully defined sets of data and data relationships), or even with other computer systems. Therefore, it is important to provide a context for any project by knowing the **boundaries** of the project: what is included in the project and what is not. For example, suppose you are asked by your supervisor to write a program to print paychecks for the people in your office. You must know whether your program simply reads hours worked from another system and prints the results or whether you must also calculate the pay information. Similarly, you must know whether the program is to calculate taxes, pensions, and benefits or whether a report of these items is to be provided with each paycheck. What you are really asking is: Where does the project begin and end? The same question applies to any system. A system is a collection of objects and activities, plus a description of the relationships that tie the objects and activities together. Typically, our system definition includes, for each activity, a list of inputs required, actions taken, and outputs produced. Thus, to begin, we must know whether any object or activity is included in the system or not.

The Elements of a System

We describe a system by naming its parts and then identifying how the component parts are related to one another. This identification is the first step in analyzing the problem presented to us.

Activities and Objects. First, we distinguish between activities and objects. An **activity** is something that happens in a system. Usually described as an event initiated by a trigger, the activity transforms one thing to another by changing a characteristic. This transformation can mean that a data element is moved from one location to another, is changed from one value to another, or is combined with other data to supply input for yet another activity. For example, an item of data can be moved from one file to another. In this case, the characteristic changed is the location. Or the value of the data item can be incremented. Finally, the address of the data item can be included in a list of parameters with the addresses of several other data items so that another routine can be called to handle all the data at once.

The elements involved in the activities are called **objects** or **entities**. Usually, these objects are related to each other in some way. For instance, the objects can be

arranged in a table or matrix. Often, objects are grouped as records, where each record is arranged in a prescribed format. An employee history record, for example, may contain objects (called fields) for each employee, such as the following:

First name	Postal code
Middle name	Salary per hour
Last name	Benefits per hour
Street address	Vacation hours accrued
City	Sick leave accrued
State	

Not only is each field in the record defined, but the size and relationship of each field to the others are named. Thus, the record description states the data type of each field, the starting location in the record, and the length of the field. In turn, since there is a record for each employee, the records are combined into a file, and file characteristics (such as maximum number of records) may be specified.

Sometimes, the objects are defined slightly differently. Instead of considering each item as a field in a larger record, the object is viewed as being independent. The object description contains a listing of the characteristics of each object, as well as a list of all the actions that can take place using the object or affecting the object. For example, consider the object "polygon." An object description may say that this object has characteristics such as number of sides and length of each side. The actions may include calculation of the area or of the perimeter. There may even be a characteristic called "polygon type," so that each instantiation of "polygon" is identified when it is a "rhombus" or "rectangle," for instance. A type may itself have an object description; "rectangle" may be composed of types "square" and "not square," for example. We will explore these concepts in Chapter 4 when we investigate requirements analysis, and in depth in Chapter 6 when we discuss object-oriented development.

Relationships and the System Boundary. Once entities and activities are defined, we match the entities with their activities. The relationships among entities and activities are clearly and carefully defined. An entity definition includes a description of where the entity originates. Some items reside in files that already exist; others are created during some activity. The entity's destination is important, too. Some items are used by only one activity, but others are destined to be input to other systems. That is, some items from one system are used by activities outside the scope of the system being examined. Thus, we can think of the system at which we are looking as having a border or boundary. Some items cross the boundary to enter our system, and others are products of our system and travel out for another system's use.

Using these concepts, we can define a **system** as a collection of things: a set of entities, a set of activities, a description of the relationships among entities and activities, and a definition of the boundary of the system. This definition of a system applies not only to computer systems but to anything in which objects interact in some way with other objects.

Examples of Systems. To see how system definition works, consider the parts of you that allow you to take in oxygen and excrete carbon dioxide and water: your respiratory system. You can define its boundary easily: If you name a particular organ of

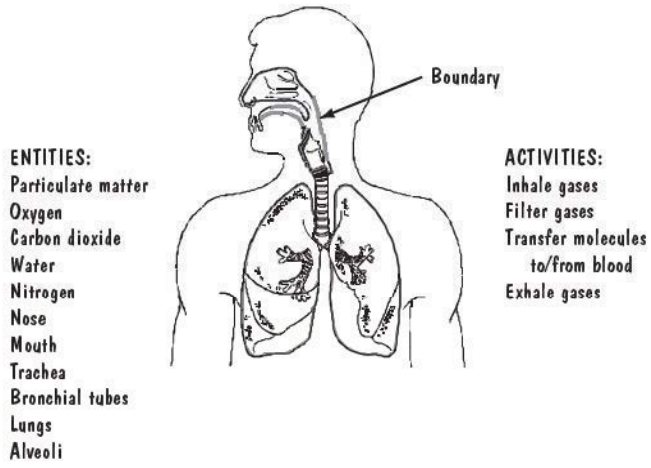


FIGURE 1.8 Respiratory system.

your body, you can say whether or not it is part of your respiratory system. Molecules of oxygen and carbon dioxide are entities or objects moving through the system in ways that are clearly defined. We can also describe the activities in the system in terms of the interactions of the entities. If necessary, we can illustrate the system by showing what enters and leaves it; we can also supply tables to describe all entities and the activities in which they are involved. Figure 1.8 illustrates the respiratory system. Note that each activity involves the entities and can be defined by describing which entities act as input, how they are processed, and what is produced (output).

We must describe our computer systems clearly, too. We work with prospective users to define the boundary of the system: Where does our work start and stop? In addition, we need to know what is on the boundary of the system and thus determine the origins of the input and destinations of the output. For example, in a system that prints paychecks, pay information may come from the company's payroll system. The system output may be a set of paychecks sent to the mail room to be delivered to the appropriate recipients. In the system shown in Figure 1.9, we can see the boundary and can understand the entities, the activities, and their relationships.

Interrelated Systems

The concept of boundary is important, because very few systems are independent of other systems. For example, the respiratory system must interact with the digestive system, the circulatory system, the nervous system, and others. The respiratory system could not function without the nervous system; neither could the circulatory system function without the respiratory system. The interdependencies may be complex. (Indeed, many of our environmental problems arise and are intensified because we do not appreciate the complexity of our ecosystem.) However, once the boundary of a system is described, it is easier for us to see what is within and without and what crosses the boundary.

In turn, it is possible for one system to exist inside another system. When we describe a computer system, we often concentrate on a small piece of what is really a

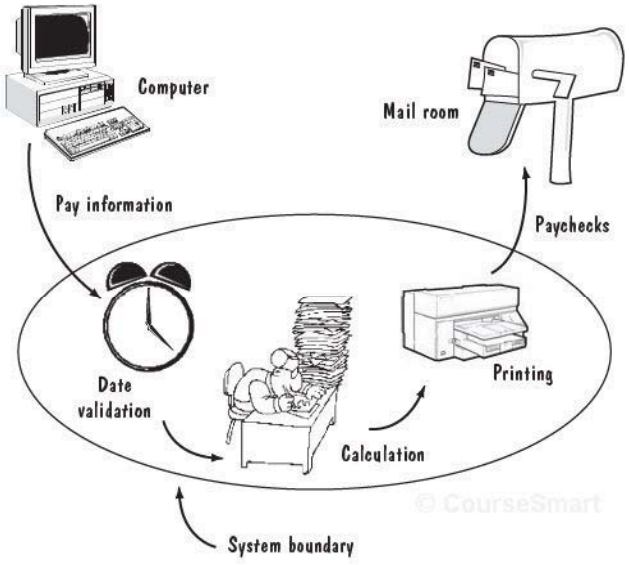


FIGURE 1.9 System definition of paycheck production.

much larger system. Such a focus allows us to define and build a much less complex system than the enveloping one. If we are careful in documenting the interactions among and between systems affecting ours, we lose nothing by concentrating on this smaller piece of a larger system.

Let us look at an example of how this can be done. Suppose we are developing a water-monitoring system where data are gathered at many points throughout a river valley. At the collection sites, several calculations are done, and the results are communicated to a central location for comprehensive reporting. Such a system may be implemented with a computer at the central site communicating with several dozen smaller computers at the remote locations. Many system activities must be considered, including the way the water data are gathered, the calculations performed at the remote locations, the communication of information to the central site, the storage of the communicated data in a database or shared data file, and the creation of reports from the data. We can view this system as a collection of systems, each with a special purpose. In particular, we can consider only the communications aspect of the larger system and develop a communications system to transmit data from a set of remote sites to a central one. If we carefully define the boundary between the communications and the larger system, the design and development of the communications system can be done independently of the larger system.

The complexity of the entire water-monitoring system is much greater than the complexity of the communications system, so our treatment of separate, smaller pieces makes our job much simpler. If the boundary definitions are detailed and correct, building the larger system from the smaller ones is relatively easy. We can describe the building process by considering the larger system in layers, as illustrated in Figure 1.10 for our water-monitoring example. A layer is a system by itself, but each layer and those it contains also form a system. The circles of the figure represent the boundaries of the respective systems, and the entire set of circles incorporates the entire water-monitoring system.

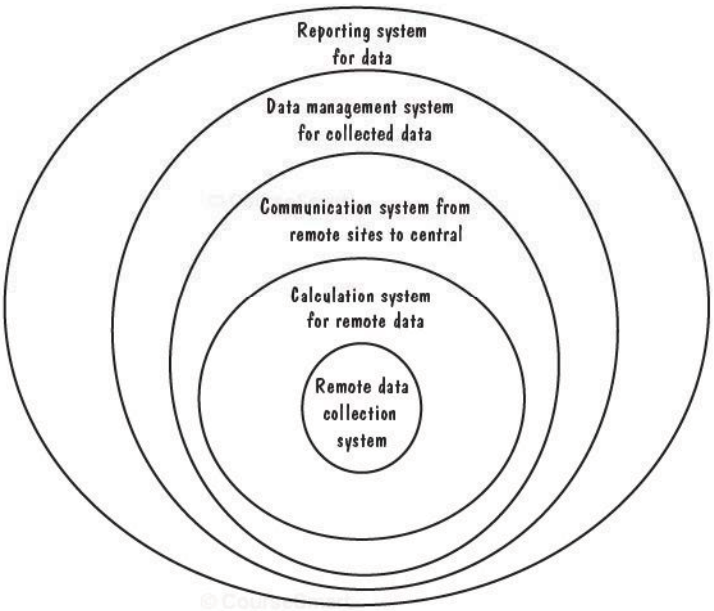


FIGURE 1.10 Layers of a water-monitoring system.

Recognizing that one system contains another is important, because it reflects the fact that an object or activity in one system is part of every system represented by the outer layers. Since more complexity is introduced with each layer, understanding any one object or activity becomes more difficult with each more encompassing system. Thus, we maximize simplicity and our consequent understanding of the system by focusing on the smallest system possible at first.

We use this idea when building a system to replace an older version, either manual or automated. We want to understand as much as possible about how both the old and new systems work. Often, the greater the difference between the two systems, the more difficult the design and development. This difficulty occurs not only because people tend to resist change, but also because the difference makes learning difficult. In building or synthesizing our grand system, it helps dramatically to construct a new system as an incremental series of intermediate systems. Rather than going from system A to system B, we may be able to go from A to A′ to A″ to B. For example, suppose A is a manual system consisting of three major functions, and B is to be an automated version of A. We can define system A′ to be a new system with function 1 automated but functions 2 and 3 still manual. Then A″ has automated functions 1 and 2, but 3 is still manual. Finally, B has all three automated functions. By dividing the "distance" from A to B in thirds, we have a series of small problems that may be easier to handle than the whole.

In our example, the two systems are very similar; the functions are the same, but the style in which they are implemented differs. However, the target system is often vastly different from the existing one. In particular, it is usually desirable that the target be free of constraints imposed by existing hardware or software. An **incremental development**

approach may incorporate a series of stages, each of which frees the previous system from another such constraint. For example, stage 1 may add a new piece of hardware, stage 2 may replace the software performing a particular set of functions, and so on. The system is slowly drawn away from old software and hardware until it reflects the new system design.

Thus, system development can first incorporate a set of changes to an actual system and then add a series of changes to generate a complete design scheme, rather than trying to jump from present to future in one move. With such an approach, we must view the system in two different ways simultaneously: statically and dynamically. The static view tells us how the system is working today, whereas the dynamic view shows us how the system is changing into what it will eventually become. One view is not complete without the other.

1.6 AN ENGINEERING APPROACH

Once we understand the system's nature, we are ready to begin its construction. At this point, the "engineering" part of software engineering becomes relevant and complements what we have done so far. Recall that we began this chapter by acknowledging that writing software is an art as well as a science. The art of producing systems involves the craft of software production. As artists, we develop techniques and tools that have proven helpful in producing useful, high-quality products. For instance, we may use an optimizing compiler as a tool to generate programs that run fast on the machines we are using. Or we can include special sort or search routines as techniques for saving time or space in our system. These software-based techniques are used just as techniques and tools are used in crafting a fine piece of furniture or in building a house. Indeed, a popular collection of programming tools is called the Programmer's Workbench, because programmers rely on them as a carpenter relies on a workbench.

Because building a system is similar to building a house, we can look to house building for other examples of why the "artistic" approach to software development is important.

Building a House

Suppose Chuck and Betsy Howell hire someone to build a house for them. Because of its size and complexity, a house usually requires more than one person on the construction team; consequently, the Howells hire McMullen Construction Company. The first event involved in the house building is a conference between the Howells and McMullen so the Howells can explain what they want. This conference explores not only what the Howells want the house to look like, but also what features are to be included. Then McMullen draws up floor plans and an architect's rendering of the house. After the Howells discuss the details with McMullen, changes are made. Once the Howells give their approval to McMullen, construction begins.

During the construction process, the Howells are likely to inspect the construction site, thinking of changes they would like. Several such changes may occur during construction, but eventually the house is completed. During construction and before the Howells move in, several components of the house are tested. For example, electricians

test the wiring circuits, plumbers make sure that pipes do not leak, and carpenters adjust for variation in wood so that the floors are smooth and level. Finally, the Howells move in. If there is something that is not constructed properly, McMullen may be called in to fix it, but eventually the Howells become fully responsible for the house.

Let us look more closely at what is involved in this process. First, since many people are working on the house at the same time, documentation is essential. Not only are floor plans and the architect's drawings necessary, but details must be written down so that specialists such as plumbers and electricians can fit their products together as the house becomes a whole.

Second, it is unreasonable to expect the Howells to describe their house at the beginning of the process and walk away until the house is completed. Instead, the Howells may modify the house design several times during construction. These modifications may result from a number of situations:

- Materials that were specified initially are no longer available. For example, certain kinds of roof tiles may no longer be manufactured.
- The Howells may have new ideas as they see the house take shape. For example, the Howells might realize that they can add a skylight to the kitchen for little additional cost.
- Availability or financial constraints may require the Howells to change requirements in order to meet their schedule or budget. For example, the special windows that the Howells wanted to order will not be ready in time to complete the house by winter, so stock windows may be substituted.
- Items or designs initially thought possible might turn out to be infeasible. For example, soil percolation tests may reveal that the land surrounding the house cannot support the number of bathrooms that the Howells had originally requested.

McMullen may also recommend some changes after construction has begun, perhaps because of a better idea or because a key member of the construction crew is unavailable. And both McMullen and the Howells may change their minds about a feature of the house even after the feature is completed.

Third, McMullen must provide blueprints, wiring and plumbing diagrams, instruction manuals for the appliances, and any other documentation that would enable the Howells to make modifications or repairs after they move in.

We can summarize this construction process in the following way:

- determining and analyzing the requirements
- producing and documenting the overall design of the house
- producing detailed specifications of the house
- identifying and designing the components
- building each component of the house
- testing each component of the house
- integrating the components and making final modifications after the residents have moved in
- continuing maintenance by the residents of the house

We have seen how the participants must remain flexible and allow changes in the original specifications at various points during construction.

It is important to remember that the house is built within the context of the social, economic, and governmental structure in which it is to reside. Just as the water-monitoring system in Figure 1.10 depicted the dependencies of subsystems, we must think of the house as a subsystem in a larger scheme. For example, construction of a house is done in the context of the city or county building codes and regulations. The McMullen employees are licensed by the city or county, and they are expected to perform according to building standards. The construction site is visited by building inspectors, who make sure that the standards are being followed. And the building inspectors set standards for quality, with the inspections serving as quality assurance checkpoints for the building project. There may also be social or customary constraints that suggest common or acceptable behavior; for example, it is not customary to have the front door open directly to the kitchen or bedroom.

At the same time, we must recognize that we cannot prescribe the activities of building a house exactly; we must leave room for decisions based on experience, to deal with unexpected or nonstandard situations. For example, many houses are fashioned from pre-existing components; doors are supplied already in the frame, bathrooms use pre-made shower stalls, and so on. But the standard house-building process may have to be altered to accommodate an unusual feature or request. Suppose that the framing is done, the drywall is up, the subfloor is laid, and the next step is putting down tile on the bathroom floor. The builders find, much to their dismay, that the walls and floor are not exactly square. This problem may not be the result of a poor process; houses are built from parts that have some natural or manufacturing variation, so problems of inexactitude can occur. The floor tile, being composed of small squares, will highlight the inexactitude if laid the standard way. It is here that art and expertise come to play. The builder is likely to remove the tiles from their backing, and lay them one at a time, making small adjustments with each one so that the overall variation is imperceptible to all but the most discerning eyes.

Thus, house building is a complex task with many opportunities for change in processes, products, or resources along the way, tempered by a healthy dose of art and expertise. The house-building process can be standardized, but there is always need for expert judgment and creativity.

Building a System

Software projects progress in a way similar to the house-building process. The Howells were the customers and users, and McMullen was the developer in our example. Had the Howells asked McMullen to build the house for Mr. Howell's parents to live in, the users, customers, and developer would have been distinct. In the same way, software development involves users, customers, and developers. If we are asked to develop a software system for a customer, the first step is meeting with the customer to determine the requirements. These requirements describe the system, as we saw before. Without knowing the boundary, the entities, and the activities, it is impossible to describe the software and how it will interact with its environment.

Once requirements are defined, we create a system design to meet the specified requirements. As we will see in Chapter 5, the system design shows the customer what

the system will look like from the customer's perspective. Thus, just as the Howells looked at floor plans and architect's drawings, we present the customer with pictures of the video display screens that will be used, the reports that will be generated, and any other descriptions that will explain how users will interact with the completed system. If the system has manual backup or override procedures, those are described as well. At first, the Howells were interested only in the appearance and functionality of their house; it was not until later that they had to decide on such items as copper or plastic pipes. Likewise, the system design (also called architectural) phase of a software project describes only appearance and functionality.

The design is then reviewed by the customer. When approved, the overall system design is used to generate the designs of the individual programs involved. Note that it is not until this step that programs are mentioned. Until functionality and appearance are determined, it often makes no sense to consider coding. In our house example, we would now be ready to discuss types of pipe or quality of electrical wiring. We can decide on plastic or copper pipes because now we know where water needs to flow in the structure. Likewise, when the system design is approved by all, we are ready to discuss programs. The basis for our discussion is a well-defined description of the software project as a system; the system design includes a complete description of the functions and interactions involved.

When the programs have been written, they are tested as individual pieces of code before they can be linked together. This first phase of testing is called module or unit testing. Once we are convinced that the pieces work as desired, we put them together and make sure that they work properly when joined with others. This second testing phase is often referred to as integration testing, as we build our system by adding one piece to the next until the entire system is operational. The final testing phase, called system testing, involves a test of the whole system to make sure that the functions and interactions specified initially have been implemented properly. In this phase, the system is compared with the specified requirements; the developer, customer, and users check that the system serves its intended purpose.

At last, the final product is delivered. As it is used, discrepancies and problems are uncovered. If ours is a turnkey system, the customer assumes responsibility for the system after delivery. Many systems are not turnkey systems, though, and the developer or other organization provides maintenance if anything goes wrong or if needs and requirements change.

Thus, development of software includes the following activities:

- requirements analysis and definition
- system design
- program design
- writing the programs (program implementation)
- unit testing
- integration testing
- system testing
- system delivery
- maintenance

In an ideal situation, the activities are performed one at a time; when you reach the end of the list, you have a completed software project. However, in reality, many of the steps are repeated. For example, in reviewing the system design, you and the customer may discover that some requirements have yet to be documented. You may work with the customer to add requirements and possibly redesign the system. Similarly, when writing and testing code, you may find that a device does not function as described by its documentation. You may have to redesign the code, reconsider the system design, or even return to a discussion with the customer about how to meet the requirements. For this reason, we define a **software development process** as any description of software development that contains some of the nine activities listed before, organized so that together they produce tested code. In Chapter 2, we will explore several of the different development processes that are used in building software. Subsequent chapters will examine each of the subprocesses and their activities, from requirements analysis through maintenance. But before we do, let us look at who develops software and how the challenge of software development has changed over the years.

1.7 MEMBERS OF THE DEVELOPMENT TEAM

Earlier in this chapter, we saw that customers, users, and developers play major roles in the definition and creation of the new product. The developers are software engineers, but each engineer may specialize in a particular aspect of development. Let us look in more detail at the role of the members of the development team.

The first step in any development process is finding out what the customer wants and documenting the requirements. As we have seen, analysis is the process of breaking things into their component parts so that we can understand them better. Thus, the development team includes one or more *requirements analysts* to work with the customer, breaking down what the customer wants into discrete requirements.

Once the requirements are known and documented, analysts work with *designers* to generate a system-level description of what the system is to do. In turn, the designers work with programmers to describe the system in such a way that *programmers* can write lines of code that implement what the requirements specify.

After the code is generated, it must be tested. Often, the first testing is done by the programmers themselves; sometimes, additional *testers* are also used to help catch faults that the programmers overlook. When units of code are integrated into functioning groups, a team of testers works with the implementation team to verify that as the system is built up by combining pieces, it works properly and according to specification.

When the development team is comfortable with the functionality and quality of the system, attention turns to the *customer*. The test team and customer work together to verify that the complete system is what the customer wants; they do this by comparing how the system works with the initial set of requirements. Then, *trainers* show users how to use the system.

For many software systems, acceptance by the customer does not mean the end of the developer's job. If faults are discovered after the system has been accepted, a *maintenance team* fixes them. Moreover, the customer's requirements may change as time passes, and corresponding changes to the system must be made. Thus, maintenance

can involve analysts who determine what requirements are added or changed, designers to determine where in the system design the change should be made, programmers to implement the changes, testers to make sure that the changed system still runs properly, and trainers to explain to users how the change affects the use of the system. Figure 1.11 illustrates how the roles of the development team correspond to the steps of development.

Students often work by themselves or in small groups as a development team for class projects. The documentation requested by the instructor is minimal; students are usually not required to write a user manual or training documents. Moreover, the assignment is relatively stable; the requirements do not change over the life of the project. Finally, student-built systems are likely to be discarded at the end of the course; their purpose is to demonstrate ability but not necessarily to solve a problem for a real customer. Thus, program size, system complexity, need for documentation, and need for maintainability are relatively small for class projects.

However, for a real customer, the system size and complexity may be large and the need for documentation and maintainability great. For a project involving many thousands of lines of code and much interaction among members of the development team, control of the various aspects of the project may be difficult. To support everyone on the development team, several people may become involved with the system at the beginning of development and remain involved throughout.

Librarians prepare and store documents that are used during the life of the system, including requirements specifications, design descriptions, program documentation, training manuals, test data, schedules, and more. Working with the librarians are

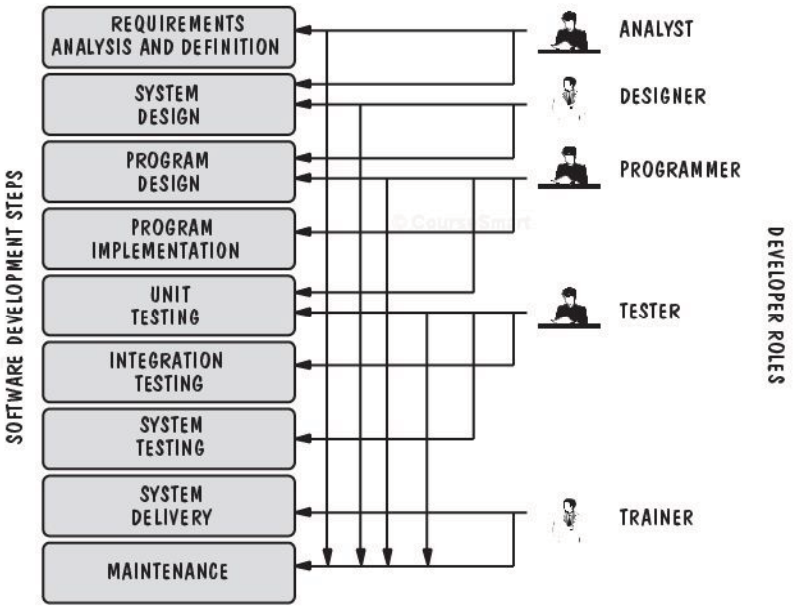


FIGURE 1.11 The roles of the development team.

the members of a *configuration management team*. Configuration management involves maintaining a correspondence among the requirements, the design, the implementation, and the tests. This cross-reference tells developers what program to alter if a change in requirements is needed, or what parts of a program will be affected if an alteration of some kind is proposed. Configuration management staff also coordinate the different versions of a system that may be built and supported. For example, a software system may be hosted on different platforms or may be delivered in a series of releases. Configuration management ensures that the functionality is consistent from one platform to another, and that it doesn't degrade with a new release.

The development roles can be assumed by one person or several. For small projects, two or three people may share all roles. However, for larger projects, the development team is often separated into distinct groups based on their function in development. Sometimes, those who maintain the system are different from those who design or write the system initially. For a very large development project, the customer can even hire one company to do the initial development and another to do the maintenance. As we discuss the development and maintenance activities in later chapters, we will look at what skills are needed by each type of development role.

1.8 HOW HAS SOFTWARE ENGINEERING CHANGED?

We have compared the building of software to the building of a house. Each year, hundreds of houses are built across the country, and satisfied customers move in. Each year, hundreds of software products are built by developers, but customers are too often unhappy with the result. Why is there a difference? If it is so easy to enumerate the steps in the development of a system, why are we as software engineers having such a difficult time producing quality software?

Think back to our house-building example. During the building process, the Howells continually reviewed the plans. They also had many opportunities to change their minds about what they wanted. In the same way, software development allows the customer to review the plans at every step and to make changes in the design. After all, if the developer produces a marvelous product that does not meet the customer's needs, the resultant system will have wasted everyone's time and effort.

For this reason, it is essential that our software engineering tools and techniques be used with an eye toward flexibility. In the past, we as developers assumed that our customers knew from the start what they wanted. That stability is not usually the case. As the various stages of a project unfold, constraints arise that were not anticipated at the beginning. For instance, after having chosen hardware and software to use for a project, we may find that a change in the customer requirements makes it difficult to use a particular database management system to produce menus exactly as promised to the customer. Or we may find that another system with which ours is to interface has changed its procedure or the format of the expected data. We may even find that hardware or software does not work quite as the vendor's documentation had promised. Thus, we must remember that each project is unique and that tools and techniques must be chosen that reflect the constraints placed on the individual project.

We must also acknowledge that most systems do not stand by themselves. They interface with other systems, either to receive or to provide information. Developing

such systems is complex simply because they require a great deal of coordination with the systems with which they communicate. This complexity is especially true of systems that are being developed concurrently. In the past, developers had difficulty ensuring the accuracy and completeness of the documentation of interfaces among systems. In subsequent chapters, we will address the issue of controlling the interface problem.

The Nature of the Change

These problems are among many that affect the success of our software development projects. Whatever approach we take, we must look both backward and forward. That is, we must look back at previous development projects to see what we have learned, not only about ensuring software quality, but also about the effectiveness of our techniques and tools. And we must look ahead to the way software development and the use of software products are likely to change our practices in the future. Wasserman (1995) points out that the changes since the 1970s have been dramatic. For example, early applications were intended to run on a single processor, usually a mainframe. The input was linear, usually a deck of cards or an input tape, and the output was alphanumeric. The system was designed in one of two basic ways: as a **transformation**, where input was converted to output, or as a **transaction**, where input determined which function would be performed. Today's software-based systems are far different and more complex. Typically, they run on multiple systems, sometimes configured in a client-server architecture with distributed functionality. Software performs not only the primary functions that the user needs, but also network control, security, user-interface presentation and processing, and data or object management. The traditional "waterfall" approach to development, which assumes a linear progression of development activities, where one begins only when its predecessor is complete (and which we will study in Chapter 2), is no longer flexible or suitable for today's systems.

In his Stevens lecture, Wasserman (1996) summarized these changes by identifying seven key factors that have altered software engineering practice, illustrated in Figure 1.12:

1. criticality of time-to-market for commercial products
2. shifts in the economics of computing: lower hardware costs and greater development and maintenance costs
3. availability of powerful desktop computing
4. extensive local- and wide-area networking
5. availability and adoption of object-oriented technology
6. graphical user interfaces using windows, icons, menus, and pointers
7. unpredictability of the waterfall model of software development

For example, the pressures of the marketplace mean that businesses must ready their new products and services before their competitors do; otherwise, the viability of the business itself may be at stake. So traditional techniques for review and testing cannot be used if they require large investments of time that are not recouped as reduced fault or failure rates. Similarly, time previously spent in optimizing code to improve speed or

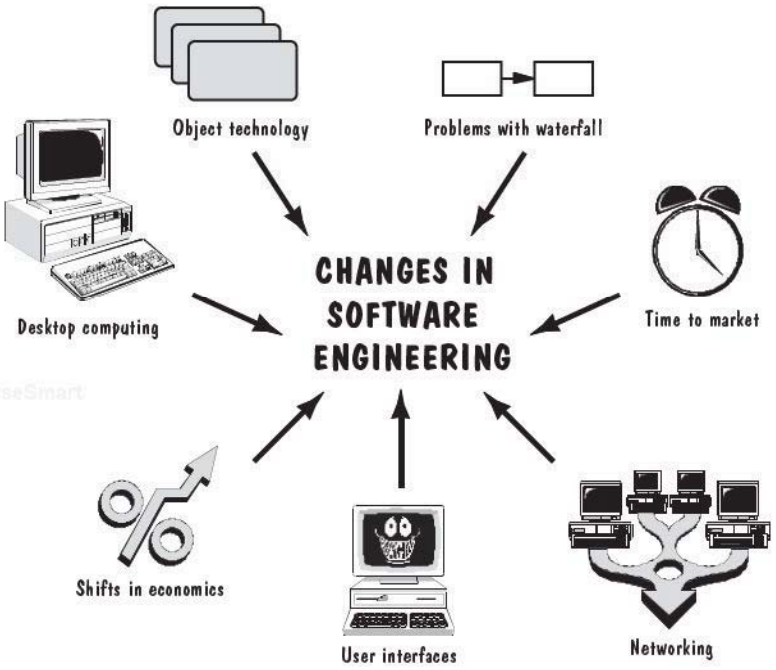


FIGURE 1.12 The key factors that have changed software development.

reduce space may no longer be a wise investment; an additional disk or memory card may be a far cheaper solution to the problem.

Moreover, desktop computing puts development power in the hands of users, who now use their systems to develop spreadsheet and database applications, small programs, and even specialized user interfaces and simulations. This shift of development responsibility means that we, as software engineers, are likely to be building more complex systems than before. Similarly, the vast networking capabilities available to most users and developers make it easier for users to find information without special applications. For instance, searching the World Wide Web is quick, easy, and effective; the user no longer needs to write a database application to find what he or she needs.

Developers now find their jobs enhanced, too. Object-oriented technology, coupled with networks and reuse repositories, makes available to developers a large collection of reusable modules for immediate and speedy inclusion in new applications. And graphical user interfaces, often developed with a specialized tool, help put a friendly face on complicated applications. Because we have become sophisticated in the way we analyze problems, we can now partition a system so we develop its subsystems in parallel, requiring a development process very different from the waterfall model. We will see in Chapter 2 that we have many choices for this process, including some that allow us to build prototypes (to verify with customers and users that the requirements are correct, and to assess the feasibility of designs) and iterate among activities. These steps help us to ensure that our requirements and designs are as fault-free as possible before we instantiate them in code.

Wasserman's Discipline of Software Engineering

Wasserman (1996) points out that any one of the seven technological changes would have a significant effect on the software development process. But taken together, they have transformed the way we work. In his presentations, DeMarco describes this radical shift by saying that we solved the easy problems first—which means that the set of problems left to be solved is much harder now than it was before. Wasserman addresses this challenge by suggesting that there are eight fundamental notions in software engineering that form the basis for an effective discipline of software engineering. We introduce them briefly here, and we return to them in later chapters to see where and how they apply to what we do.

Abstraction. Sometimes, looking at a problem in its "natural state" (i.e., as expressed by the customer or user) is a daunting task. We cannot see an obvious way to tackle the problem in an effective or even feasible way. An **abstraction** is a description of the problem at some level of generalization that allows us to concentrate on the key aspects of the problem without getting mired in the details. This notion is different from a **transformation**, where we translate the problem to another environment that we understand better; transformation is often used to move a problem from the real world to the mathematical world, so we can manipulate numbers to solve the problem.

Typically, abstraction involves identifying classes of objects that allow us to group items together; this way, we can deal with fewer things and concentrate on the commonalities of the items in each class. We can talk of the properties or attributes of the items in a class and examine the relationships among properties and classes. For example, suppose we are asked to build an environmental monitoring system for a large and complex river. The monitoring equipment may involve sensors for air quality, water quality, temperature, speed, and other characteristics of the environment. But, for our purposes, we may choose to define a class called "sensor"; each item in the class has certain properties, regardless of the characteristic it is monitoring: height, weight, electrical requirements, maintenance schedule, and so on. We can deal with the class, rather than its elements, in learning about the problem context, and in devising a solution. In this way, the classes help us to simplify the problem statement and focus on the essential elements or characteristics of the problem.

We can form hierarchies of abstractions, too. For instance, a sensor is a type of electrical device, and we may have two types of sensors: water sensors and air sensors.

Thus, we can form the simple hierarchy illustrated in Figure 1.13. By hiding some of the details, we can concentrate on the essential nature of the objects with which we must deal and derive solutions that are simple and elegant. We will take a closer look at abstraction and information hiding in Chapters 5, 6, and 7.

Analysis and Design Methods and Notations. When you design a program as a class assignment, you usually work on your own. The documentation that you produce is a formal description of your notes to yourself about why you chose a particular approach, what the variable names mean, and which algorithm you implemented. But when you work with a team, you must communicate with many other participants in the development process. Most engineers, no matter what kind of engineering they do,

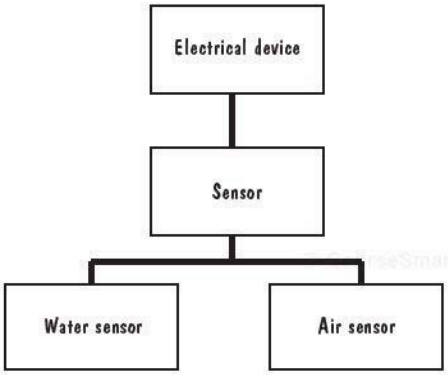


FIGURE 1.13 Simple hierarchy for monitoring equipment.

use a standard notation to help them communicate, and to document decisions. For example, an architect draws a diagram or blueprint that any other architect can understand. More importantly, the common notation allows the building contractor to understand the architect's intent and ideas. As we will see in Chapters 4, 5, 6, and 7, there are few similar standards in software engineering, and the misinterpretation that results is one of the key problems of software engineering today.

Analysis and design methods offer us more than a communication medium. They allow us to build models and check them for completeness and consistency. Moreover, we can more readily reuse requirements and design components from previous projects, increasing our productivity and quality with relative ease.

But there are many open questions to be resolved before we can settle on a common set of methods and tools. As we will see in later chapters, different tools and techniques address different aspects of a problem, and we need to identify the modeling primitives that will allow us to capture all the important aspects of a problem with a single technique. Or we need to develop a representation technique that can be used with all methods, possibly tailored in some way.

User Interface Prototyping. **Prototyping** means building a small version of a system, usually with limited functionality, that can be used to

- help the user or customer identify the key requirements of a system
- demonstrate the feasibility of a design or approach

Often, the prototyping process is iterative: We build a prototype, evaluate it (with user and customer feedback), consider how changes might improve the product or design, and then build another prototype. The iteration ends when we and our customers think we have a satisfactory solution to the problem at hand.

Prototyping is often used to design a good **user interface**: the part of the system with which the user interacts. However, there are other opportunities for using prototypes, even in **embedded systems** (i.e., in systems where the software functions are not explicitly visible to the user). The prototype can show the user what functions will be available, regardless of whether they are implemented in software or hardware. Since the user interface is, in a sense, a bridge between the application domain and the software

development team, prototyping can bring to the surface issues and assumptions that may not have been clear using other approaches to requirements analysis. We will consider the role of user interface prototyping in Chapters 4 and 5.

Software Architecture. The overall architecture of a system is important not only to the ease of implementing and testing it, but also to the speed and effectiveness of maintaining and changing it. The quality of the architecture can make or break a system; indeed, Shaw and Garlan (1996) present architecture as a discipline on its own whose effects are felt throughout the entire development process. The architectural structure of a system should reflect the principles of good design that we will study in Chapters 5 and 7.

A system's architecture describes the system in terms of a set of architectural units, and a map of how the units relate to one another. The more independent the units, the more modular the architecture and the more easily we can design and develop the pieces separately. Wasserman (1996) points out that there are at least five ways that we can partition the system into units:

1. modular decomposition: based on assigning functions to modules
2. data-oriented decomposition: based on external data structures
3. event-oriented decomposition: based on events that the system must handle
4. outside-in design: based on user inputs to the system
5. object-oriented design: based on identifying classes of objects and their interrelationships

These approaches are not mutually exclusive. For example, we can design a user interface with event-oriented decomposition, while we design the database using object-oriented or data-oriented design. We will examine these techniques in further detail in later chapters. The importance of these approaches is their capture of our design experience, enabling us to capitalize on our past projects by reusing both what we have done and what we learned by doing it.

Software Process. Since the late 1980s, many software engineers have paid careful attention to the *process* of developing software, as well as to the products that result. The organization and discipline in the activities have been acknowledged to contribute to the quality of the software and to the speed with which it is developed. However, Wasserman notes that

> the great variations among application types and organizational cultures make it impossible to be prescriptive about the process itself. Thus, it appears that the software process is not fundamental to software engineering in the same way as are abstraction and modularization. (Wasserman 1996)

Instead, he suggests that different types of software need different processes. In particular, Wasserman suggests that enterprisewide applications need a great deal of control, whereas individual and departmental applications can take advantage of rapid application development, as we illustrate in Figure 1.14.

By using today's tools, many small and medium-sized systems can be built by one or two developers, each of whom must take on multiple roles. The tools may include a

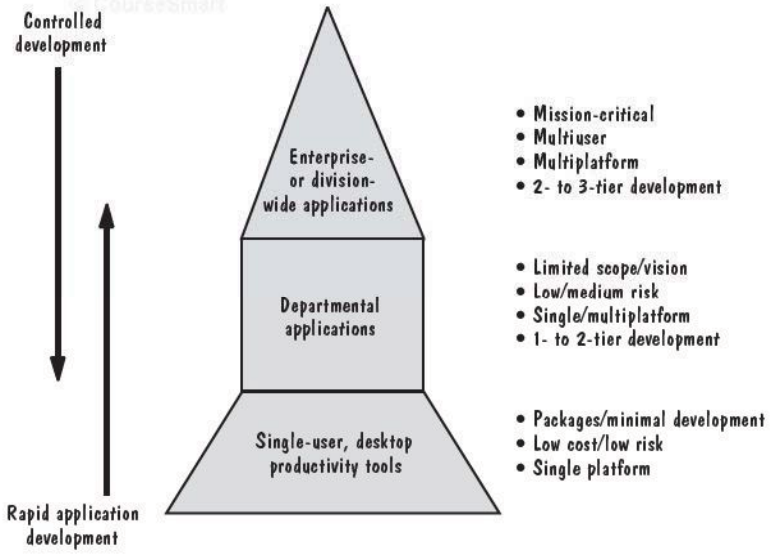


FIGURE 1.14 Differences in development (Wasserman 1996).

text editor, programming environment, testing support, and perhaps a small database to capture key data elements about the products and processes. Because the project's risk is relatively low, little management support or review is needed.

However, large, complex systems need more structure, checks, and balances. These systems often involve many customers and users, and development continues over a long period of time. Moreover, the developers do not always have control over the entire development, as some critical subsystems may be supplied by others or be implemented in hardware. This type of high-risk system requires analysis and design tools, project management, configuration management, more sophisticated testing tools, and a more rigorous system of review and causal analysis. In Chapter 2, we will take a careful look at several process alternatives to see how varying the process addresses different goals. Then, in Chapters 12 and 13, we evaluate the effectiveness of some processes and look at ways to improve them.

Reuse. In software development and maintenance, we often take advantage of the commonalities across applications by reusing items from previous development. For example, we use the same operating system or database management system from one development project to the next, rather than building a new one each time. Similarly, we reuse sets of requirements, parts of designs, and groups of test scripts or data when we build systems that are similar to but not the same as what we have done before. Barnes and Bollinger (1991) point out that reuse is not a new idea, and they provide many interesting examples of how we reuse much more than just code.

Prieto-Díaz (1991) introduced the notion of reusable components as a business asset. Companies and organizations invest in items that are reusable and then gain quantifiable benefit when those items are used again in subsequent projects. However,

establishing a long-term, effective reuse program can be difficult, because there are several barriers:

- It is sometimes faster to build a small component than to search for one in a repository of reusable components.
- It may take extra time to make a component general enough to be reusable easily by other developers in the future.
- It is difficult to document the degree of quality assurance and testing that have been done, so that a potential reuser can feel comfortable about the quality of the component.
- It is not clear who is responsible if a reused component fails or needs to be updated.
- It can be costly and time-consuming to understand and reuse a component written by someone else.
- There is often a conflict between generality and specificity.

We will look at reuse in more detail in Chapter 12, examining several examples of successful reuse.

Measurement. Improvement is a driving force in software engineering research: improving our processes, resources, and methods so that we produce and maintain better products. But sometimes we express improvement goals generally, with no quantitative description of where we are and where we would like to go. For this reason, software measurement has become a key aspect of good software engineering practice. By quantifying where we can and what we can, we describe our actions and their outcomes in a common mathematical language that allows us to evaluate our progress. In addition, a quantitative approach permits us to compare progress across disparate projects. For example, when John Young was CEO of Hewlett-Packard, he set goals of "10X," a tenfold improvement in quality and productivity, for every project at Hewlett-Packard, regardless of application type or domain (Grady and Caswell 1987).

At a lower level of abstraction, measurement can help to make specific characteristics of our processes and products more visible. It is often useful to transform our understanding of the real, empirical world to elements and relationships in the formal, mathematical world, where we can manipulate them to gain further understanding. As illustrated in Figure 1.15, we can use mathematics and statistics to solve a problem, look for trends, or characterize a situation (such as with means and standard deviations). This new information can then be mapped back to the real world and applied as part of a solution to the empirical problem we are trying to solve. Throughout this book, we will see examples of how measurement is used to support analysis and decision making.

Tools and Integrated Environments. For many years, vendors touted CASE (Computer-Aided Software Engineering) tools, where standardized, integrated development environments would enhance software development. However, we have seen how different developers use different processes, methods, and resources, so a unifying approach is easier said than done.

On the other hand, researchers have proposed several frameworks that allow us to compare and contrast both existing and proposed environments. These frameworks

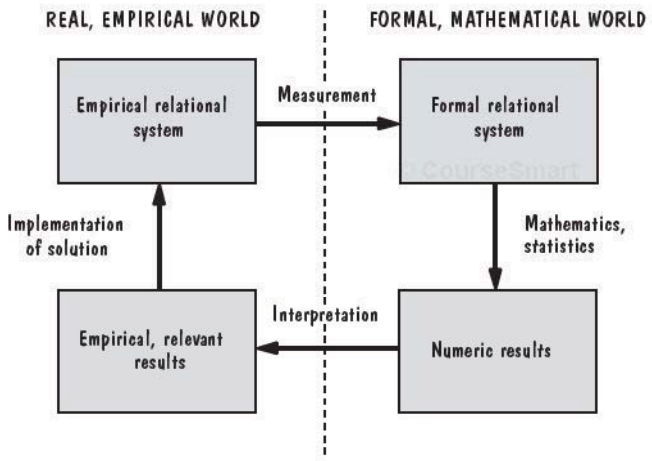


FIGURE 1.15 Using measurement to help find a solution.

permit us to examine the services provided by each software engineering environment and to decide which environment is best for a given problem or application development.

One of the major difficulties in comparing tools is that vendors rarely address the entire development life cycle. Instead, they focus on a small set of activities, such as design or testing, and it is up to the user to integrate the selected tools into a complete development environment. Wasserman (1990) has identified five issues that must be addressed in any tool integration:

1. platform integration: the ability of tools to interoperate on a heterogeneous network
2. presentation integration: commonality of user interface
3. process integration: linkage between the tools and the development process
4. data integration: the way tools share data
5. control integration: the ability for one tool to notify and initiate action in another

In each of the subsequent chapters of this book, we will examine tools that support the activities and concepts we describe in the chapter.

You can think of the eight concepts described here as eight threads woven through the fabric of this book, tying together the disparate activities we call software engineering. As we learn more about software engineering, we will revisit these ideas to see how they unify and elevate software engineering as a scientific discipline.

1.9 INFORMATION SYSTEMS EXAMPLE

Throughout this book, we will end each chapter with two examples, one of an information system and the other of a real-time system. We will apply the concepts described in the chapter to some aspect of each example, so that you can see what the concepts mean in practice, not just in theory.

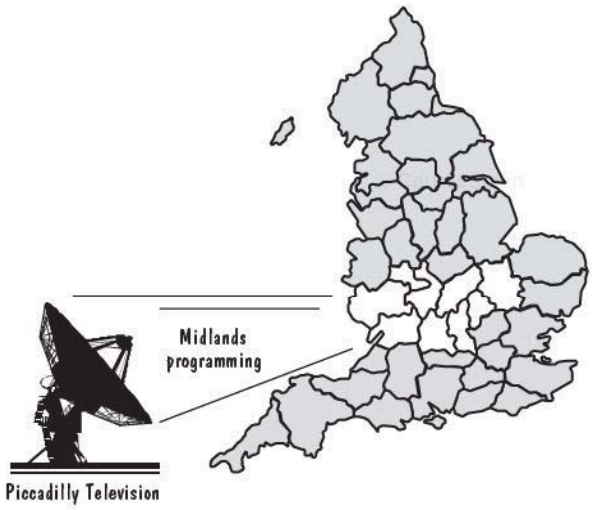


FIGURE 1.16 Piccadilly Television franchise area.

Our information system example is drawn (with permission) from *Complete Systems Analysis: The Workbook, the Textbook, the Answers,* by James and Suzanne Robertson (Robertson and Robertson 1994). It involves the development of a system to sell advertising time for Piccadilly Television, the holder of a regional British television franchise. Figure 1.16 illustrates the Piccadilly Television viewing area. As we shall see, the constraints on the price of television time are many and varied, so the problem is both interesting and difficult. In this book, we highlight aspects of the problem and its solution; the Robertsons' book shows you detailed methods for capturing and analyzing the system requirements.

In Britain, the broadcasting board issues an eight-year franchise to a commercial television company, giving it exclusive rights to broadcast its programs in a carefully defined region of the country. In return, the franchisee must broadcast a prescribed balance of drama, comedy, sports, children's and other programs. Moreover, there are restrictions on which programs can be broadcast at which times, as well as rules about the content of programs and commercial advertising.

A commercial advertiser has several choices to reach the Midlands audience: Piccadilly, the cable channels, and the satellite channels. However, Piccadilly attracts most of the audience. Thus, Piccadilly must set its rates to attract a portion of an advertiser's national budget. One of the ways to attract an advertiser's attention is with audience ratings that reflect the number and type of viewers at different times of the day. The ratings are reported in terms of program type, audience type, time of day, television company, and more. But the advertising rate depends on more than just the ratings. For example, the rate per hour may be cheaper if the advertiser buys a large number of hours. Moreover, there are restrictions on the type of advertising at certain times and for certain programs. For example,

- Advertisements for alcohol may be shown only after 9 P.M.
- If an actor is in a show, then an advertisement with that actor may not be broadcast within 45 minutes of the show.

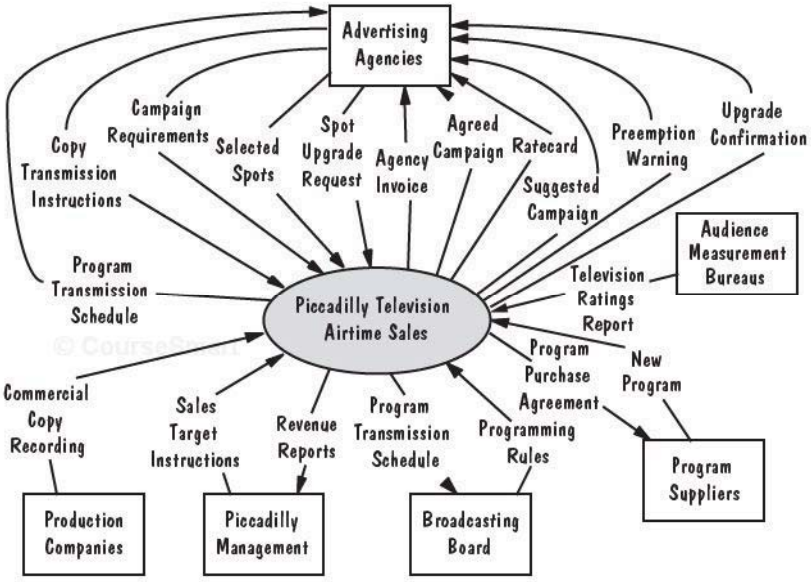


FIGURE 1.17 Piccadilly context diagram showing system boundary (Robertson and Robertson 1994).

- If an advertisement for a class of product (such as an automobile) is scheduled for a particular commercial break, then no other advertisement for something in that class may be shown during that break.

As we explore this example in more detail, we will note the additional rules and regulations about advertising and its cost. The system context diagram in Figure 1.17 shows us the system boundary and how it relates to these rules. The shaded oval is the Piccadilly system that concerns us as our information system example; the system boundary is simply the perimeter of the oval. The arrows and boxes display the items that can affect the working of the Piccadilly system, but we consider them only as a collection of inputs and outputs, with their sources and destinations, respectively.

In later chapters, we will make visible the activities and elements inside the shaded oval (i.e., within the system boundary). We will examine the design and development of this system using the software engineering techniques that are described in each chapter.

1.10 REAL-TIME EXAMPLE

Our real-time example is based on the embedded software in the Ariane-5, a space rocket belonging to the European Space Agency (ESA). On June 4, 1996, on its maiden flight, the Ariane-5 was launched and performed perfectly for approximately 40 seconds. Then, it began to veer off course. At the direction of an Ariane ground controller, the rocket was destroyed by remote control. The destruction of the uninsured rocket was a loss not only of the rocket itself, but also of the four satellites it

contained; the total cost of the disaster was $500 million (Lions et al. 1996; Newsbytes home page 1996).

Software is involved in almost all aspects of the system, from the guidance of the rocket to the internal workings of its component parts. The failure of the rocket and its subsequent destruction raise many questions about software quality. As we will see in later chapters, the inquiry board that investigated the cause of the problem focused on software quality and its assurance. In this chapter, we look at quality in terms of the business value of the rocket.

There were many organizations with a stake in the success of Ariane-5: the ESA, the Centre National d'Etudes Spatiales (CNES, the French space agency in overall command of the Ariane program), and 12 other European countries. The rocket's loss was another in a series of delays and problems to affect the Ariane program, including a nitrogen leak during engine testing in 1995 that killed two engineers. However, the June 1996 incident was the first whose cause was directly attributed to software failure.

The business impact of the incident went well beyond the $500 million in equipment. In 1996, the Ariane-4 rocket and previous variants held more than half of the world's launch contracts, ahead of American, Russian, and Chinese launchers. Thus, the credibility of the program was at stake, as well as the potential business from future Ariane rockets.

The future business was based in part on the new rocket's ability to carry heavier payloads into orbit than previous launchers could. Ariane-5 was designed to carry a single satellite up to 6.8 tons or two satellites with a combined weight of 5.9 tons. Further development work hoped to add an extra ton to the launch capacity by 2002. This increased carrying capacity has clear business advantages; often, operators reduce their costs by sharing launches, so Ariane can offer to host several companies' payloads at the same time.

Consider what quality means in the context of this example. The destruction of Ariane-5 turned out to be the result of a requirement that was misspecified by the customer. In this case, the developer might claim that the system is still high quality; it was just built to the wrong specification. Indeed, the inquiry board formed to investigate the cause and cure of the disaster noted that

> The Board's findings are based on thorough and open presentations from the Ariane-5 project teams, and on documentation which has demonstrated the high quality of the Ariane-5 programme as regards engineering work in general and completeness and traceability of documents. (Lions et al. 1996)

But from the user's and customer's point of view, the specification process should have been good enough to identify the specification flaw and force the customer to correct the specification before damage was done. The inquiry board acknowledged that

> The supplier of the SRI [the subsystem in which the cause of the problem was eventually located] was only following the specification given to it, which stipulated that in the event of any detected exception the processor was to be stopped. The exception which occurred was not due to random failure but a design error. The exception was detected, but inappropriately handled because the view had been taken that software should be considered correct until it is shown to be at fault. The Board has reason to believe that this view is also accepted in other areas of Ariane-5 software design. The Board is in favour of the opposite view, that software should be assumed to be faulty until applying the currently accepted best practice methods can demonstrate that it is correct. (Lions et al. 1996)

In later chapters, we will investigate this example in more detail, looking at the design, testing, and maintenance implications of the developers' and customers' decisions. We will see how poor systems engineering at the beginning of development led to a series of poor decisions that led in turn to disaster. On the other hand, the openness of all concerned, including ESA and the inquiry board, coupled with high-quality documentation and an earnest desire to get at the truth quickly, resulted in quick resolution of the immediate problem and an effective plan to prevent such problems in the future.

A systems view allowed the inquiry board, in cooperation with the developers, to view the Ariane-5 as a collection of subsystems. This collection reflects the analysis of the problem, as we described in this chapter, so that different developers can work on separate subsystems with distinctly different functions. For example:

> The attitude of the launcher and its movements in space are measured by an Inertial Reference System (SRI). It has its own internal computer, in which angles and velocities are calculated on the basis of information from a "strap-down" inertial platform, with laser gyros and accelerometers. The data from the SRI are transmitted through the databus to the On-Board Computer (OBC), which executes the flight program and controls the nozzles of the solid boosters and the Vulcain cryogenic engine, via servovalves and hydraulic actuators. (Lions et al. 1996)

But the synthesis of the solution must include an overview of all the component parts, where the parts are viewed together to determine if the "glue" that holds them together is sufficient and appropriate. In the case of Ariane-5, the inquiry board suggested that the customers and developers should have worked together to find the critical software and make sure that it could handle not only anticipated but also unanticipated behavior.

> This means that critical software—in the sense that failure of the software puts the mission at risk—must be identified at a very detailed level, that exceptional behaviour must be confined, and that a reasonable back-up policy must take software failures into account. (Lions et al. 1996)

1.11 WHAT THIS CHAPTER MEANS FOR YOU

This chapter has introduced many concepts that are essential to good software engineering research and practice. You, as an individual software developer, can use these concepts in the following ways:

- When you are given a problem to solve (whether or not the solution involves software), you can analyze the problem by breaking it into its component parts, and the relationships among the parts. Then, you can synthesize a solution by solving the individual subproblems and merging them to form a unified whole.
- You must understand that the requirements may change, even as you are analyzing the problem and building a solution. So your solution should be well-documented and flexible, and you should document your assumptions and the algorithms you use (so that they are easy to change later).
- You must view quality from several different perspectives, understanding that technical quality and business quality may be very different.

- You can use abstraction and measurement to help identify the essential aspects of the problem and solution.
- You can keep the system boundary in mind, so that your solution does not overlap with the related systems that interact with the one you are building.

1.12 WHAT THIS CHAPTER MEANS FOR YOUR DEVELOPMENT TEAM

Much of your work will be done as a member of a larger development team. As we have seen in this chapter, development involves requirements analysis, design, implementation, testing, configuration management, quality assurance, and more. Some of the people on your team may wear multiple hats, as may you, and the success of the project depends in large measure on the communication and coordination among the team members. We have seen in this chapter that you can aid the success of your project by selecting

- a development process that is appropriate to your team size, risk level, and application domain
- tools that are well-integrated and support the type of communication your project demands
- measurements and supporting tools to give you as much visibility and understanding as possible

1.13 WHAT THIS CHAPTER MEANS FOR RESEARCHERS

Many of the issues discussed in this chapter are good subjects for further research. We have noted some of the open issues in software engineering, including the need to find

- the right levels of abstraction to make the problem easy to solve
- the right measurements to make the essential nature of the problem and solution visible and helpful
- an appropriate problem decomposition, where each subproblem is solvable
- a common framework or notation to allow easy and effective tool integration, and to maximize communication among project participants

In later chapters, we will describe many techniques. Some have been used and are well-proven software development practices, whereas others are proposed and have been demonstrated only on small, "toy," or student projects. We hope to show you how to improve what you are doing now and at the same time to inspire you to be creative and thoughtful about trying new techniques and processes in the future.

1.14 TERM PROJECT

It is impossible to learn software engineering without participating in developing a software project with your colleagues. For this reason, each chapter of this book will present information about a term project that you can perform with a team of classmates. The project, based on a real system in a real organization, will allow you to address some of

the very real challenges of analysis, design, implementation, testing, and maintenance. In addition, because you will be working with a team, you will deal with issues of team diversity and project management.

The term project involves the kinds of loans you might negotiate with a bank when you want to buy a house. Banks generate income in many ways, often by borrowing money from their depositors at a low interest rate and then lending that same money back at a higher interest rate in the form of bank loans. However, long-term property loans, such as mortgages, typically have terms of up to 15, 25, or even 30 years. That is, you have 15, 25, or 30 years to repay the loan: the principal (the money you originally borrowed) plus interest at the specified rate. Although the income from interest on these loans is lucrative, the loans tie up money for a long time, preventing the banks from using their money for other transactions. Consequently, the banks often sell their loans to consolidating organizations, taking less long-term profit in exchange for freeing the capital for use in other ways.

The application for your term project is called the Loan Arranger. It is fashioned on ways in which a (mythical) Financial Consolidation Organization (FCO) handles the loans it buys from banks. The consolidation organization makes money by purchasing loans from banks and selling them to investors. The bank sells the loan to FCO, getting the principal in return. Then, as we shall see, FCO sells the loan to investors who are willing to wait longer than the bank to get their return.

To see how the transactions work, consider how you get a loan (called a "mortgage") for a house. You may purchase a $150,000 house by paying $50,000 as an initial payment (called the "down payment") and taking a loan for the remaining $100,000. The "terms" of your loan from the First National Bank may be for 30 years at 5% interest. This terminology means that the First National Bank gives you 30 years (the term of the loan) to pay back the amount you borrowed (the "principal") plus interest on whatever you do not pay back right away. For example, you can pay the $100,000 by making a payment once a month for 30 years (that is, 360 "installments" or "monthly payments"), with interest on the unpaid balance. If the initial balance is $100,000, the bank calculates your monthly payment using the amount of principal, the interest rate, the amount of time you have to pay off the loan, and the assumption that all monthly payments should be the same amount.

For instance, suppose the bank tells you that your monthly payment is to be $536.82. The first month's interest is $(1/12) \times (.05) \times (\$100{,}000)$, or $416.67. The rest of the payment ($536.82 − 416.67) pays for reducing the principal: $120.15. For the second month, you now owe $100,000 minus the $120.15, so the interest is reduced to $(1/12) \times (.05) \times (\$100{,}000 - 120.15)$, or $416.17. Thus, during the second month, only $416.17 of the monthly payment is interest, and the remainder, $120.65, is applied to the remaining principal. Over time, you pay less interest and more toward reducing the remaining balance of principal, until you have paid off the entire principal and own your property free and clear of any encumbrance by the bank.

First National Bank may sell your loan to FCO some time during the period when you are making payments. First National negotiates a price with FCO. In turn, FCO may sell your loan to ABC Investment Corporation. You still make your mortgage payments each month, but your payment goes to ABC, not First National. Usually, FCO sells its loans in "bundles," not individual loans, so that an investor buys a collection of loans based on risk, principal involved, and expected rate of return. In other words, an

investor such as ABC can contact FCO and specify how much money it wishes to invest, for how long, how much risk it is willing to take (based on the history of the people or organizations paying back the loan), and how much profit is expected.

The Loan Arranger is an application that allows an FCO analyst to select a bundle of loans to match an investor's desired investment characteristics. The application accesses information about loans purchased by FCO from a variety of lending institutions. When an investor specifies investment criteria, the system selects the optimal bundle of loans that satisfies the criteria. While the system will allow some advanced optimizations, such as selecting the best bundle of loans from a subset of those available (for instance, from all loans in Massachusetts, rather than from all the loans available), the system will still allow an analyst to manually select loans in a bundle for the client. In addition to bundle selection, the system also automates information management activities, such as updating bank information, updating loan information, and adding new loans when banks provide that information each month.

We can summarize this information by saying that the Loan Arranger system allows a loan analyst to access information about mortgages (home loans, described here simply as "loans") purchased by FCO from multiple lending institutions with the intention of repackaging the loans to sell to other investors. The loans purchased by FCO for investment and resale are collectively known as the loan portfolio. The Loan Arranger system tracks these portfolio loans in its repository of loan information. The loan analyst may add, view, update, or delete loan information about lenders and the set of loans in the portfolio. Additionally, the system allows the loan analyst to create "bundles" of loans for sale to investors. A user of Loan Arranger is a loan analyst who tracks the mortgages purchased by FCO.

In later chapters, we will explore the system's requirements in more depth. For now, if you need to brush up on your understanding of principal and interest, you can review your old math books or look at http://www.interest.com/hugh/calc/formula.html.

1.15 KEY REFERENCES

You can find out about software faults and failures by looking in the Risks Forum, moderated by Peter Neumann. A paper copy of some of the Risks is printed in each issue of *Software Engineering Notes*, published by the Association for Computer Machinery's Special Interest Group on Software Engineering (SIGSOFT). The Risks archives are available on ftp.sri.com, cd risks. The Risks Forum newsgroup is available online at comp.risks, or you can subscribe via the automated list server at risks-request@CSL.sri.com.

You can find out more about the Ariane-5 project from the European Space Agency's Web site: http://www.esrin.esa.it/htdocs/esa/ariane. A copy of the joint ESA/CNES press release describing the mission failure (in English) is at http://www.esrin.esa.it/htdocs/tidc/Press/Press96/press19.html. A French version of the press release is at http://www.cnes.fr/Acces_Espace/Vol_50x.html. An electronic copy of the Ariane-5 Flight 501 Failure Report is at http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html.

Leveson and Turner (1993) describe the Therac software design and testing problems in careful detail.

The January 1996 issue of *IEEE Software* is devoted to software quality. In particular, the introductory article by Kitchenham and Pfleeger (1996) describes and critiques several quality frameworks, and the article by Dromey (1996) discusses how to define quality in a measurable way.

For more information about the Piccadilly Television example, you may consult (Robertson and Robertson 1994) or explore the Robertsons' approach to requirements at www.systemsguild.com.

1.16 EXERCISES

1. The following article appeared in the *Washington Post* (Associated Press 1996):

PILOT'S COMPUTER ERROR CITED IN PLANE CRASH. AMERICAN AIRLINES SAYS ONE-LETTER CODE WAS REASON JET HIT MOUNTAIN IN COLOMBIA.

Dallas, Aug. 23—The captain of an American Airlines jet that crashed in Colombia last December entered an incorrect one-letter computer command that sent the plane into a mountain, the airline said today.

The crash killed all but four of the 163 people aboard.

American's investigators concluded that the captain of the Boeing 757 apparently thought he had entered the coordinates for the intended destination, Cali.

But on most South American aeronautical charts, the one-letter code for Cali is the same as the one for Bogota, 132 miles in the opposite direction.

The coordinates for Bogota directed the plane toward the mountain, according to a letter by Cecil Ewell, American's chief pilot and vice president for flight. The codes for Bogota and Cali are different in most computer databases, Ewell said.

American spokesman John Hotard confirmed that Ewell's letter, first reported in the *Dallas Morning News*, is being delivered this week to all of the airline's pilots to warn them of the coding problem.

American's discovery also prompted the Federal Aviation Administration to issue a bulletin to all airlines, warning them of inconsistencies between some computer databases and aeronautical charts, the newspaper said.

The computer error is not the final word on what caused the crash. The Colombian government is investigating and is expected to release its findings by October.

Pat Cariseo, spokesman for the National Transportation Safety Board, said Colombian investigators also are examining factors such as flight crew training and air traffic control.

The computer mistake was found by investigators for American when they compared data from the jet's navigation computer with information from the wreckage, Ewell said.

The data showed the mistake went undetected for 66 seconds while the crew scrambled to follow an air traffic controller's orders to take a more direct approach to the Cali airport.

Three minutes later, while the plane still was descending and the crew trying to figure out why the plane had turned, it crashed.

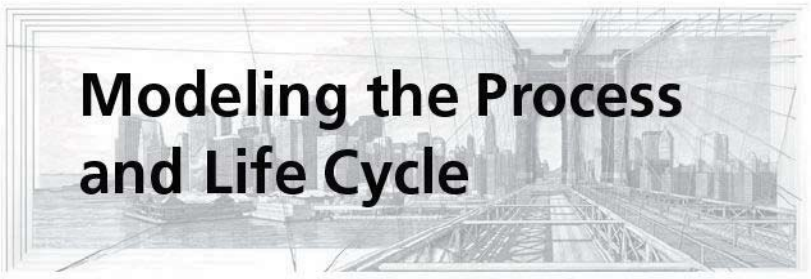
Ewell said the crash presented two important lessons for pilots.

"First of all, no matter how many times you go to South America or any other place—the Rocky Mountains—you can never, never, never assume anything," he told the newspaper. Second, he said, pilots must understand they can't let automation take over responsibility for flying the airplane.

Is this article evidence that we have a software crisis? How is aviation better off because of software engineering? What issues should be addressed during software development so that problems like this will be prevented in the future?

2. Give an example of problem analysis where the problem components are relatively simple, but the difficulty in solving the problem lies in the interconnections among subproblem components.
3. Explain the difference between errors, faults, and failures. Give an example of an error that leads to a fault in the requirements; the design; the code. Give an example of a fault in the requirements that leads to a failure; a fault in the design that leads to a failure; a fault in the test data that leads to a failure.
4. Why can a count of faults be a misleading measure of product quality?
5. Many developers equate technical quality with overall product quality. Give an example of a product with high technical quality that is not considered high quality by the customer. Are there ethical issues involved in narrowing the view of quality to consider only technical quality? Use the Therac-25 example to illustrate your point.
6. Many organizations buy commercial software, thinking it is cheaper than developing and maintaining software in-house. Describe the pros and cons of using COTS software. For example, what happens if the COTS products are no longer supported by their vendors? What must the customer, user, and developer anticipate when designing a product that uses COTS software in a large system?
7. What are the legal and ethical implications of using COTS software? Of using subcontractors? For example, who is responsible for fixing the problem when the major system fails as a result of a fault in COTS software? Who is liable when such a failure causes harm to the users, directly (as when the automatic brakes fail in a car) or indirectly (as when the wrong information is supplied to another system, as we saw in Exercise 1). What checks and balances are needed to ensure the quality of COTS software before it is integrated into a larger system?
8. The Piccadilly Television example, as illustrated in Figure 1.17, contains a great many rules and constraints. Discuss three of them and explain the pros and cons of keeping them outside the system boundary.
9. When the Ariane-5 rocket was destroyed, the news made headlines in France and elsewhere. *Liberation,* a French newspaper, called it "A 37-billion-franc fireworks display" on the front page. In fact, the explosion was front-page news in almost all European newspapers and headed the main evening news bulletins on most European TV networks. By contrast, the invasion by a hacker of Panix, a New York-based Internet provider, forced the Panix system to close down for several hours. News of this event appeared only on the front page of the business section of the *Washington Post*. What is the responsibility of the press when reporting software-based incidents? How should the potential impact of software failures be assessed and reported?

2 Modeling the Process and Life Cycle



In this chapter, we look at
- what we mean by a "process"
- software development products, processes, and resources
- several models of the software development process
- tools and techniques for process modeling

We saw in Chapter 1 that engineering software is both a creative and a step-by-step process, often involving many people producing many different kinds of products. In this chapter, we examine the steps in more detail, looking at ways to organize our activities, so that we can coordinate what we do and when we do it. We begin the chapter by defining what we mean by a process, so that we understand what must be included when we model software development. Next, we examine several types of software process models. Once we know the type of model we wish to use, we take a close look at two types of modeling techniques: static and dynamic. Finally, we apply several of these techniques to our information systems and real-time examples.

2.1 THE MEANING OF PROCESS

When we provide a service or create a product, whether it be developing software, writing a report, or taking a business trip, we always follow a sequence of steps to accomplish a set of tasks. The tasks are usually performed in the same order each time; for example, you do not usually put up the drywall before the wiring for a house is installed or bake a cake before all the ingredients are mixed together. We can think of a set of ordered tasks as a **process**: a series of steps involving activities, constraints, and resources that produce an intended output of some kind.

A process usually involves a set of tools and techniques, as we defined them in Chapter 1. Any process has the following characteristics:

- The process prescribes all of the major process activities.
- The process uses resources, subject to a set of constraints (such as a schedule), and produces intermediate and final products.
- The process may be composed of subprocesses that are linked in some way. The process may be defined as a hierarchy of processes, organized so that each subprocess has its own process model.
- Each process activity has entry and exit criteria, so that we know when the activity begins and ends.
- The activities are organized in a sequence, so that it is clear when one activity is performed relative to the other activities.
- Every process has a set of guiding principles that explain the goals of each activity.
- Constraints or controls may apply to an activity, resource, or product. For example, the budget or schedule may constrain the length of time an activity may take or a tool may limit the way in which a resource may be used.

When the process involves the building of some product, we sometimes refer to the process as a **life cycle**. Thus, the software development process is sometimes called the **software life cycle**, because it describes the life of a software product from its conception to its implementation, delivery, use, and maintenance.

Processes are important because they impose consistency and structure on a set of activities. These characteristics are useful when we know how to do something well and we want to ensure that others do it the same way. For example, if Sam is a good bricklayer, he may write down a description of the bricklaying process he uses so that Sara can learn how to do it as well. He may take into account the differences in the way people prefer to do things; for instance, he may write his instructions so that Sara can lay bricks whether she is right- or left-handed. Similarly, a software development process can be described in flexible ways that allow people to design and build software using preferred techniques and tools; a process model may require design to occur before coding, but may allow many different design techniques to be used. For this reason, the process helps to maintain a level of consistency and quality in products or services that are produced by many different people.

A process is more than a procedure. We saw in Chapter 1 that a procedure is like a recipe: a structured way of combining tools and techniques to produce a product. A process is a collection of procedures, organized so that we build products to satisfy a set of goals or standards. In fact, the process may suggest that we choose from several procedures, as long as the goal we are addressing is met. For instance, the process may require that we check our design components before coding begins. The checking can be done using informal reviews or formal inspections, each an activity with its own procedure, but both addressing the same goal.

The process structure guides our actions by allowing us to examine, understand, control, and improve the activities that comprise the process. To see how, consider the process of making chocolate cake with chocolate icing. The process may contain several procedures, such as buying the ingredients and finding the appropriate cooking utensils.

The recipe describes the procedure for actually mixing and baking the cake. The recipe contains activities (such as "beat the egg before mixing with other ingredients"), constraints (such as the temperature requirement in "heat the chocolate to the melting point before combining with the sugar"), and resources (such as sugar, flour, eggs, and chocolate). Suppose Chuck bakes a chocolate cake according to this recipe. When the cake is done, he tastes a sample and decides that the cake is too sweet. He looks at the recipe to see which ingredient contributes to the sweetness: sugar. Then, he bakes another cake, but this time he reduces the amount of sugar in the new recipe. Again he tastes the cake, but now it does not have enough chocolate flavor. He adds a measure of cocoa powder to his second revision and tries again. After several iterations, each time changing an ingredient or an activity (such as baking the cake longer, or letting the chocolate mixture cool before combining with the egg mixture), Chuck arrives at a cake to his liking. Without the recipe to document this part of the process, Chuck would not have been able to make changes easily and evaluate the results.

Processes are also important for enabling us to capture our experiences and pass them along to others. Just as master chefs pass on their favorite recipes to their colleagues and friends, master craftspeople can pass along documented processes and procedures. Indeed, the notions of apprenticeship and mentoring are based on the idea that we share our experience so we can pass down our skills from senior people to junior ones.

In the same way, we want to learn from our past development projects, document the practices that work best to produce high-quality software, and follow a software development process so we can understand, control, and improve what happens as we build products for our customers. We saw in Chapter 1 that software development usually involves the following stages:

- requirements analysis and definition
- system design
- program design
- writing the programs (program implementation)
- unit testing
- integration testing
- system testing
- system delivery
- maintenance

Each stage is itself a process (or collection of processes) that can be described as a set of activities. And each activity involves constraints, outputs, and resources. For example, the requirements analysis and definitions stage need as initial input a statement of desired functions and features that the user expresses in some way. The final output from this stage is a set of requirements, but there may be intermediate products as the dialog between user and developer results in changes and alternatives. We have constraints, too, such as a budget and schedule for producing the requirements document, and standards about the kinds of requirements to include and perhaps the notation used to express them.

Each of these stages is addressed in this book. For each one, we will take a close look at the processes, resources, activities, and outputs that are involved, and we will learn how

they contribute to the quality of the final product: useful software. There are many ways to address each stage of development; each configuration of activities, resources, and outputs constitutes a process, and a collection of processes describes what happens at each stage. For instance, design can involve a prototyping process, where many of the design decisions are explored so that developers can choose an appropriate approach, and a reuse process, where previously generated design components are included in the current design.

Each process can be described in a variety of ways, using text, pictures, or a combination. Software engineering researchers have suggested a variety of formats for such descriptions, usually organized as a model that contains key process features. For the remainder of this chapter, we examine a variety of software development process models, to see how organizing process activities can make development more effective.

2.2 SOFTWARE PROCESS MODELS

Many process models are described in the software engineering literature. Some are *prescriptions* for the way software development should progress, and others are *descriptions* of the way software development is done in actuality. In theory, the two kinds of models should be similar or the same, but in practice, they are not. Building a process model and discussing its subprocesses help the team understand the gap between what should be and what is.

There are several other reasons for modeling a process:

- When a group writes down a description of its development process, it forms a common understanding of the activities, resources, and constraints involved in software development.
- Creating a process model helps the development team find inconsistencies, redundancies, and omissions in the process and in its constituent parts. As these problems are noted and corrected, the process becomes more effective and focused on building the final product.
- The model should reflect the goals of development, such as building high-quality software, finding faults early in development, and meeting required budget and schedule constraints. As the model is built, the development team evaluates candidate activities for their appropriateness in addressing these goals. For example, the team may include requirements reviews, so that problems with the requirements can be found and fixed before design begins.
- Every process should be tailored for the special situation in which it will be used. Building a process model helps the development team understand where that tailoring is to occur.

Every software development process model includes system requirements as input and a delivered product as output. Many such models have been proposed over the years. Let us look at several of the most popular models to understand their commonalities and differences.

Waterfall Model

One of the first models to be proposed is the **waterfall model**, illustrated in Figure 2.1, where the stages are depicted as cascading from one to another (Royce 1970). As the

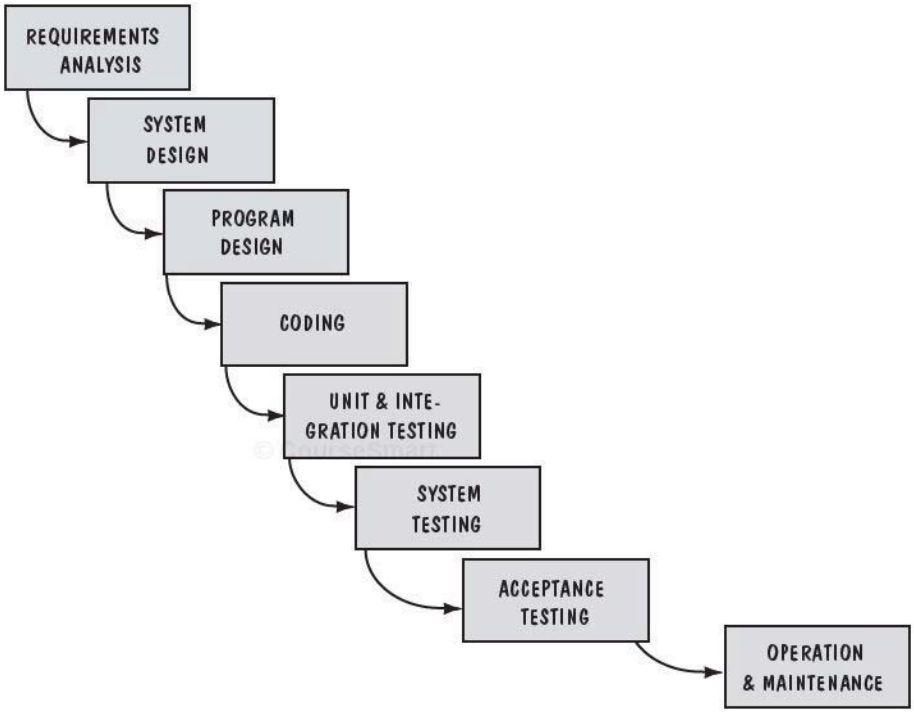


FIGURE 2.1 The waterfall model.

figure implies, one development stage should be completed before the next begins. Thus, when all of the requirements are elicited from the customer, analyzed for completeness and consistency, and documented in a requirements document, then the development team can go on to system design activities. The waterfall model presents a very high-level view of what goes on during development, and it suggests to developers the sequence of events they should expect to encounter.

The waterfall model has been used to prescribe software development activities in a variety of contexts. For example, it was the basis for software development deliverables in U.S. Department of Defense contracts for many years, defined in Department of Defense Standard 2167-A. Associated with each process activity were milestones and deliverables, so that project managers could use the model to gauge how close the project was to completion at a given point in time. For instance, "unit and integration testing" in the waterfall ends with the milestone "code modules written, tested, and integrated"; the intermediate deliverable is a copy of the tested code. Next, the code can be turned over to the system testers so it can be merged with other system components (hardware or software) and tested as a larger whole.

The waterfall model can be very useful in helping developers lay out what they need to do. Its simplicity makes it easy to explain to customers who are not familiar with software development; it makes explicit which intermediate products are necessary in order to begin the next stage of development. Many other, more complex

models are really just embellishments of the waterfall, incorporating feedback loops and extra activities.

Many problems with the waterfall model have been discussed in the literature, and two of them are summarized in Sidebar 2.1. The biggest problem with the waterfall model is that it does not reflect the way code is really developed. Except for very well-understood problems, software is usually developed with a great deal of iteration. Often, software is used in a solution to a problem that has never before been solved or whose solution must be upgraded to reflect some change in business climate or operating environment. For example, an airplane manufacturer may require software for a new airframe that will be bigger or faster than existing models, so there are new challenges to address, even though the software developers have a great deal of experience in building aeronautical software. Neither the users nor the developers know all the key factors that affect the desired outcome, and much of the time spent during requirements analysis, as we will see in Chapter 4, may be devoted to understanding the items and processes affected by the system and its software, as well as the relationship between the system and the environment in which it will operate. Thus, the actual software development process, if uncontrolled, may look like Figure 2.2; developers may

SIDEBAR 2.1 DRAWBACKS OF THE WATERFALL MODEL

Ever since the waterfall model was introduced, it has had many critics. For example, McCracken and Jackson (1981) pointed out that the model imposes a project management structure on system development. "To contend that any life cycle scheme, even with variations, can be applied to all system development is either to fly in the face of reality or to assume a life cycle so rudimentary as to be vacuous."

Notice that the waterfall model shows how each major phase of development terminates in the production of some artifact (such as requirements, design, or code). There is no insight into how each activity transforms one artifact to another, such as requirements to design. Thus, the model provides no guidance to managers and developers on how to handle changes to products and activities that are likely to occur during development. For instance, when requirements change during coding activities, the subsequent changes to design and code are not addressed by the waterfall model.

Curtis, Krasner, Shen, and Iscoe (1987) note that the waterfall model's major shortcoming is its failure to treat software as a problem-solving process. The waterfall model was derived from the hardware world, presenting a manufacturing view of software development. But manufacturing produces a particular item and reproduces it many times. Software is not developed like that; rather, it evolves as the problem becomes understood and the alternatives are evaluated. Thus, software is a creation process, not a manufacturing process. The waterfall model tells us nothing about the typical back-and-forth activities that lead to creating a final product. In particular, creation usually involves trying a little of this or that, developing and evaluating prototypes, assessing the feasibility of requirements, contrasting several designs, learning from failure, and eventually settling on a satisfactory solution to the problem at hand.

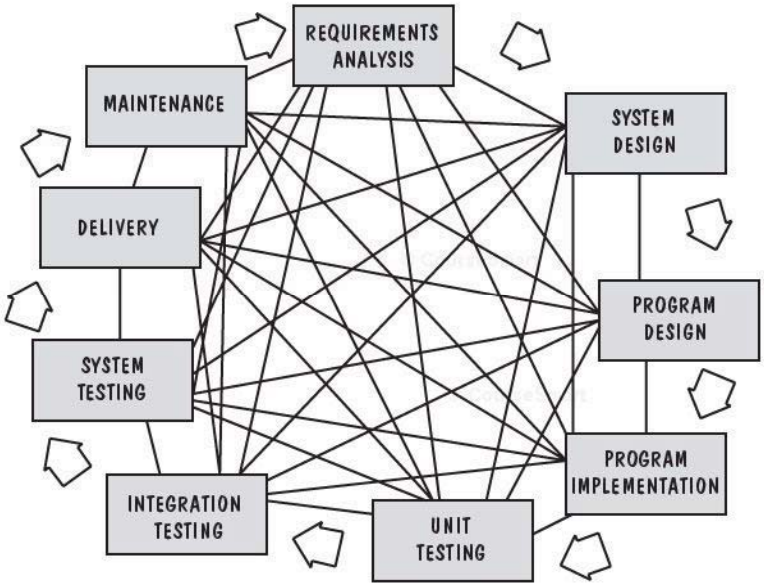


FIGURE 2.2 The software development process in reality.

thrash from one activity to the next and then back again, as they strive to gather knowledge about the problem and how the proposed solution addresses it.

The software development process can help to control the thrashing by including activities and subprocesses that enhance understanding. Prototyping is such a subprocess; a **prototype** is a partially developed product that enables customers and developers to examine some aspect of the proposed system and decide if it is suitable or appropriate for the finished product. For example, developers may build a system to implement a small portion of some key requirements to ensure that the requirements are consistent, feasible, and practical; if not, revisions are made at the requirements stage rather than at the more costly testing stage. Similarly, parts of the design may be prototyped, as shown in Figure 2.3. Design prototyping helps developers assess alternative design strategies and decide which is best for a particular project. As we will see in Chapter 5, the designers may address the requirements with several radically different designs to see which has the best properties. For instance, a network may be built as a ring in one prototype and a star in another, and performance characteristics evaluated to see which structure is better at meeting performance goals or constraints.

Often, the user interface is built and tested as a prototype, so that the users can understand what the new system will be like, and the designers get a better sense of how the users like to interact with the system. Thus, major kinks in the requirements are addressed and fixed well before the requirements are officially validated during system testing; **validation** ensures that the system has implemented all of the requirements, so that each system function can be traced back to a particular requirement in the specification. System testing also verifies the requirements; **verification** ensures that each function works correctly. That is, validation makes sure that the developer is building the

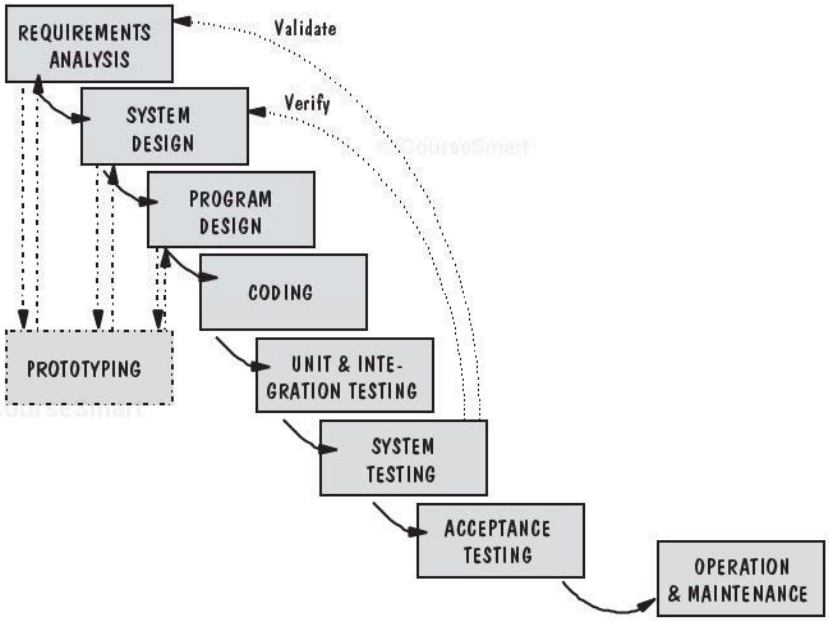


FIGURE 2.3 The waterfall model with prototyping.

right product (according to the specification), and verification checks the quality of the implementation. Prototyping is useful for verification and validation, but these activities can occur during other parts of the development process, as we will see in later chapters.

V Model

The **V model** is a variation of the waterfall model that demonstrates how the testing activities are related to analysis and design (German Ministry of Defense 1992). As shown in Figure 2.4, coding forms the point of the V, with analysis and design on the left, testing and maintenance on the right. Unit and integration testing addresses the correctness of programs, as we shall see in later chapters. The V model suggests that unit and integration testing can also be used to verify the program design. That is, during unit and integration testing, the coders and test team members should ensure that all aspects of the program design have been implemented correctly in the code. Similarly, system testing should verify the system design, making sure that all system design aspects are correctly implemented. Acceptance testing, which is conducted by the customer rather than the developer, validates the requirements by associating a testing step with each element of the specification; this type of testing checks to see that all requirements have been fully implemented before the system is accepted and paid for.

The model's linkage of the left side with the right side of the V implies that if problems are found during verification and validation, then the left side of the V can be reexecuted to fix and improve the requirements, design, and code before the testing steps on the right side are reenacted. In other words, the V model makes more explicit some of the iteration and rework that are hidden in the waterfall depiction. Whereas

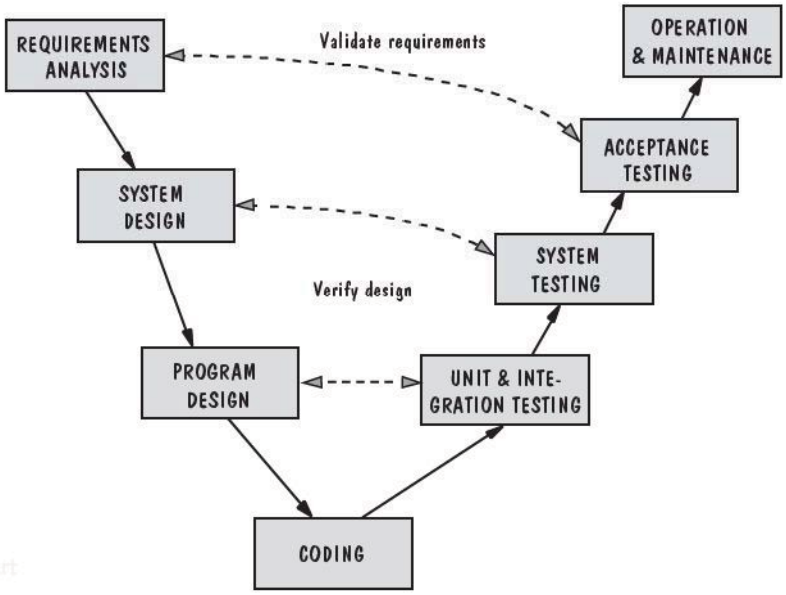


FIGURE 2.4 The V model.

the focus of the waterfall is often documents and artifacts, the focus of the V model is activity and correctness.

Prototyping Model

We have seen how the waterfall model can be amended with prototyping activities to improve understanding. But prototyping need not be solely an adjunct of a waterfall; it can itself be the basis for an effective process model, shown in Figure 2.5. Since the

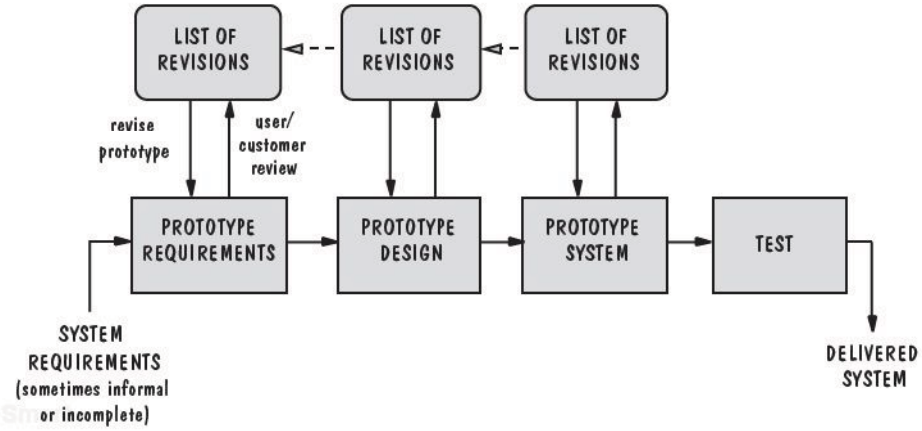


FIGURE 2.5 The prototyping model.

prototyping model allows all or part of a system to be constructed quickly to understand or clarify issues, it has the same objective as an engineering prototype, where requirements or design require repeated investigation to ensure that the developer, user, and customer have a common understanding both of what is needed and what is proposed. One or more of the loops for prototyping requirements, design, or the system may be eliminated, depending on the goals of the prototyping. However, the overall goal remains the same: reducing risk and uncertainty in development.

For example, system development may begin with a nominal set of requirements supplied by the customers and users. Then, alternatives are explored by having interested parties look at possible screens, tables, reports, and other system output that are used directly by the customers and users. As the users and customers decide on what they want, the requirements are revised. Once there is common agreement on what the requirements should be, the developers move on to design. Again, alternative designs are explored, often with consultation with customers and users.

The initial design is revised until the developers, users, and customers are happy with the result. Indeed, considering design alternatives sometimes reveals a problem with the requirements, and the developers drop back to the requirements activities to reconsider and change the requirements specification. Eventually, the system is coded and alternatives are discussed, with possible iteration through requirements and design again.

Operational Specification

For many systems, uncertainty about the requirements leads to changes and problems later in development. Zave (1984) suggests a process model that allows the developers and customers to examine the requirements and their implications early in the development process, where they can discuss and resolve some of the uncertainty. In the **operational specification model**, the system requirements are evaluated or executed in a way that demonstrates the behavior of the system. That is, once the requirements are specified, they can be enacted using a software package, so their implications can be assessed before design begins. For example, if the specification requires the proposed system to handle 24 users, an executable form of the specification can help analysts determine whether that number of users puts too much of a performance burden on the system.

This type of process is very different from traditional models such as the waterfall model. The waterfall model separates the functionality of the system from the design (i.e., what the system is to do is separated from how the system does it), intending to keep the customer needs apart from the implementation. However, an operational specification allows the functionality and the design to be merged. Figure 2.6 illustrates how an operational specification works. Notice that the operational specification is similar to prototyping; the process enables user and developer to examine requirements early on.

Transformational Model

Balzer's **transformational model** tries to reduce the opportunity for error by eliminating several major development steps. Using automated support, the transformational process applies a series of transformations to change a specification into a deliverable system (Balzer 1981a).

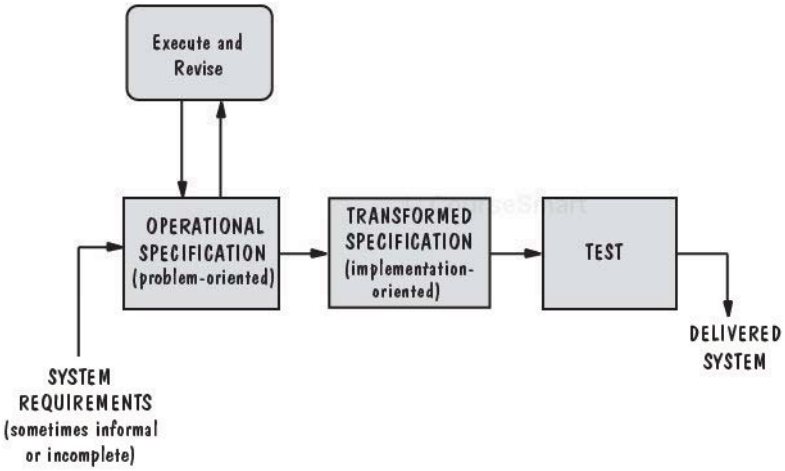


FIGURE 2.6 The operational specification model.

Sample transformations can include

- changing the data representations
- selecting algorithms
- optimizing
- compiling

Because many paths can be taken from the specification to the delivered system, the sequence of transformations and the decisions they reflect are kept as a formal development record.

The transformational approach holds great promise. However, a major impediment to its use is the need for a formal specification expressed precisely so the transformations can operate on it, as shown in Figure 2.7. As formal specification methods become more popular, the transformational model may gain wider acceptance.

Phased Development: Increments and Iterations

In the early years of software development, customers were willing to wait a long time for software systems to be ready. Sometimes years would pass between the time the requirements documents were written and the time the system was delivered, called the **cycle time**. However, today's business environment no longer tolerates long delays. Software helps to distinguish products in the marketplace, and customers are always looking for new quality and functionality. For example, in 1996, 80 percent of Hewlett-Packard's revenues were derived from products introduced in the previous two years. Consequently, new process models were developed to help reduce cycle time.

One way to reduce cycle time is to use phased development, as shown in Figure 2.8. The system is designed so that it can be delivered in pieces, enabling the users to have some functionality while the rest is being developed. Thus, there are usually two systems functioning in parallel: the production system and the development system. The

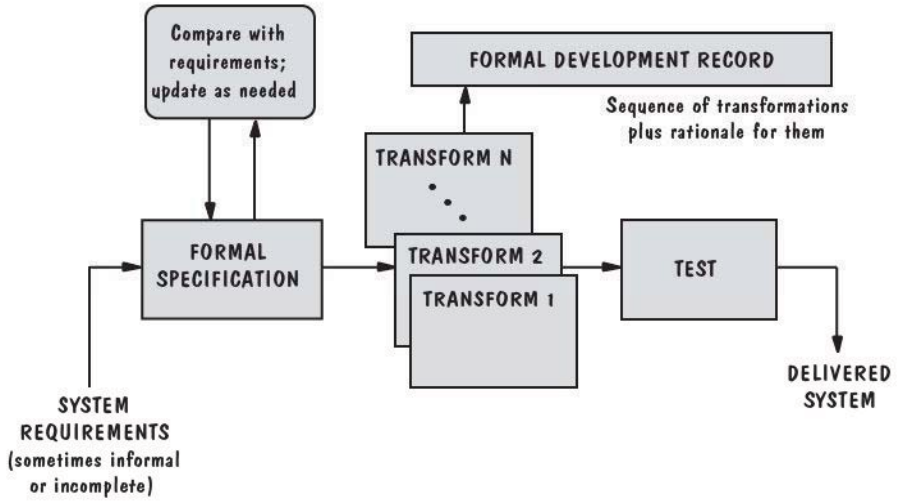


FIGURE 2.7 The transformational model.

operational or **production system** is the one currently being used by the customer and user; the **development system** is the next version that is being prepared to replace the current production system. Often, we refer to the systems in terms of their release numbers: the developers build Release 1, test it, and turn it over to the users as the first operational release. Then, as the users use Release 1, the developers are building Release 2. Thus, the developers are always working on Release $n + 1$ while Release n is operational.

There are many ways for the developers to decide how to organize development into releases. The two most popular approaches are incremental development and iterative development. In **incremental development**, the system as specified in the requirements documents is partitioned into subsystems by functionality. The releases are defined by beginning with one small, functional subsystem and then adding functionality

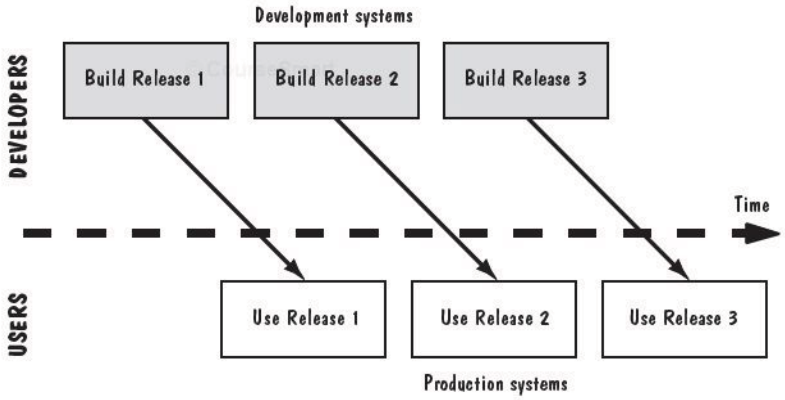


FIGURE 2.8 The phased-development model.

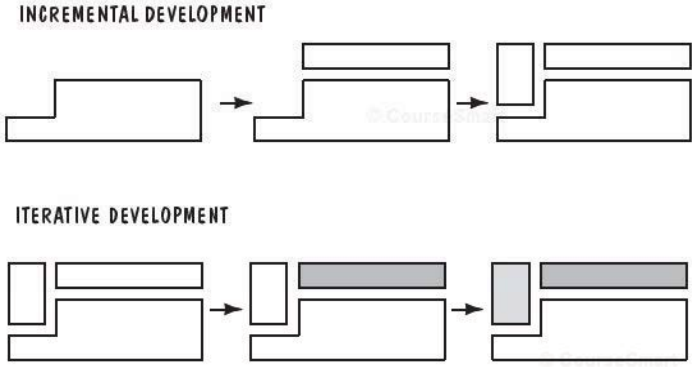


FIGURE 2.9 The incremental and iterative models.

with each new release. The top of Figure 2.9 shows how incremental development slowly builds up to full functionality with each new release.

However, **iterative development** delivers a full system at the very beginning and then changes the functionality of each subsystem with each new release. The bottom of Figure 2.9 illustrates three releases in an iterative development.

To understand the difference between incremental and iterative development, consider a word processing package. Suppose the package is to deliver three types of functionality: creating text, organizing text (i.e., cutting and pasting), and formatting text (such as using different type sizes and styles). To build such a system using incremental development, we might provide only the creation functions in Release 1, then both creation and organization in Release 2, and finally creation, organization, and formatting in Release 3. However, using iterative development, we would provide primitive forms of all three types of functionality in Release 1. For example, we can create text and then cut and paste it, but the cutting and pasting functions might be clumsy or slow. So in the next iteration, Release 2, we have the same functionality, but have enhanced the quality; now cutting and pasting are easy and quick. Each release improves on the previous ones in some way.

In reality, many organizations use a combination of iterative and incremental development. A new release may include new functionality, but existing functionality from the current release may have been enhanced. These forms of phased development are desirable for several reasons:

1. Training can begin on an early release, even if some functions are missing. The training process allows developers to observe how certain functions are executed, suggesting enhancements for later releases. In this way, the developers can be very responsive to the users.
2. Markets can be created early for functionality that has never before been offered.
3. Frequent releases allow developers to fix unanticipated problems globally and quickly, as they are reported from the operational system.
4. The development team can focus on different areas of expertise with different releases. For instance, one release can change the system from a command-driven

one to a point-and-click interface, using the expertise of the user-interface specialists; another release can focus on improving system performance.

Spiral Model

Boehm (1988) viewed the software development process in light of the risks involved, suggesting that a spiral model could combine development activities with risk management to minimize and control risk. The spiral model, shown in Figure 2.10, is in some sense like the iterative development shown in Figure 2.9. Beginning with the requirements and an initial plan for development (including a budget, constraints, and alternatives for staffing, design, and development environment), the process inserts a step to evaluate risks and prototype alternatives before a "concept of operations" document is produced to describe at a high level how the system should work. From that document, a set of requirements is specified and scrutinized to ensure that the requirements are as complete and consistent as possible. Thus, the concept of operations is the product of the first iteration, and the requirements are the principal product of the second. In the third iteration, system development produces the design, and the fourth enables testing.

With each iteration, the risk analysis weighs different alternatives in light of the requirements and constraints, and prototyping verifies feasibility or desirability before a particular alternative is chosen. When risks are identified, the project managers must decide how to eliminate or minimize the risk. For example, designers may not be sure whether users will prefer one type of interface over another. To minimize the risk of choosing an interface that will prevent productive use of the new system, the designers can prototype each interface and run tests to see which is preferred, or even choose to include

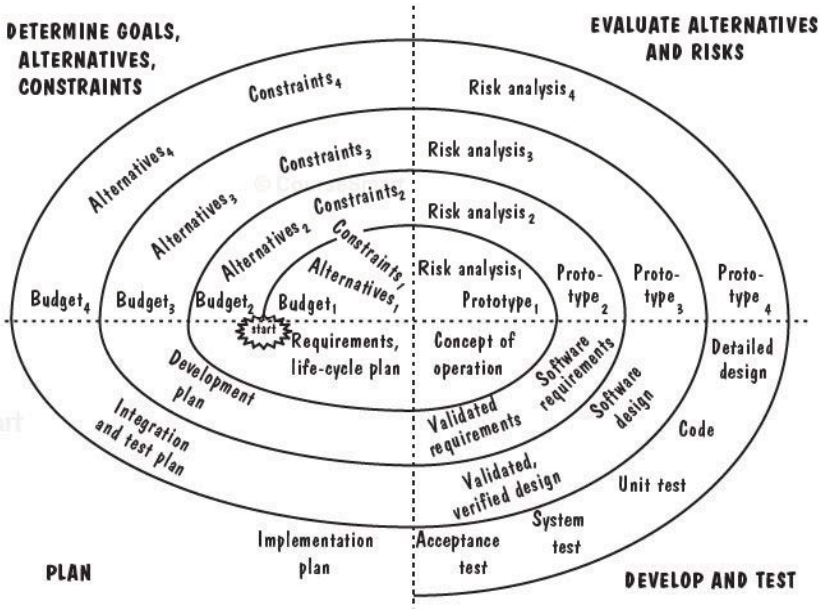


FIGURE 2.10 The spiral model.

two different interfaces in the design, so the users can select an interface when they log on. Constraints such as budget and schedule help to determine which risk-management strategy is chosen. We will discuss risk management in more detail in Chapter 3.

Agile Methods

Many of the software development processes proposed and used from the 1970s through the 1990s tried to impose some form of rigor on the way in which software is conceived, documented, developed, and tested. In the late 1990s, some developers who had resisted this rigor formulated their own principles, trying to highlight the roles that flexibility could play in producing software quickly and capably. They codified their thinking in an "agile manifesto" that focuses on four tenets of an alternative way of thinking about software development (Agile Alliance 2001):

- They value individuals and interactions over processes and tools. This philosophy includes supplying developers with the resources they need and then trusting them to do their jobs well. Teams organize themselves and communicate through face-to-face interaction rather than through documentation.
- They prefer to invest time in producing working software rather than in producing comprehensive documentation. That is, the primary measure of success is the degree to which the software works properly.
- They focus on customer collaboration rather than contract negotiation, thereby involving the customer in key aspects of the development process.
- They concentrate on responding to change rather than on creating a plan and then following it, because they believe that it is impossible to anticipate all requirements at the beginning of development.

The overall goal of agile development is to satisfy the customer by "early and continuous delivery of valuable software" (Agile Alliance 2001). Many customers have business needs that change over time, reflecting not only newly discovered needs but also the need to respond to changes in the marketplace. For example, as software is being designed and constructed, a competitor may release a new product that requires a change in the software's planned functionality. Similarly, a government agency or standards body may impose a regulation or standard that affects the software's design or requirements. It is thought that by building flexibility into the development process, agile methods can enable customers to add or change requirements late in the development cycle.

There are many examples of agile processes in the current literature. Each is based on a set of principles that implement the tenets of the agile manifesto. Examples include the following.

- **Extreme programming (XP),** described in detail below, is a set of techniques for leveraging the creativity of developers and minimizing the amount of administrative overhead.
- **Crystal** is a collection of approaches based on the notion that every project needs a different set of policies, conventions, and methodologies. Cockburn (2002), the creator of Crystal, believes that people have a major influence on software quality, and thus the quality of projects and processes improves as the quality of the

people involved improves. Productivity increases through better communication and frequent delivery, because there is less need for intermediate work products.

- **Scrum** was created at Object Technology in 1994 and was subsequently commercialized by Schwaber and Beedle (2002). It uses iterative development, where each 30-day iteration is called a "sprint," to implement the product's backlog of prioritized requirements. Multiple self-organizing and autonomous teams implement product increments in parallel. Coordination is done at a brief daily status meeting called a "scrum" (as in rugby).
- **Adaptive software development (ASD)** has six basic principles. There is a mission that acts as a guideline, setting out the destination but not prescribing how to get there. Features are viewed as the crux of customer value, so the project is organized around building components to provide the features. Iteration is important, so redoing is as critical is doing; change is embraced, so that a change is viewed not as a correction but as an adjustment to the realities of software development. Fixed delivery times force developers to scope down the requirements essential for each version produced. At the same time, risk is embraced, so that the developers tackle the hardest problems first.

Often, the phrase "extreme programming" is used to describe the more general concept of agile methods. In fact, XP is a particular form of agile process, with guiding principles that reflect the more general tenets of the agile manifesto. Proponents of XP emphasize four characteristics of agility: communication, simplicity, courage, and feedback. *Communication* involves the continual interchange between customers and developers. *Simplicity* encourages developers to select the simplest design or implementation to address the needs of their customers. *Courage* is described by XP creators as commitment to delivering functionality early and often. *Feedback* loops are built into the various activities during the development process. For example, programmers work together to give each other feedback on the best way to implement a design, and customers work with developers to perform planning exercises.

These characteristics are embedded in what are known as the twelve facets of XP.

- *The planning game:* In this aspect of XP, the customer, who is on-site, defines what is meant by "value," so that each requirement can be assessed according to how much value is added by implementing it. The users write stories about how the system should work, and the developers then estimate the resources necessary to realize the stories. The stories describe the actors and actions involved, much like the use cases we define in more detail in Chapters 4 and 6. Each story relates one requirement; two or three sentences are all that is needed to explain the value of the requirement in sufficient detail for the developer to specify test cases and estimate resources for implementing the requirement. Once the stories are written, the prospective users prioritize requirements, splitting and merging them until consensus is reached on what is needed, what is testable, and what can be done with the resources available. The planners then generate a map of each release, documenting what the release includes and when it will be delivered.
- *Small releases:* The system is designed so that functionality can be delivered as soon as possible. Functions are decomposed into small parts, so that some functionality

can be delivered early and then improved or expanded on in later releases. The small releases require a phased-development approach, with incremental or iterative cycles.

- *Metaphor:* The development team agrees on a common vision of how the system will operate. To support its vision, the team chooses common names and agrees on a common way of addressing key issues.
- *Simple design:* Design is kept simple by addressing only current needs. This approach reflects the philosophy that anticipating future needs can lead to unnecessary functionality. If a particular portion of a system is very complex, the team may build a spike—a quick and narrow implementation—to help it decide how to proceed.
- *Writing tests first:* To ensure that the customer's needs are the driving force behind development, test cases are written first, as a way of forcing customers to specify requirements that can be tested and verified once the software is built. Two kinds of tests are used in XP: functional tests that are specified by the customer and executed by both developers and users, and unit tests that are written and run by developers. In XP, functional tests are automated and, ideally, run daily. The functional tests are considered to be part of the system specification. Unit tests are written both before and after coding, to verify that each modular portion of the implementation works as designed. Both functional and unit testing are described in more detail in Chapter 8.
- *Refactoring:* As the system is built, it is likely that requirements will change. Because a major characteristic of XP philosophy is to design only to current requirements, it is often the case that new requirements force the developers to reconsider their existing design. **Refactoring** refers to revisiting the requirements and design, reformulating them to match new and existing needs. Sometimes refactoring addresses ways to restructure design and code without perturbing the system's external behavior. The refactoring is done in small steps, supported by unit tests and pair programming, with simplicity guiding the effort. We will discuss the difficulties of refactoring in Chapter 5.
- *Pair programming:* As noted in Chapter 1, there is a tension between viewing software engineering as an art and as a science. Pair programming attempts to address the artistic side of software development, acknowledging that the apprentice–master metaphor can be useful in teaching novice software developers how to develop the instincts of masters. Using one keyboard, two paired programmers develop a system from the specifications and design. One person has responsibility for finishing the code, but the pairing is flexible: a developer may have more than one partner on a given day. We will see in Chapter 7 how pair programming compares with the more traditional approach of individuals working separately until their modules have been unit-tested.
- *Collective ownership:* In XP, any developer can make a change to any part of the system as it is being developed. In Chapter 11, we will address the difficulties in managing change, including the errors introduced when two people try to change the same module simultaneously.
- *Continuous integration:* Delivering functionality quickly means that working systems can be promised to the customer daily and sometimes even hourly. The

emphasis is on small increments or improvements rather than on grand leaps from one revision to the next.

- *Sustainable pace:* XP's emphasis on people includes acknowledging that fatigue can produce errors. So proponents of XP suggest a goal of 40 hours for each work week; pushing developers to devote heroic amounts of time to meeting deadlines is a signal that the deadlines are unreasonable or that there are insufficient resources for meeting them.
- *On-site customer:* Ideally, a customer should be present on-site, working with the developers to determine requirements and providing feedback about how to test them.
- *Coding standards:* Many observers think of XP and other agile methods as providing an unconstrained environment where anything goes. But in fact XP advocates clear definition of coding standards, to encourage teams to be able to understand and change each other's work. These standards support other practices, such as testing and refactoring. The result should be a body of code that appears to have been written by one person, and is consistent in its approach and expression.

Extreme programming and agile methods are relatively new. The body of evidence for its effectiveness is small but growing. We will revisit many agile methods and concepts, and their empirical evaluation, in later chapters, as we discuss their related activities.

The process models presented in this chapter are only a few of those that are used or discussed. Other process models can be defined and tailored to the needs of the user, customer, and developer. As Sidebar 2.3 notes, we should really capture the development process as a collection of process models, rather than focusing on a single model or view.

SIDEBAR 2.2 WHEN IS EXTREME TOO EXTREME?

As with most software development approaches, agile methods are not without their critics. For example, Stephens and Rosenberg (2003) point out that many of extreme programming's practices are interdependent, a vulnerability if one of them is modified. To see why, suppose some people are uncomfortable with pair programming. More coordination and documentation may be required to address the shared vision that is missing when people work on their own. Similarly, many developers prefer to do some design before they write code. Scrum addresses this preference by organizing around monthly sprints. Elssamadissy and Schalliol (2002) note that, in extreme programming, requirements are expressed as a set of test cases that must be passed by the software. This approach may cause customer representatives to focus on the test cases instead of the requirements. Because the test cases are a detailed expression of the requirements and may be solution oriented, the emphasis on test cases can distract the representatives from the project's underlying goals and can lead to a situation where the system passes all the tests but is not what the customers thought they were paying for. As we will see in Chapter 5, refactoring may be the Achilles heel of agile methods; it is difficult to rework a software system without degrading its architecture.

SIDEBAR 2.3 COLLECTIONS OF PROCESS MODELS

We saw in Sidebar 2.1 that the development process is a problem-solving activity, but few of the popular process models include problem solving. Curtis, Krasner, and Iscoe (1988) performed a field study of 17 large projects, to determine which problem-solving factors should be captured in process models to aid our understanding of software development. In particular, they looked at the behavioral and organizational factors that affect project outcomes. Their results suggest a layered behavioral model of software development, including five key perspectives: the business milieu, the company, the project, the team, and the individual. The individual view provides information about cognition and motivation, and project and team views tell us about group dynamics. The company and business milieu provide information about organizational behavior that can affect both productivity and quality. This model does not replace traditional process models; rather, it is orthogonal, supplementing the traditional models with information on how behavior affects the creation and production activities.

As the developers and customers learn about the problem, they integrate their knowledge of domains, technology, and business to produce an appropriate solution. By viewing development as a collection of coordinating processes, we can see the effects of learning, technical communication, customer interaction, and requirements negotiation. Current models that prescribe a series of development tasks "provide no help in analyzing how much new information must be learned by the project staff, how discrepant requirements should be negotiated, how a design team can resolve architectural conflicts, and how these and similar factors contribute to a project's inherent uncertainty and risk" (Curtis, Krasner, and Iscoe 1988). However, when we include models of cognitive, social, and organizational processes, we begin to see the causes of bottlenecks and inefficiency. It is this insight that enables managers to understand and control the development process. And by aggregating behavior across layers of models, we can see how each model contributes to or compounds the effects of another model's factors.

No matter what process model is used, many activities are common to all. As we investigate software engineering in later chapters, we will examine each development activity to see what it involves and to find out what tools and techniques make us more effective and productive.

2.3 TOOLS AND TECHNIQUES FOR PROCESS MODELING

There are many choices for modeling tools and techniques, once you decide what you want to capture in your process model; we have seen several modeling approaches in our model depictions in the preceding section. The appropriate technique for you depends on your goals and your preferred work style. In particular, your choice for notation depends on what you want to capture in your model. The notations range from textual ones that express processes as functions, to graphical ones that depict processes

as hierarchies of boxes and arrows, to combinations of pictures and text that link the graphical depiction to tables and functions elaborating on the high-level illustration. Many of the modeling notations can also be used for representing requirements and designs; we examine some of them in later chapters.

In this chapter, the notation is secondary to the type of model, and we focus on two major categories, static and dynamic. A **static model** depicts the process, showing that the inputs are transformed to outputs. A **dynamic model** enacts the process, so the user can see how intermediate and final products are transformed over time.

Static Modeling: Lai Notation

There are many ways to model a process statically. In the early 1990s, Lai (1991) developed a comprehensive process notation that is intended to enable someone to model any process at any level of detail. It builds on a paradigm where people perform roles while resources perform activities, leading to the production of artifacts. The process model shows the relationships among the roles, activities, and artifacts, and state tables show information about the completeness of each artifact at a given time.

In particular, the elements of a process are viewed in terms of seven types:

1. **Activity:** Something that will happen in a process. This element can be related to what happens before and after, what resources are needed, what triggers the activity's start, what rules govern the activity, how to describe the algorithms and lessons learned, and how to relate the activity to the project team.
2. **Sequence:** The order of activities. The sequence can be described using triggers, programming constructs, transformations, ordering, or satisfaction of conditions.
3. **Process model:** A view of interest about the system. Thus, parts of the process may be represented as a separate model, either to predict process behavior or to examine certain characteristics.
4. **Resource:** A necessary item, tool, or person. Resources can include equipment, time, office space, people, techniques, and so on. The process model identifies how much of each resource is needed for each activity.
5. **Control:** An external influence over process enactment. The controls may be manual or automatic, human or mechanical.
6. **Policy:** A guiding principle. This high-level process constraint influences process enactment. It may include a prescribed development process, a tool that must be used, or a mandatory management style.
7. **Organization:** The hierarchical structure of process agents, with physical grouping corresponding to logical grouping and related roles. The mapping from physical to logical grouping should be flexible enough to reflect changes in physical environment.

The process description itself has several levels of abstraction, including the software development process that directs certain resources to be used in constructing specific modules, as well as generic models that may resemble the spiral or waterfall models. Lai's notation includes several templates, such as an Artifact Definition Template, which records information about particular artifacts.

Lai's approach can be applied to modeling software development processes; later in this chapter, we use it to model the risk involved in development. However, to demonstrate its use and its ability to capture many facets of a complex activity, we apply it to a relatively simple but familiar process, driving an automobile. Table 2.1 contains a description of the key resource in this process, a car.

Other templates define relations, process states, operations, analysis, actions, and roles. Graphical diagrams represent the relationships between elements, capturing the main relationships and secondary ones. For example, Figure 2.11 illustrates the process of starting a car. The "initiate" box represents the entrance conditions, and the "park" box represents an exit condition. The left-hand column of a condition box lists artifacts, and the right-hand column is the artifact state.

TABLE 2.1 Artifact Definition Form for Artifact "CAR" (Lai 1991)

Name	*Car*	
Synopsis	*This is the artifact that represents a class of cars.*	
Complexity type	*Composite*	
Data type	*(car c, user-defined)*	
Artifact-state list		
parked	((*state_of*(*car.engine*) = *off*) (*state_of*(*car.gear*) = *park*) (*state_of*(*car.speed*) = *stand*))	*Car is not moving, and engine is not running.*
initiated	((*state_of*(*car.engine*) = *on*) (*state_of*(*car.key_hole*) = *has-key*) (*state_of*(*car-driver*(*car.*)) = *in-car*) *state_of*(*car.gear*) = *drive*) (*state_of*(*car.speed*) = *stand*))	*Car is not moving, but the engine is running.*
moving	((*state_of*(*car.engine*) = *on*) (*state_of*(*car.keyhole*) = *has-key*) (*state_of*(*car-driver*(*car.*)) = *driving*) ((*state_of*(*car.gear*) = *drive*) *or* (*state_of*(*car.gear*) = *reverse*)) ((*state_of*(*car.speed*) = *stand*) *or* (*state_of*(*car.speed*) = *slow*) *or* (*state_of*(*car.speed*) = *medium*) *or* (*state_of*(*car.speed*) = *high*))	*Car is moving forward or backward.*
Subartifact list		
	doors	*The four doors of a car*
	engine	*The engine of a car*
	keyhole	*The ignition keyhole of a car*
	gear	*The gear of a car*
	speed	*The speed of a car*
Relations list		
car-key	*This is the relation between a car and a key.*	
car-driver	*This is the relation between a car and a driver.*	

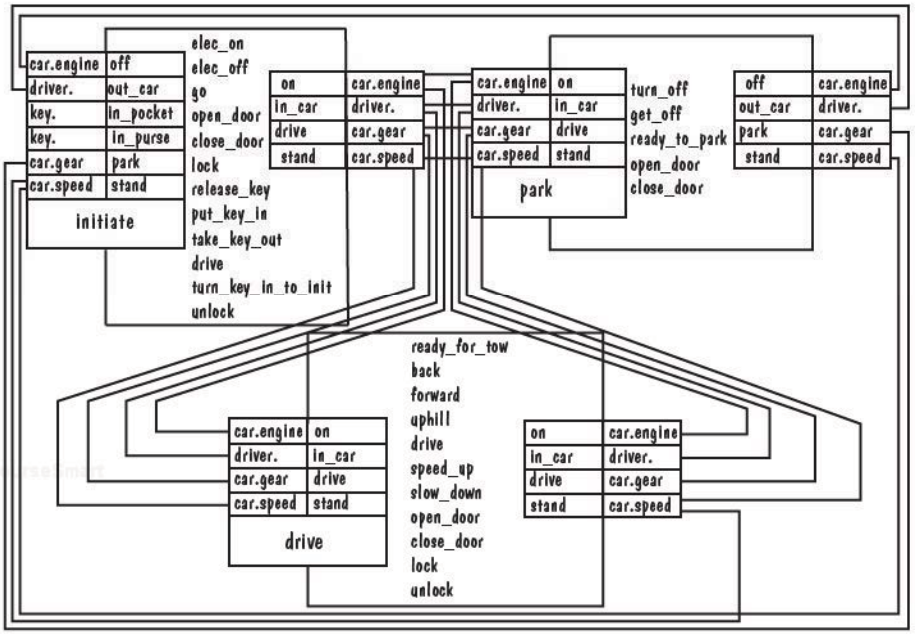
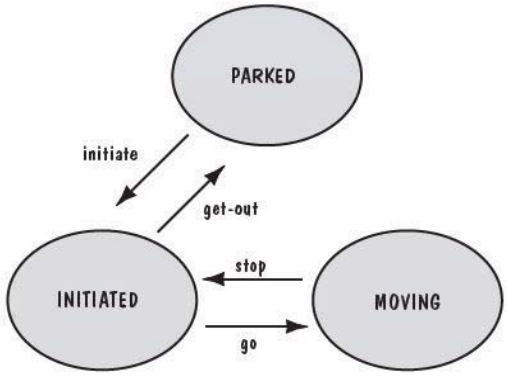


FIGURE 2.11 The process of starting a car (Lai 1991).

Transition diagrams supplement the process model by showing how the states are related to one another. For example, Figure 2.12 illustrates the transitions for a car.

Lai's notation is a good example of how multiple structures and strategies can be used to capture a great deal of information about the software development process. But it is also useful in organizing and depicting process information about user requirements, as the car example demonstrates.

FIGURE 2.12 Transition diagram for a car (Lai 1991).



Dynamic Modeling: System Dynamics

A desirable property of a process model is the ability to enact the process, so that we can watch what happens to resources and artifacts as activities occur. In other words, we want to describe a model of the process and then watch as software shows us how resources flow through activities to become outputs. This dynamic process view enables us to simulate the process and make changes before the resources are actually expended. For example, we can use a dynamic process model to help us decide how many testers we need or when we must initiate testing in order to finish on schedule. Similarly, we can include or exclude activities to see their effects on effort and schedule. For instance, we can add a code-review activity, making assumptions about how many faults we will find during the review, and determine whether reviewing shortens test time significantly.

There are several ways to build dynamic process models. The systems dynamics approach, introduced by Forrester in the 1950s, has been useful for simulating diverse processes, including ecological, economic, and political systems (Forrester 1991). Abdel-Hamid and Madnick have applied system dynamics to software development, enabling project managers to "test out" their process choices before imposing them on developers (Abdel-Hamid 1989; Abdel-Hamid and Madnick 1991).

To see how system dynamics works, consider how the software development process affects productivity. We can build a descriptive model of the various activities that involve developers' time and then look at how changes in the model increase or decrease the time it takes to design, write, and test the code. First, we must determine which factors affect overall productivity. Figure 2.13 depicts Abdel-Hamid's understanding of these

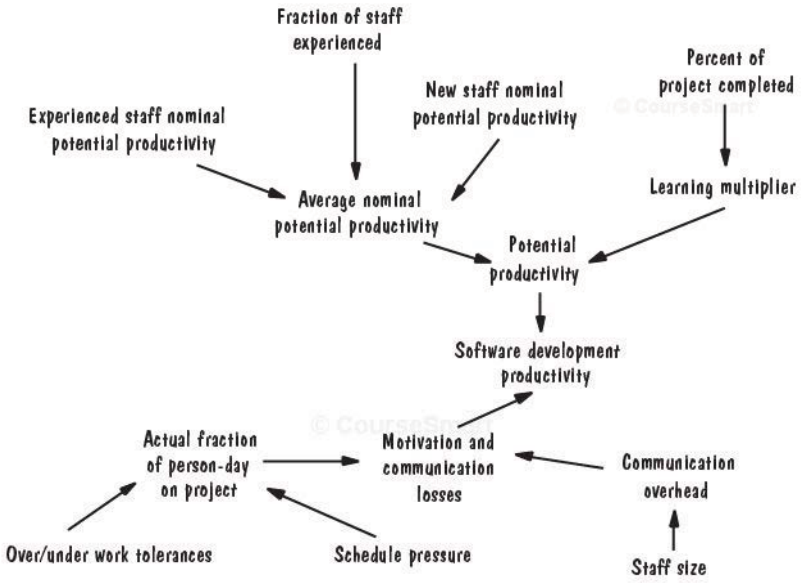


FIGURE 2.13 Model of factors contributing to productivity (Abdel-Hamid 1996).

factors. The arrows indicate how changes in one factor affect changes in another. For example, if the fraction of experienced staff increases from one-quarter to one-half of the people assigned to the project, then we would expect the average potential productivity to increase, too. Similarly, the larger the staff (reflected in staff size), the more time is devoted to communication among project members (communication overhead).

The figure shows us that average nominal potential productivity is affected by three things: the productivity of the experienced staff, the fraction of experienced staff, and the productivity of the new staff. At the same time, new staff must learn about the project; as more of the project is completed, the more the new staff must learn before they can become productive members of the team.

Other issues affect the overall development productivity. First, we must consider the fraction of each day that each developer can devote to the project. Schedule pressures affect this fraction, as do the developers' tolerances for workload. Staff size affects productivity, too, but the more staff, the more likely it is that time will be needed just to communicate information among team members. Communication and motivation, combined with the potential productivity represented in the upper half of Figure 2.13, suggest a general software development productivity relationship.

Thus, the first step in using system dynamics is to identify these relationships, based on a combination of empirical evidence, research reports, and intuition. The next step is to quantify the relationships. The quantification can involve direct relationships, such as that between staff size and communication. We know that if n people are assigned to a project, then there are $n(n-1)/2$ potential pairs of people who must communicate and coordinate with one another. For some relationships, especially those that involve resources that change over time, we must assign distributions that describe the building up and diminishing of the resource. For example, it is rare for everyone on a project to begin work on the first day. The systems analysts begin, and coders join the project once the significant requirements and design components are documented. Thus, the distribution describes the rise and fall (or even the fluctuation, such as availability around holidays or summer vacations) of the resources.

A system dynamics model can be extensive and complex. For example, Abdel-Hamid's software development model contains more than 100 causal links; Figure 2.14 shows an overview of the relationships he defined. He defined four major areas that affect productivity: software production, human resource management, planning, and control. Production includes issues of quality assurance, learning, and development rate. Human resources address hiring, turnover, and experience. Planning concerns schedules and the pressures they cause, and control addresses progress measurement and the effort required to finish the project.

Because the number of links can be quite large, system dynamics models are supported by software that captures both the links and their quantitative descriptions and then simulates the overall process or some subprocess.

The power of system dynamics is impressive, but this method should be used with caution. The simulated results depend on the quantified relationships, which are often heuristic or vague, not clearly based on empirical research. However, as we will see in later chapters, a historical database of measurement information about the various aspects of development can help us gain confidence in our understanding of relationships, and thus in the results of dynamic models.

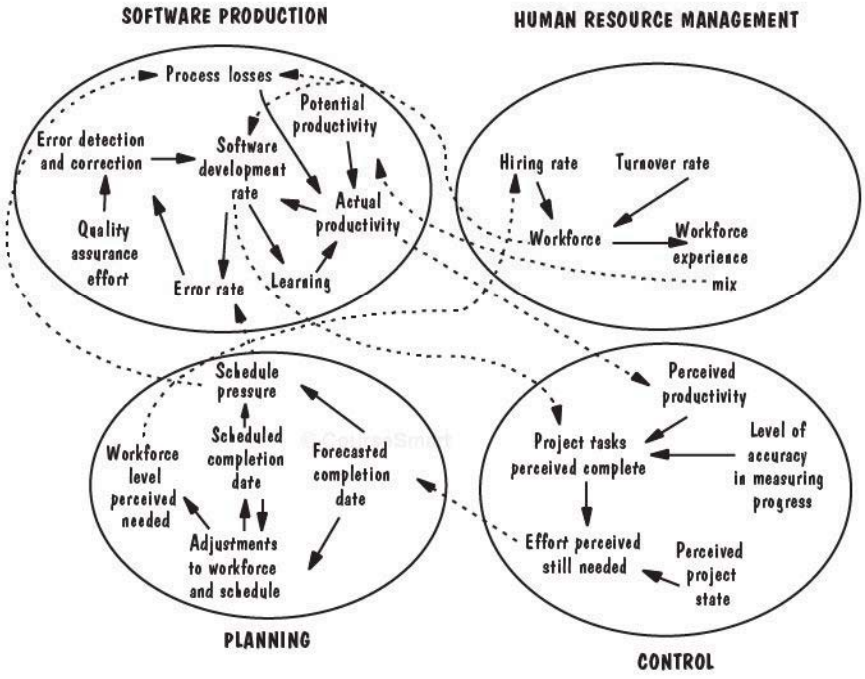


FIGURE 2.14 Structure of software development (Abdel-Hamid 1996).

SIDEBAR 2.4 PROCESS PROGRAMMING

In the mid-1980s, Osterweil (1987) proposed that software engineering processes be specified using algorithmic descriptions. That is, if a process is well-understood, we should be able to write a program to describe the process, and then run the program to enact the process. The goal of process programming is to eliminate uncertainty, both by having enough understanding to write software to capture its essence, and by turning the process into a deterministic solution to the problem.

Were process programming possible, we could have management visibility into all process activities, automate all activities, and coordinate and change all activities with ease. Thus, process programs could form the basis of an automated environment to produce software.

However, Curtis, Krasner, Shen, and Iscoe (1987) point out that Osterweil's analogy to computer programming does not capture the inherent variability of the underlying development process. When a computer program is written, the programmer assumes that the implementation environment works properly; the operating system, database manager, and hardware are reliable and correct, so there is little variability in the computer's response to an instruction. But when a process program issues an instruction to a member of the project team, there is great variability in the way the task is executed and in the results produced. As

we will see in Chapter 3, differences in skill, experience, work habits, understanding the customer's needs, and a host of other factors can increase variability dramatically. Curtis and his colleagues suggest that process programming be restricted only to those situations with minimal variability. Moreover, they point out that Osterweil's examples provide information only about the sequencing of tasks; the process program does not help to warn managers of impending problems. "The coordination of a web of creative intellectual tasks does not appear to be improved greatly by current implementations of process programming, because the most important source of coordination is to ensure that all of the interacting agents share the same mental model of how the system should operate" (Curtis et al. 1987).

2.4 PRACTICAL PROCESS MODELING

Process modeling has long been a focus of software engineering research. But how practical is it? Several researchers report that, used properly, process modeling offers great benefits for understanding processes and revealing inconsistencies. For example, Barghouti, Rosenblum, Belanger, and Alliegro (1995) conducted two case studies to determine the feasibility, utility, and limitations of using process models in large organizations. In this section, we examine what they did and what they found.

Marvel Case Studies

In both studies, the researchers used MSL, the Marvel Specification Language, to define the process, and then generated a Marvel process enactment environment for it (Kaiser, Feiler, and Popovich 1988; Barghouti and Kaiser 1991). MSL uses three main constructs—classes, rules, and tool envelopes—to produce a three-part process description:

1. a rule-based specification of process behavior
2. an object-oriented definition of the model's information process
3. a set of envelopes to interface between Marvel and external software tools used to execute the process.

The first case study involved an AT&T call-processing network that carried phone calls, and a separate signaling network responsible for routing the calls and balancing the network's load. Marvel was used to describe the Signaling Fault Resolution process that is responsible for detecting, servicing, and resolving problems with the signaling network. Workcenter 1 monitored the network, detected faults, and referred the fault to one of the two other workcenters. Workcenter 2 handled software or human faults that required detailed analysis, and Workcenter 3 dealt with hardware failures. Figure 2.15 depicts this process. Double dashed lines indicate which activity uses the tool or database represented by an oval. A rectangle is a task or activity, and a diamond is a decision. Arrows indicate the flow of control. As you can see, the figure provides an overview but is not detailed enough to capture essential process elements.

Consequently, each of the entities and workcenters is modeled using MSL. Figure 2.16 illustrates how that is done. The upper half of the figure defines the class

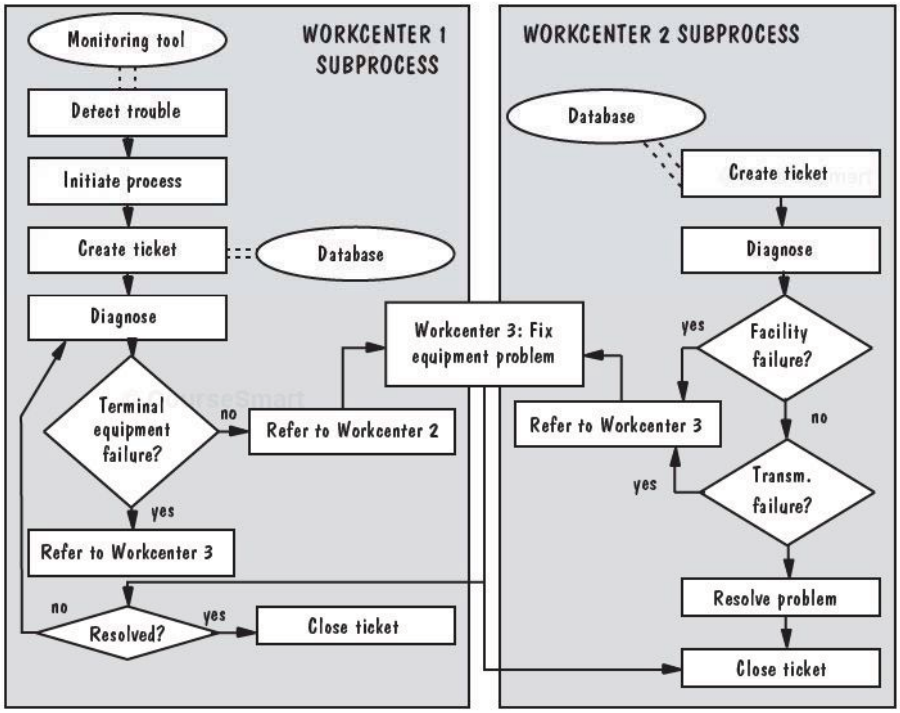


FIGURE 2.15 Signaling Fault Resolution process (Barghouti et al. 1995).

```
TICKET:: superclass ENTITY
  status: (initial, open, referred_out, referral_done,
                    closed, fixed) = initial;
  diagnostics     : (terminal, non_terminal, none) = none;
  level           : integer;
  description     : text;
  referred_to     : link WORKCENTER;
  referrals       : set_of link TICKET;
  process         : link PROC_INST;
end
```

Class definition for trouble tickets

```
diagnose [?t: TICKET]:
  (exists PROC_INST ?p suchthat (linkto [?t.process ?p]))
  :
  (and (?t.status = open}(?t.diagnostics = none))
  {TICKET_UTIL diagnose ?t.Name}
  (and (?t.diagnostics = terminal)
          (?p.last_task = diagnose)
          (?p.next_task = refer_to_WC3));
  (and (?t.diagnostics = non_terminal)
          (?p.last_task = diagnose)
          (?p.next_task = refer_to_WC2));
```

Rule for diagnosing ticket

FIGURE 2.16 Examples of Marvel commands (Barghouti et al. 1995).

TICKET, where a ticket represents the trouble ticket (or problem report) written whenever a failure occurs. As we will see in the chapters on testing, trouble tickets are used to track a problem from its occurrence to its resolution. The entire network was represented with 22 such MSL classes; all information created or required by a process was included.

Next, the model addressed behavioral aspects of the Signaling Fault Resolution process. The lower half of Figure 2.16 is an MSL rule that corresponds loosely to the box of Figure 2.15 labeled "Diagnose." Thus, the MSL describes the rule for diagnosing open problems; it is fired for each open ticket. When the process model was done, there were 21 MSL rules needed to describe the system.

The second case study addressed part of the software maintenance process for AT&T's 5ESS switching software. Unlike the first case study, where the goal was process improvement, the second study aimed only to document the process steps and interactions by capturing them in MSL. The model contained 25 classes and 26 rules.

For each model, the MSL process descriptions were used to generate "process enactment environments," resulting in a database populated with instances of the information model's classes. Then, researchers simulated several scenarios to verify that the models performed as expected. During the simulation, they collected timing and resource utilization data, providing the basis for analyzing likely process performance. By changing the rules and executing a scenario repeatedly, the timings were compared and contrasted, leading to significant process improvement without major investment in resources.

The modeling and simulation exercises were useful for early problem identification and resolution. For example, the software maintenance process definition uncovered three types of problems with the existing process documentation: missing task inputs and outputs, ambiguous input and output criteria, and inefficiency in the process definition. The signaling fault model simulation discovered inefficiencies in the separate descriptions of the workcenters.

Barghouti and his colleagues note the importance of dividing the process modeling problem into two pieces: modeling the information and modeling the behavior. By separating these concerns, the resulting model is clear and concise. They also point out that computer-intensive activities are more easily modeled than human-intensive ones, a lesson noted by Curtis and his colleagues, too.

Desirable Properties of Process Modeling Tools and Techniques

There are many process modeling tools and techniques, and researchers continue to work to determine which ones are most appropriate for a given situation. But there are some characteristics that are helpful, regardless of technique. Curtis, Kellner, and Over (1992) have identified five categories of desirable properties:

1. *Facilitates human understanding and communication.* The technique should be able to represent the process in a form that most customers and developers can understand, encouraging communication about the process and agreement on its form and improvements. The technique should include sufficient information to allow one or more people to actually perform the process. And the model or tool should form a basis for training.
2. *Supports process improvement.* The technique should identify the essential components of a development or maintenance process. It should allow reuse of processes

or subprocesses on subsequent projects, compare alternatives, and estimate the impact of changes before the process is actually put into practice. Similarly, the technique should assist in selecting tools and techniques for the process, in encouraging organizational learning, and in supporting continuing evolution of the process.

3. *Supports process management.* The technique should allow the process to be project-specific. Then, developers and customers should be able to reason about attributes of software creation or evolution. The technique should also support planning and forecasting, monitoring and managing the process, and measuring key process characteristics.
4. *Provides automated guidance in performing the process.* The technique should define all or part of the software development environment, provide guidance and suggestions, and retain reusable process representations for later use.
5. *Supports automated process execution.* The technique should automate all or part of the process, support cooperative work, capture relevant measurement data, and enforce rules to ensure process integrity.

These characteristics can act as useful guidelines for selecting a process modeling technique for your development project. Item 4 is especially important if your organization is attempting to standardize its process; tools can help prompt developers about what to do next and provide gateways and checkpoints to assure that an artifact meets certain standards before the next steps are taken. For example, a tool can check a set of code components, evaluating their size and structure. If size or structure exceeds predefined limits, the developers can be notified before testing begins, and some components may be reviewed and perhaps redesigned.

2.5 INFORMATION SYSTEMS EXAMPLE

Let us consider which development process to use for supporting our information system example, the Piccadilly Television advertising program. Recall that there are many constraints on what kinds of advertising can be sold when, and that the regulations may change with rulings by the Advertising Standards Authority and other regulatory bodies. Thus, we want to build a software system that is easily maintained and changed. There is even a possibility that the constraints may change as we are building the system.

The waterfall model may be too rigid for our system, since it permits little flexibility after the requirements analysis stage is complete. Prototyping may be useful for building the user interface, so we may want to include some kind of prototyping in our model. But most of the uncertainty lies in the advertising regulations and business constraints. We want to use a process model that can be used and reused as the system evolves. A variation of the spiral model may be a good candidate for building the Piccadilly system, because it encourages us to revisit our assumptions, analyze our risks, and prototype various system characteristics. The repeated evaluation of alternatives, shown in the upper-left-hand quadrant of the spiral, helps us build flexibility into our requirements and design.

Boehm's representation of the spiral is high-level, without enough detail to direct the actions of analysts, designers, coders, and testers. However, there are many techniques and tools for representing the process model at finer levels of detail. The choice

of technique or tool depends in part on personal preference and experience, and in part on suitability for the type of process being represented. Let us see how Lai's notation might be used to represent part of the Piccadilly system's development process.

Because we want to use the spiral model to help us manage risk, we must include a characterization of "risk" in our process model. That is, risk is an artifact that we must describe, so we can measure and track risk in each iteration of our spiral. Each potential problem has an associated risk, and we can think of the risk in terms of two facets: probability and severity. **Probability** is the likelihood that a particular problem will occur, and **severity** is the impact it will have on the system. For example, suppose we are considering the problem of insufficient training in the development method being used to build the Piccadilly system. We may decide to use an object-oriented approach, but we may find that the developers assigned to the project have little or no experience in object orientation. This problem may have a low probability of occurring, since all new employees are sent to an intensive, four-week course on object-oriented development. On the other hand, should the problem actually occur, it would have a severe impact on the ability of the development team to finish the software within the assigned schedule. Thus, the probability of occurrence is low, but the severity is large.

We can represent these risk situations in a Lai artifact table, shown in Table 2.2. Here, risk is the artifact, with subartifacts probability and severity. For simplicity, we

TABLE 2.2 Artifact Definition Form for Artifact "Risk"

Name	*Risk (ProblemX)*	
Synopsis process	*This is the artifact that represents the risk that problem X will occur and have a negative affect on some aspect of the development process.*	
Complexity type	*Composite*	
Data type	*(risk_s, user_defined)*	
Artifact-state list		
low	$((state\_of(probability.x) = low)$ $(state\_of(severity.x) = small)$	Probability of problem is low, severity problem impact is small.
high-medium	$((state\_of(probability.x) = low)$ $(state\_of(severity.x) = large))$	Probability of problem is low, severity problem impact is large.
low-medium	$((State\_of(probability.x) = high)$ $(state\_of(severity.x) = small))$	Probability of problem is high, severity problem impact is small.
high	$((state\_of(probability.x) = high)$ $(state\_of(severity.x) = large))$	Probability of problem is high, severity problem impact is large.
Subartifact list		
	probability.x	*The probability that problem X will occur.*
	severity.x	*The severity of the impact should problem X occur on the project.*

have chosen only two states for each subartifact: low and high for probability, and small and large for severity. In fact, each of the subartifacts can have a large range of states (such as extremely small, very small, somewhat small, medium, somewhat high, very high, extremely high), leading to many different states for the artifact itself.

In the same way, we can define the other aspects of our development process and use diagrams to illustrate the activities and their interconnections. Modeling the process in this way has many advantages, not the least of which is building a common understanding of what development will entail. If users, customers, and developers participate in defining and depicting Piccadilly's development process, each will have expectations about what activities are involved, what they produce, and when each product can be expected. In particular, the combination of spiral model and risk table can be used to evaluate the risks periodically. With each revolution around the spiral, the probability and severity of each risk can be revisited and restated; when risks are unacceptably high, the process model can be revised to include risk mitigation and reduction techniques, as we will see in Chapter 3.

2.6 REAL-TIME EXAMPLE

The Ariane-5 software involved the reuse of software from Ariane-4. Reuse was intended to reduce risk, increase productivity, and increase quality. Thus, any process model for developing new Ariane software should include reuse activities. In particular, the process model must include activities to check the quality of reusable components, with safeguards to make sure that the reused software works properly within the context of the design of the new system.

Such a process model might look like the simplified model of Figure 2.17. The boxes in the model represent activities. The arrows entering the box from the left are resources, and those leaving on the right are outputs. Those entering from the top are controls or constraints, such as schedules, budgets, or standards. And those entering from below are mechanisms that assist in performing the activity, such as tools, databases, or techniques.

The Ariane-4 reuse process begins with the software's mission, namely, controlling a new rocket, as well as software from previous airframes, unmet needs, and other software components available from other sources (such as purchased software or reuse repositories from other projects). Based on the business strategy of the aerospace builder, the developers can identify reusable subprocesses, describe them (perhaps with annotations related to past experience), and place them in a library for consideration by the requirements analysts. The reusable processes will often involve reusable components (i.e., reusable requirements, design or code components, or even test cases, process descriptions, and other documents and artifacts).

Next, the requirements analysts examine the requirements for the new airframe and the reusable components that are available in the library. They produce a revised set of requirements, consisting of a mix of new and reused requirements. Then, the designers use those requirements to design the software. Once their design is complete, they evaluate all reused design components to certify that they are correct and consistent with the new parts of the design and the overall intention of the system as described in the requirements. Finally, the certified components are used to build or

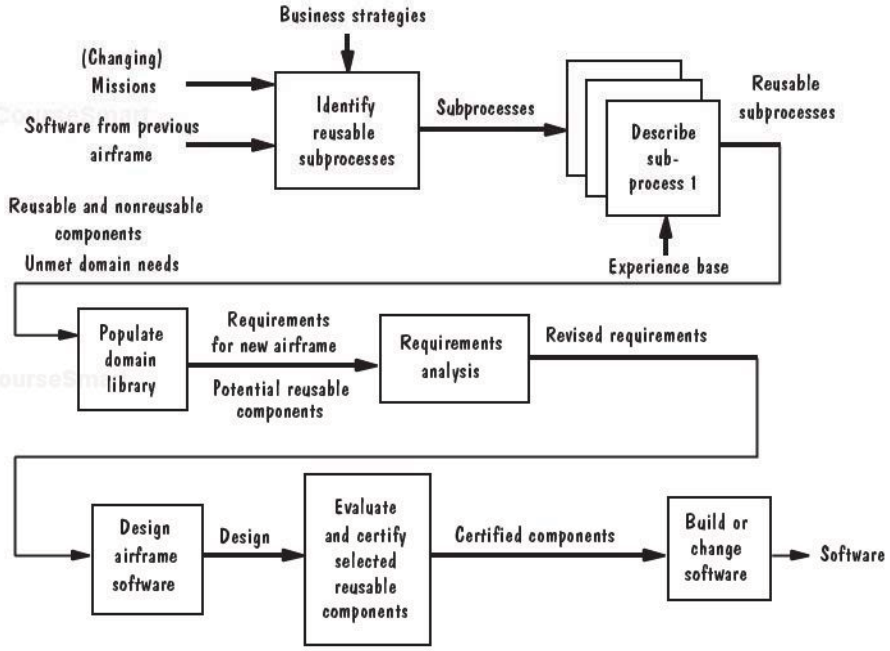


FIGURE 2.17 Reuse process model for new airframe software.

change the software and produce the final system. As we will see in later chapters, such a process might have prevented the destruction of Ariane-5.

2.7 WHAT THIS CHAPTER MEANS FOR YOU

In this chapter, we have seen that the software development process involves activities, resources, and products. A process model is useful for guiding your behavior when you are working with a group. Detailed process models tell you how to coordinate and collaborate with your colleagues as you design and build a system. We have also seen that process models include organizational, functional, behavioral, and other perspectives, so you can focus on particular aspects of the development process to enhance your understanding or guide your actions.

2.8 WHAT THIS CHAPTER MEANS FOR YOUR DEVELOPMENT TEAM

A process model has clear advantages for your development team, too. A good model shows each team member which activities occur when, and by whom, so that the division of duties is clear. In addition, the project manager can use process tools to enact the process, simulating activities and tracking resources to determine the best mix of people and activities in order to meet the project's budget and schedule. This simulation is done before resources are actually committed, so time and money are saved by not having to

backtrack or correct mistakes. Indeed, iteration and incremental development can be included in the process model, so the team can learn from prototyping or react to evolving requirements and still meet the appropriate deadlines.

2.9 WHAT THIS CHAPTER MEANS FOR RESEARCHERS

Process modeling is a rich field of research interest in software engineering. Many software developers feel that, by using a good process, the quality of the products of development can be guaranteed. Thus, there are several areas into which researchers are looking:

- *Process notations:* How to write down the process in a way that is understandable to those who must carry it out
- *Process models:* How to depict the process, using an appropriate set of activities, resources, products, and tools
- *Process modeling support tools:* How to enact or simulate a process model, so that resource availability, usage, and performance can be assessed
- *Process measurement and assessment:* How to determine which activities, resources, subprocesses, and model types are best for producing high-quality products in a specified time or environment

Many of these efforts are coordinated with process improvement research, an area we will investigate in Chapter 13.

2.10 TERM PROJECT

It is early in the development process of the Loan Arranger system for FCO. You do not yet have a comprehensive set of requirements for the system. All you have is an overview of system functionality, and a sense of how the system will be used to support FCO's business. Many of the terms used in the overview are unfamiliar to you, so you have asked the customer representatives to prepare a glossary. They give you the description in Table 2.3.

This information clarifies some concepts for you, but you are still far from having a good set of requirements. Nevertheless, you can make some preliminary decisions about how the development should proceed. Review the processes presented in this chapter and determine which ones might be appropriate for developing the Loan Arranger. For each process, make a list of its advantages and disadvantages with respect to the Loan Arranger.

2.11 KEY REFERENCES

As a result of the Fifth International Software Process Workshop, a working group chaired by Kellner formulated a standard problem, to be used to evaluate and compare some of the more popular process modeling techniques. The problem was designed to be complex enough so that it would test a technique's ability to include each of the following:

- multiple levels of abstraction
- control flow, sequencing, and constraints on sequencing

TABLE 2.3 Glossary of Terms for the Loan Arranger

Borrower: A borrower is the recipient of money from a lender. Borrowers may receive loans jointly; that is, each loan may have multiple borrowers. Each borrower has an associated name and a unique borrower identification number.

Borrower's risk: The risk factor associated with any borrower is based on the borrower's payment history. A borrower with no loans outstanding is assigned a nominal borrower's risk factor of 50. The risk factor decreases when the borrower makes payments on time but increases when a borrower makes a payment late or defaults on a loan. The borrower's risk is calculated using the following formula:

$$\begin{aligned}\text{Risk} = 50 &- [10 \times (\text{number of years of loans in good standing})] \\ &+ [20 \times (\text{number of years of loans in late standing})] \\ &+ [30 \times (\text{number of years of loans in default standing})]\end{aligned}$$

For example, a borrower may have three loans. The first loan was taken out two years ago, and all payments have been made on time. That loan is in good standing and has been so for two years. The second and third loans are four and five years old, respectively, and each one was in good standing until recently. Thus, each of the two late-standing loans has been in late standing only for one year. Thus, the risk is

$$50 - [10 \times 2] + [20 \times (1 + 1)] + [30 \times 0] = 70.$$

The maximum risk value is 100, and the minimum risk value is 1.

Bundle: A bundle is a collection of loans that has been associated for sale as a single unit to an investor. Associated with each bundle is the total value of loans in the bundle, the period of time over which the loans in the bundle are active (i.e., for which borrowers are still making payments on the loans), an estimate of the risk involved in purchasing the bundle, and the profit to be made when all loans are paid back by the borrowers.

Bundle risk: The risk of a loan bundle is the weighted average of the risks of the loans in the bundle, with each loan's risk (see *loan risk,* below) weighted according to that loan's value. To calculate the weighted average over n loans, assume that each loan Li has remaining principal Pi and loan risk Ri. The weighted average is then

$$\frac{\sum_{i=1}^{n} PiRi}{\sum_{i=1}^{n} Pi}$$

Discount: The discount is the price at which FCO is willing to sell a loan to an investor. It is calculated according to the formula

$$\text{Discount} = (\text{principal remaining}) \times [(\text{interest rate}) \times (0.2 + (.005 \times (101 - (\text{loan risk}))))]$$

Interest rate type: An interest rate on a loan is either fixed or adjustable. A fixed-rate loan (called an FM) has the same interest rate for the term of the mortgage. An adjustable rate loan (called an ARM) has a rate that changes each year, based on a government index supplied by the U.S. Department of the Treasury.

Investor: An investor is a person or organization that is interested in purchasing a bundle of loans from FCO.

Investment request: An investor makes an investment request, specifying a maximum degree of risk at which the investment will be made, the minimum amount of profit required in a bundle, and the maximum period of time over which the loans in the bundle must be paid.

Lender: A lender is an institution that makes loans to borrowers. A lender can have zero, one, or many loans.

Lender information: Lender information is descriptive data that are imported from outside the application. Lender information cannot be changed or deleted. The following information is associated with each lender: lender name (institution), lender contact (person at that institution), phone number for contact, a unique lender identification number. Once added to the system, a lender entry can be edited but not removed.

(continues)

TABLE 2.3 (*continued*)

Lending institution: A synonym for lender. See *lender*.

Loan: A loan is a set of information that describes a home loan and the borrower-identifying information associated with the loan. The following information is associated with each loan: loan amount, interest rate, interest rate type (adjustable or fixed), settlement date (the date the borrower originally borrowed the money from the lender), term (expressed as number of years), borrower, lender, loan type (jumbo or regular), and property (identified by the address of the property). A loan must have exactly one associated lender and exactly one associated borrower. In addition, each loan is identified with a loan risk and a loan status.

Loan analyst: The loan analyst is a professional employee of FCO who is trained in using the Loan Arranger system to manage and bundle loans. Loan analysts are familiar with the terminology of loans and lending, but they may not have all the relevant information at hand with which to evaluate a single loan or collection of loans.

Loan risk: Each loan is associated with a level of risk, indicated by an integer from 1 to 100. 1 represents the lowest-risk loan; that is, it is unlikely that the borrower will be late or default on this loan. 100 represents the highest risk; that is, it is almost certain that the borrower will default on this loan.

Loan status: A loan can have one of three status designations: good, late, or default. A loan is in good status if the borrower has made all payments up to the current time. A loan is in late status if the borrower's last payment was made but not by the payment due date. A loan is in default status if the borrower's last payment was not received within 10 days of the due date.

Loan type: A loan is either a jumbo mortgage, where the property is valued in excess of $275,000, or a regular mortgage, where the property value is $275,000 or less.

Portfolio: The collection of loans purchased by FCO and available for inclusion in a bundle. The repository maintained by the Loan Arranger contains information about all of the loans in the portfolio.

- decision points
- iteration and feedback to earlier steps
- user creativity
- object and information management, as well as flow through the process
- object structure, attributes, and interrelationships
- organizational responsibility for specific tasks
- physical communication mechanisms for information transfer
- process measurements
- temporal aspects (both absolute and relative)
- tasks executed by humans
- professional judgment or discretion
- connection to narrative explanations
- tasks invoked or executed by a tool
- resource constraints and allocation, schedule determination
- process modification and improvement
- multiple levels of aggregation and parallelism

Eighteen different process modeling techniques were applied to the common problem, and varying degrees of satisfaction were found with each one. The results are reported in Kellner and Rombach (1990).

Curtis, Kellner, and Over (1992) present a comprehensive survey of process modeling techniques and tools. The paper also summarizes basic language types and constructs and gives examples of process modeling approaches that use those language types.

Krasner et al. (1992) describe lessons learned when implementing a software process modeling system in a commercial environment.

Several Web sites contain information about process modeling.

- The U.S. Software Engineering Institute (SEI) continues to investigate process modeling as part of its process improvement efforts. A list of its technical reports and activities can be found at http://www.sei.cmu.edu. The information at http://www.sei.cmu.edu/collaborating/spins/ describes Software Process Improvement Networks, geographically-based groups of people interested in process improvement who often meet to hear speakers or discuss process-related issues.
- The European Community has long sponsored research in process modeling and a process model language. Descriptions of current research projects are available at http://cordis.europa.eu/fp7/projects_en.html.
- The Data and Analysis Centre for Software Engineering maintains a list of resources about software process at https://www.thedacs.com/databases/url/key/39.

More information about Lai notation is available in David Weiss and Robert Lai's book, Software Product Line Engineering: A Family-based Software Development Process (Weiss and Lai 1999).

The University of Southern California's Center for Software Engineering has developed a tool to assist you in selecting a process model suitable for your project's requirements and constraints. It can be ftp-ed from ftp://usc.edu/pub/soft_engineering/demos/pmsa.zip, and more information can be found on the Center's Web site: http://sunset.usc.edu.

Journals such as *Software Process—Improvement and Practice* have articles addressing the role of process modeling in software development and maintenance. They also report the highlights of relevant conferences, such as the International Software Process Workshop and the International Conference on Software Engineering. The July/August 2000 issue of *IEEE Software* focuses on process diversity and has several articles about the success of a process maturity approach to software development.

There are many resources available for learning about agile methods. The Agile Manifesto is posted at http://www.agilealliance.org. Kent Beck's (1999) is the seminal book on extreme programming, and Alistair Cockburn (2002) describes the Crystal family of methodologies. Martin Beck (1999) explains refactoring, which is one of the most difficult steps of XP. Two excellent references on agile methods are Robert C. Martin's (2003) book on agile software development, and Daniel H. Steinberg and Daniel W. Palmer's (2004) book on extreme software engineering. Two Web sites providing additional information about extreme programming are http://www.xprgramming.com and http://www.extremeprogramming.org.

2.12 EXERCISES

1. How does the description of a system relate to the notion of process models? For example, how do you decide what the boundary should be for the system described by a process model?
2. For each of the process models described in this chapter, what are the benefits and drawbacks of using the model?
3. For each of the process models described in this chapter, how does the model handle a significant change in requirements late in development?
4. Draw a diagram to capture the process of buying an airplane ticket for a business trip.
5. Draw a Lai artifact table to define a module. Make sure that you include artifact states that show the module when it is untested, partially tested, and completely tested.
6. Using the notation of your choice, draw a process diagram of a software development process that prototypes three different designs and choose the best from among them.
7. Examine the characteristics of good process models described in Section 2.4. Which characteristics are essential for processes to be used on projects where the problem and solution are not well understood?
8. In this chapter, we suggested that software development is a creation process, not a manufacturing process. Discuss the characteristics of manufacturing that apply to software development and explain which characteristics of software development are more like a creative endeavor.
9. Should a development organization adopt a single process model for all of its software development? Discuss the pros and cons.
10. Suppose your contract with a customer specifies that you use a particular software development process. How can the work be monitored to enforce the use of this process?
11. Consider the processes introduced in this chapter. Which ones give you the most flexibility to change in reaction to changing requirements?
12. Suppose Amalgamated, Inc., requires you to use a given process model when it contracts with you to build a system. You comply, building software using the prescribed activities, resources, and constraints. After the software is delivered and installed, your system experiences a catastrophic failure. When Amalgamated investigates the source of the failure, you are accused of not having done code reviews that would have found the source of the problem before delivery. You respond that code reviews were not in the required process. What are the legal and ethical issues involved in this dispute?

3 Planning and Managing the Project

In this chapter, we look at
- tracking project progress
- project personnel and organization
- effort and schedule estimation
- risk management
- using process modeling with project planning

As we saw in the previous chapters, the software development cycle includes many steps, some of which are repeated until the system is complete and the customers and users are satisfied. However, before committing funds for a software development or maintenance project, a customer usually wants an estimate of how much the project will cost and how long the project will take. This chapter examines the activities necessary to plan and manage a software development project.

3.1 TRACKING PROGRESS

Software is useful only if it performs a desired function or provides a needed service. Thus, a typical project begins when a customer approaches you to discuss a perceived need. For example, a large national bank may ask you for help in building an information system that allows the bank's clients to access their account information, no matter where in the world the clients are. Or you may be contacted by marine biologists who would like a system to connect with their water-monitoring equipment and perform statistical analyses of the data gathered. Usually, customers have several questions to be answered:

- Do you understand my problem and my needs?
- Can you design a system that will solve my problem or satisfy my needs?
- How long will it take you to develop such a system?
- How much will it cost to have you develop such a system?

Answering the last two questions requires a well-thought-out project schedule. A **project schedule** describes the software development cycle for a particular project by enumerating the phases or stages of the project and breaking each into discrete tasks or activities to be done. The schedule also portrays the interactions among these activities and estimates the time that each task or activity will take. Thus, the schedule is a timeline that shows when activities will begin and end, and when the related development products will be ready.

In Chapter 1, we learned that a systems approach involves both analysis and synthesis: breaking the problem into its component parts, devising a solution for each part, and then putting the pieces together to form a coherent whole. We can use this approach to determine the project schedule. We begin by working with customers and potential users to understand what they want and need. At the same time, we make sure that they are comfortable with our knowledge of their needs. We list all project **deliverables**, that is, the items that the customer expects to see during project development. Among the deliverables may be

- documents
- demonstrations of function
- demonstrations of subsystems
- demonstrations of accuracy
- demonstrations of reliability, security, or performance

Next, we determine what activities must take place in order to produce these deliverables. We may use some of the process modeling techniques we learned in Chapter 2, laying out exactly what must happen and which activities depend on other activities, products, or resources. Certain events are designated to be milestones, indicating to us and our customers that a measurable level of progress has been made. For example, when the requirements are documented, inspected for consistency and completeness, and turned over to the design team, the requirements specification may be a project milestone. Similarly, milestones may include the completion of the user's manual, the performance of a given set of calculations, or a demonstration of the system's ability to communicate with another system.

In our analysis of the project, we must distinguish clearly between milestones and activities. An **activity** is a part of the project that takes place over a period of time, whereas a **milestone** is the completion of an activity—a particular point in time. Thus, an activity has a beginning and an end, whereas a milestone is the end of a specially designated activity. For example, the customer may want the system to be accompanied by an online operator tutorial. The development of the tutorial and its associated programs is an activity; it culminates in the demonstration of those functions to the customer: the milestone.

By examining the project carefully in this way, we can separate development into a succession of phases. Each phase is composed of steps, and each step can be subdivided further if necessary, as shown in Figure 3.1.

To see how this analysis works, consider the phases, steps, and activities of Table 3.1, which describes the building of a house. First, we consider two phases: landscaping the lot and building the house itself. Then, we break each phase into smaller steps, such as

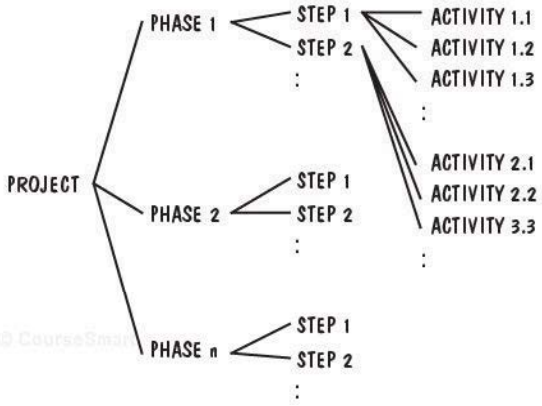


FIGURE 3.1 Phases, steps, and activities in a project.

clearing and grubbing, seeding the turf, and planting trees and shrubs. Where necessary, we can divide the steps into activities; for example, finishing the interior involves completing the interior plumbing, interior electrical work, wallboard, interior painting, floor covering, doors, and fixtures. Each activity is a measurable event and we have objective criteria to determine when the activity is complete. Thus, any activity's end can be a milestone, and Table 3.2 lists the milestones for phase 2.

This analytical breakdown gives us and our customers an idea of what is involved in constructing a house. Similarly, analyzing a software development or maintenance project and identifying the phases, steps, and activities, both we and our customers have a better grasp of what is involved in building and maintaining a system. We saw in Chapter 2 that a process model provides a high-level view of the phases and steps, so process modeling is a useful way to begin analyzing the project. In later chapters, we will see that the major phases, such as requirements engineering, implementation, or testing, involve many activities, each of which contributes to product or process quality.

Work Breakdown and Activity Graphs

Analysis of this kind is sometimes described as generating a **work breakdown structure** for a given project, because it depicts the project as a set of discrete pieces of work. Notice that the activities and milestones are items that both customer and developer can use to track development or maintenance. At any point in the process, the customer may want to follow our progress. We developers can point to activities, indicating what work is under way, and to milestones, indicating what work has been completed. However, a project's work breakdown structure gives no indication of the interdependence of the work units or of the parts of the project that can be developed concurrently.

We can describe each activity with four parameters: the precursor, duration, due date, and endpoint. A **precursor** is an event or set of events that must occur before the activity can begin; it describes the set of conditions that allows the activity to begin. The duration is the length of time needed to complete the activity. The **due date** is the date by which the activity must be completed, frequently determined by contractual deadlines. Signifying that the activity has ended, the **endpoint** is usually a milestone or

TABLE 3.1 Phases, Steps, and Activities of Building a House

Phase 1: Landscaping the Lot	Phase 2: Building the House
Step 1.1: Clearing and grubbing	*Step 2.1: Prepare the site*
Activity 1.1.1: Remove trees	Activity 2.1.1: Survey the land
Activity 1.1.2: Remove stumps	Activity 2.1.2: Request permits
Step 1.2: Seeding the turf	Activity 2.1.3: Excavate for the foundation
Activity 1.2.1: Aerate the soil	Activity 2.1.4: Buy materials
Activity 1.2.2: Disperse the seeds	*Step 2.2: Building the exterior*
Activity 1.2.3: Water and weed	Activity 2.2.1: Lay the foundation
Step 1.3: Planting shrubs and trees	Activity 2.2.2: Build the outside walls
Activity 1.3.1: Obtain shrubs and trees	Activity 2.2.3: Install exterior plumbing
Activity 1.3.2: Dig holes	Activity 2.2.4: Exterior electrical work
Activity 1.3.3: Plant shrubs and trees	Activity 2.2.5: Exterior siding
Activity 1.3.4: Anchor the trees and mulch around them	Activity 2.2.6: Paint the exterior
	Activity 2.2.7: Install doors and fixtures
	Activity 2.2.8: Install roof
	Step 2.3: Finishing the interior
	Activity 2.3.1: Install the interior plumbing
	Activity 2.3.2: Install interior electrical work
	Activity 2.3.3: Install wallboard
	Activity 2.3.4: Paint the interior
	Activity 2.3.5: Install floor covering
	Activity 2.3.6: Install doors and fixtures

TABLE 3.2 Milestones in Building a House

1.1.	Survey complete
1.2.	Permits issued
1.3.	Excavation complete
1.4.	Materials on hand
2.1.	Foundation laid
2.2.	Outside walls complete
2.3.	Exterior plumbing complete
2.4.	Exterior electrical work complete
2.5.	Exterior siding complete
2.6.	Exterior painting complete
2.7.	Doors and fixtures mounted
2.8.	Roof complete
3.1.	Interior plumbing complete
3.2.	Interior electrical work complete
3.3.	Wallboard in place
3.4.	Interior painting complete
3.5.	Floor covering laid
3.6.	Doors and fixtures mounted

deliverable. We can illustrate the relationships among activities by using these parameters. In particular, we can draw an **activity graph** to depict the dependencies; the nodes of the graph are the project milestones, and the lines linking the nodes represent the activities involved. Figure 3.2 is an activity graph for the work described in phase 2 of Table 3.1.

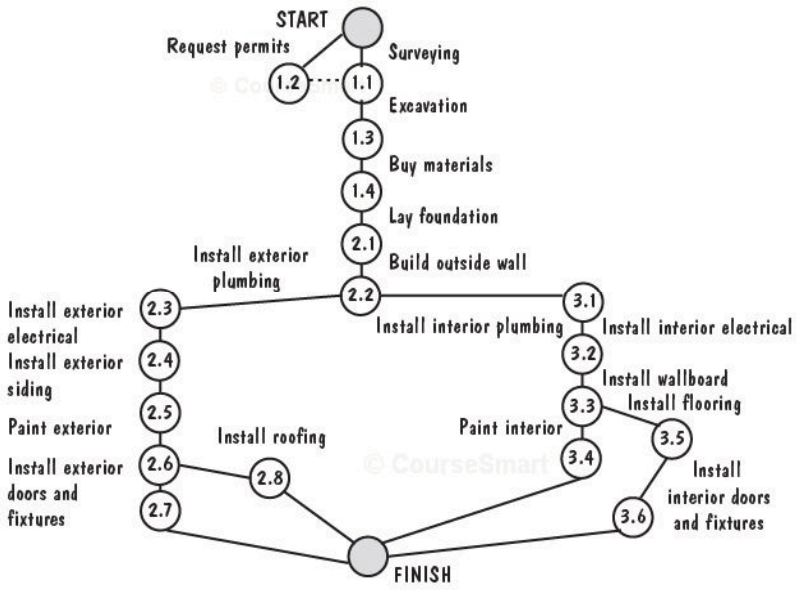


FIGURE 3.2 Activity graph for building a house.

Many important characteristics of the project are made visible by the activity graph. For example, it is clear from Figure 3.2 that neither of the two plumbing activities can start before milestone 2.2 is reached; that is, 2.2 is a precursor to both interior and exterior plumbing. Furthermore, the figure shows us that several things can be done simultaneously. For instance, some of the interior and exterior activities are independent (such as installing wallboard, connecting exterior electrical plumbing, and others leading to milestones 2.6 and 3.3, respectively). The activities on the left-hand path do not depend on those on the right for their initiation, so they can be worked on concurrently. Notice that there is a dashed line from requesting permits (node 1.2) to surveying (node 1.1). This line indicates that these activities must be completed before excavation (the activity leading to milestone 1.3) can begin. However, since there is no real activity that occurs after reaching milestone 1.2 in order to get to milestone 1.1, the dashed line indicates a relationship without an accompanying activity.

It is important to realize that activity graphs depend on an understanding of the parallel nature of tasks. If work cannot be done in parallel, then the (mostly straight) graph is not useful in depicting how tasks will be coordinated. Moreover, the graphs must reflect a realistic depiction of the parallelism. In our house-building example, it is clear that some of the tasks, like plumbing, will be done by different people from those doing other tasks, like electrical work. But on software development projects, where some people have many skills, the theoretical parallelism may not reflect reality. A restricted number of people assigned to the project may result in the same person doing many things in series, even though they could be done in parallel by a larger development team.

Estimating Completion

We can make an activity graph more useful by adding to it information about the estimated time it will take to complete each activity. For a given activity, we label the corresponding edge of the graph with the estimate. For example, for the activities in phase 2 of Table 2.1, we can append to the activity graph of Figure 3.2 estimates of the number of days it will take to complete each activity. Table 3.3 contains the estimates for each activity.

The result is the graph shown in Figure 3.3. Notice that milestones 2.7, 2.8, 3.4, and 3.6 are precursors to the finish. That is, these milestones must all be reached in order to consider the project complete. The zeros on the links from those nodes to the finish show that no additional time is needed. There is also an implicit zero on the link from node 1.2 to 1.1, since no additional time is accrued on the dashed link.

This graphical depiction of the project tells us a lot about the project's schedule. For example, since we estimated that the first activity would take 3 days to complete, we cannot hope to reach milestone 1.1 before the end of day 3. Similarly, we cannot reach milestone 1.2 before the end of day 15. Because the beginning of excavation (activity 1.3) cannot begin until milestones 1.1 and 1.2 are both reached, excavation cannot begin until the beginning of day 16.

Analyzing the paths among the milestones of a project in this way is called the **Critical Path Method (CPM)**. The paths can show us the minimum amount of time it will take to complete the project, given our estimates of each activity's duration. Moreover, CPM reveals those activities that are most critical to completing the project on time.

TABLE 3.3 Activities and Time Estimates

Activity	Time Estimate (in Days)
Step 1: Prepare the site	
Activity 1.1: Survey the land	3
Activity 1.2: Request permits	15
Activity 1.3: Excavate for the foundation	10
Activity 1.4: Buy materials	10
Step 2: Building the exterior	
Activity 2.1: Lay the foundation	15
Activity 2.2: Build the outside walls	20
Activity 2.3: Install exterior plumbing	10
Activity 2.4: Install exterior electrical work	10
Activity 2.5: Install exterior siding	8
Activity 2.6: Paint the exterior	5
Activity 2.7: Install doors and fixtures	6
Activity 2.8: Install roof	9
Step 3: Finishing the interior	
Activity 3.1: Install interior plumbing	12
Activity 3.2: Install interior electrical work	15
Activity 3.3: Install wallboard	9
Activity 3.4: Paint the interior	18
Activity 3.5: Install floor covering	11
Activity 3.6: Install doors and fixtures	7

To see how CPM works, consider again our house-building example. First, we notice that the activities leading to milestones 1.1 (surveying) and 1.2 (requesting permits) can occur concurrently. Since excavation (the activity culminating in milestone 1.3) cannot begin until day 16, surveying has 15 days in which to be completed, even though it is only 3 days in duration. Thus, surveying has 15 days of available time, but requires only 3 days of real time. In the same way, for each activity in our graph, we can compute a pair of times: real time and available time. The **real time** or **actual time** for an activity is the estimated amount of time required for the activity to be completed, and the **available time** is the amount of time available in the schedule for the activity's completion. **Slack time** or **float** for an activity is the difference between the available time and the real time for that activity:

$$\text{Slack time} = \text{available time} - \text{real time}$$

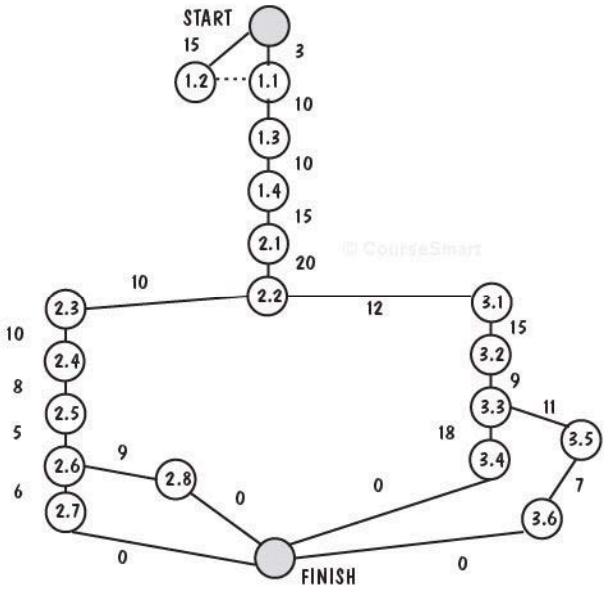


FIGURE 3.3 Activity graph with durations.

Another way of looking at slack time is to compare the earliest time an activity may begin with the latest time the activity may begin without delaying the project. For example, surveying may begin on day 1, so the earliest start time is day 1. However, because it will take 15 days to request and receive permits, surveying can begin as late as day 13 and still not hold up the project schedule. Therefore,

$$\text{Slack time} = \text{latest start time} - \text{earliest start time}$$

Let us compute the slack for our example's activities to see what it tells us about the project schedule. We compute slack by examining all paths from the start to the finish. As we have seen, it must take 15 days to complete milestones 1.1 and 1.2. An additional 55 days are used in completing milestones 1.3, 1.4, 2.1, and 2.2. At this point, there are four possible paths to be taken:

1. Following milestones 2.3 through 2.7 on the graph requires 39 days.
2. Following milestones 2.3 through 2.8 on the graph requires 42 days.
3. Following milestones 3.1 through 3.4 on the graph requires 54 days.
4. Following milestones 3.1 through 3.6 on the graph requires 54 days.

Because milestones 2.7, 2.8, 3.4, and 3.6 must be met before the project is finished, our schedule is constrained by the longest path. As you can see from Figure 3.3 and our preceding calculations, the two paths on the right require 124 days to complete, and the two paths on the left require fewer days. To calculate the slack, we can work backward along the path to see how much slack time there is for each activity leading to a node. First, we note that there is zero slack on the longest path. Then, we examine each of the remaining nodes to calculate the slack for the activities leading to them. For example,

54 days are available to complete the activities leading to milestones 2.3, 2.4, 2.5, 2.6, and 2.8, but only 42 days are needed to complete these. Thus, this portion of the graph has 12 days of slack. Similarly, the portion of the graph for activities 2.3 through 2.7 requires only 39 days, so we have 15 days of slack along this route. By working forward through the graph in this way, we can compute the earliest start time and slack for each of the activities. Then, we compute the latest start time for each activity by moving from the finish back through each node to the start. Table 3.4 shows the results: the slack time for each activity in Figure 3.3. (At milestone 2.6, the path can branch to 2.7 or 2.8. The latest start times in Table 3.4 are calculated by using the route from 2.6 to 2.8, rather than from 2.6 to 2.7.)

The longest path has a slack of zero for each of its nodes, because it is the path that determines whether or not the project is on schedule. For this reason, it is called the critical path. Thus, the **critical path** is the one for which the slack at every node is zero. As you can see from our example, there may be more than one critical path.

TABLE 3.4 Slack Time for Project Activities

Activity	Earliest Start Time	Latest Start Time	Slack
1.1	1	13	12
1.2	1	1	0
1.3	16	16	0
1.4	26	26	0
2.1	36	36	0
2.2	51	51	0
2.3	71	83	12
2.4	81	93	12
2.5	91	103	12
2.6	99	111	12
2.7	104	119	15
2.8	104	116	12
3.1	71	71	0
3.2	83	83	0
3.3	98	98	0
3.4	107	107	0
3.5	107	107	0
3.6	118	118	0
Finish	124	124	0

Since the critical path has no slack, there is no margin for error when performing the activities along its route.

Notice what happens when an activity on the critical path begins late (i.e., later than its earliest start time). The late start pushes all subsequent critical path activities forward, forcing them to be late, too, if there is no slack. And for activities not on the critical path, the subsequent activities may also lose slack time. Thus, the activity graph helps us to understand the impact of any schedule slippage.

Consider what happens if the activity graph has several loops in it. Loops may occur when an activity must be repeated. For instance, in our house-building example, the building inspector may require the plumbing to be redone. In software development, a design inspection may require design or requirements to be respecified. The appearance of these loops may change the critical path as the loop activities are exercised more than once. In this case, the effects on the schedule are far less easy to evaluate.

Figure 3.4 is a bar chart that shows some software development project activities, including information about the early and late start dates; this chart is typical of those produced by automated project management tools. The horizontal bars represent the duration of each activity; those bars composed of asterisks indicate the critical path. Activities depicted by dashes and Fs are not on the critical path, and an F represents float or slack time.

Critical path analysis of a project schedule tells us who must wait for what as the project is being developed. It also tells us which activities must be completed on schedule to avoid delay. This kind of analysis can be enhanced in many ways. For instance, our house-building example supposes that we know exactly how long each activity will take. Often, this is not the case. Instead, we have only an estimated duration for an activity, based on our knowledge of similar projects and events. Thus, to each activity, we can

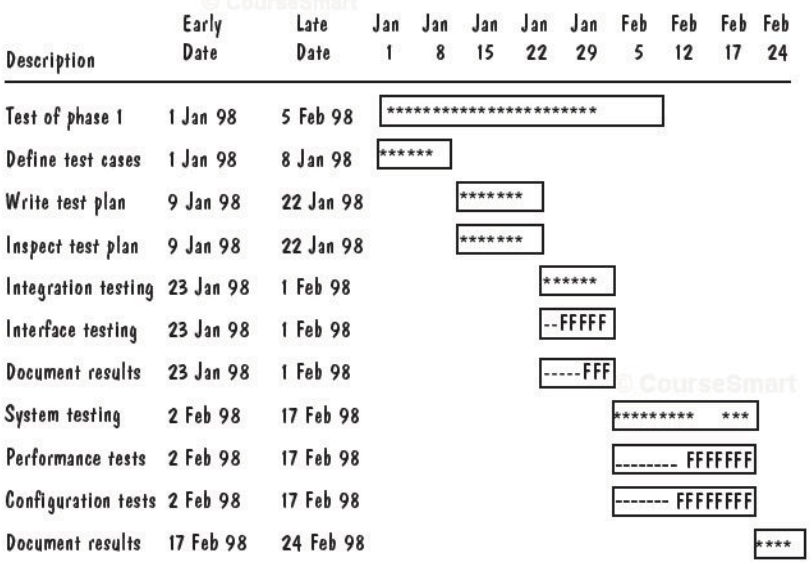


FIGURE 3.4 CPM bar chart.

assign a probable duration according to some probability distribution, so that each activity has associated with it an expected value and a variance. In other words, instead of knowing an exact duration, we estimate a window or interval in which the actual time is likely to fall. The expected value is a point within the interval, and the variance describes the width of the interval. You may be familiar with a standard probability distribution called a normal distribution, whose graph is a bell-shaped curve. The Program Evaluation and Review Technique (PERT) is a popular critical path analysis technique that assumes a normal distribution. (See Hillier and Lieberman [2001] for more information about PERT.) PERT determines the probability that the earliest start time for an activity is close to the scheduled time for that activity. Using information such as probability distribution, latest and earliest start times, and the activity graph, a PERT program can calculate the critical path and identify those activities most likely to be bottlenecks. Many project managers use the CPM or PERT method to examine their projects. However, these methods are valuable only for stable projects in which several activities take place concurrently. If the project's activities are mostly sequential, then almost all activities are on the critical path and are candidates for bottlenecks. Moreover, if the project requires redesign or rework, the activity graph and critical path are likely to change during development.

Tools to Track Progress

There are many tools that can be used to keep track of a project's progress. Some are manual, others are simple spreadsheet applications, and still others are sophisticated tools with complex graphics. To see what kinds of tools may be useful on your projects, consider the work breakdown structure depicted in Figure 3.5. Here, the overall objective is to build a system involving communications software, and the project manager has described the work in terms of five steps: system planning, system design, coding, testing, and delivery. For simplicity, we concentrate on the first two steps. Step 1 is then partitioned into four activities: reviewing the specifications, reviewing the budget, reviewing the schedule, and developing a project plan. Similarly, the system design is

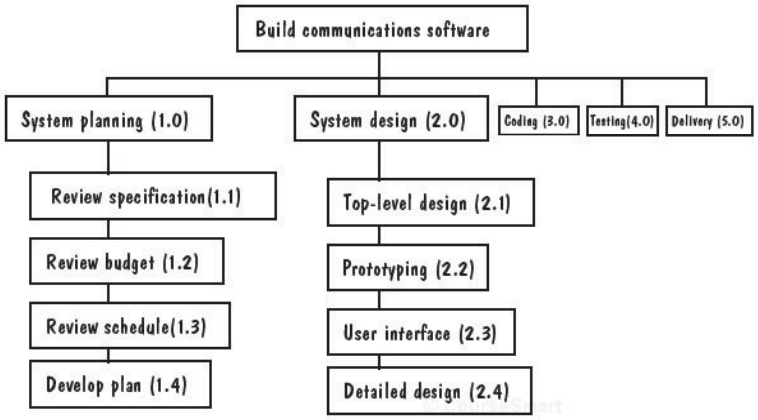


FIGURE 3.5 Example work breakdown structure.

developed by doing a top-level design, prototyping, designing the user interface, and then creating a detailed design.

Many project management software systems draw a work breakdown structure and also assist the project manager in tracking progress by step and activity. For example, a project management package may draw a **Gantt chart**, a depiction of the project where the activities are shown in parallel, with the degree of completion indicated by a color or icon. The chart helps the project manager to understand which activities can be performed concurrently, and also to see which items are on the critical path.

Figure 3.6 is a Gantt chart for the work breakdown structure of Figure 3.5. The project began in January, and the dashed vertical line labeled "today" indicates that the project team is working during the middle of May. A vertical bar shows progress on each activity, and the color of the bar denotes completion, duration, or criticality. A diamond icon shows us where there has been slippage, and the triangles designate an activity's start and finish. The Gantt chart is similar to the CPM chart of Figure 3.4, but it includes more information.

Simple charts and graphs can provide information about resources, too. For example, Figure 3.7 graphs the relationship between the people assigned to the project and those needed at each stage of development; it is typical of graphs produced by project management tools. It is easy to see that during January, February, and March,

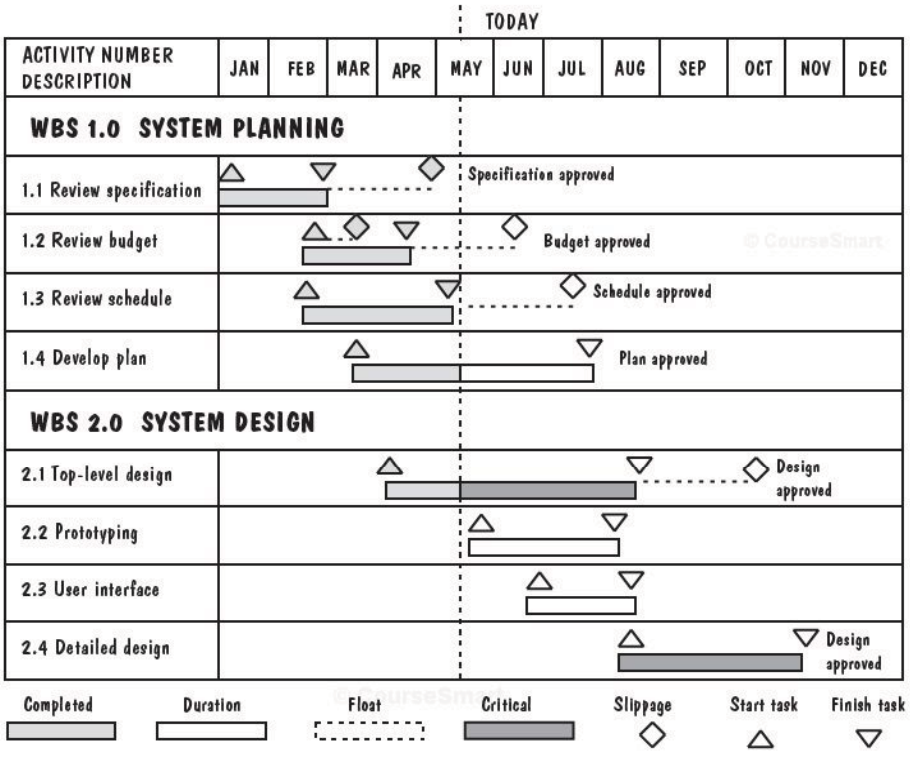


FIGURE 3.6 Gantt chart for example work breakdown structure.

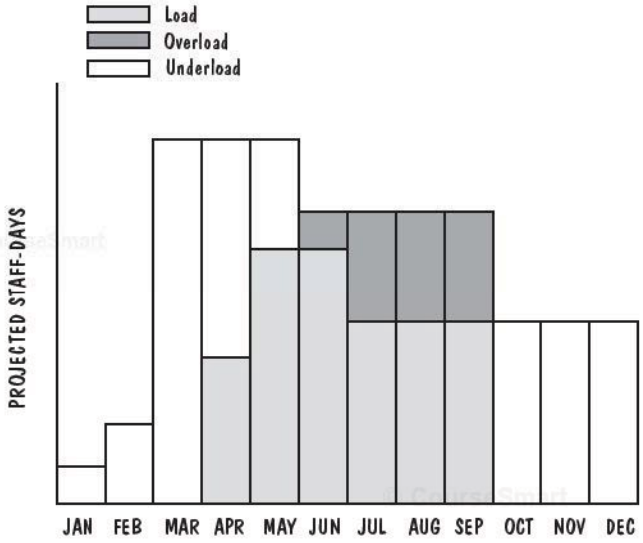


FIGURE 3.7 Resource histogram.

people are needed but no one is assigned. In April and May, some team members are working, but not enough to do the required job. On the other hand, the period during which there are too many team members is clearly shown: from the beginning of June to the end of September. The resource allocation for this project is clearly out of balance. By changing the graph's input data, you can change the resource allocation and try to reduce the overload, finding the best resource load for the schedule you have to meet.

Later in this chapter, we will see how to estimate the costs of development. Project management tools track actual costs against the estimates, so that budget progress can be assessed, too. Figure 3.8 shows an example of how expenditures can be

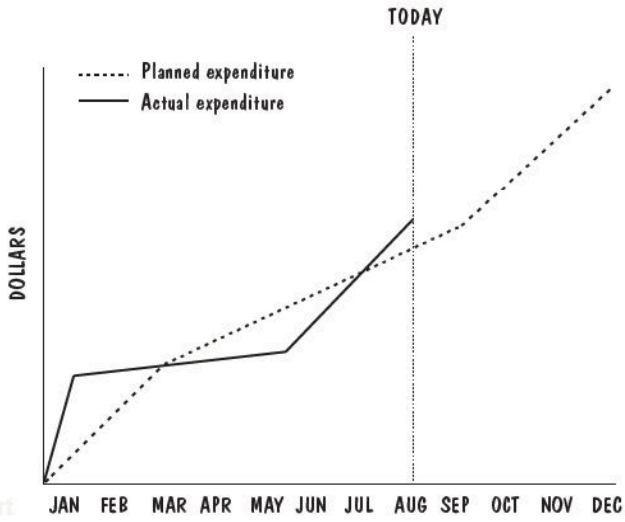


FIGURE 3.8 Tracking planned vs. actual expenditures.

monitored. By combining budget tracking with personnel tracking, you can use project management tools to determine the best resources for your limited budget.

3.2 PROJECT PERSONNEL

To determine the project schedule and estimate the associated effort and costs, we need to know approximately how many people will be working on the project, what tasks they will perform, and what abilities and experience they must have so they can do their jobs effectively. In this section, we look at how to decide who does what and how the staff can be organized.

Staff Roles and Characteristics

In Chapter 2, we examined several software process models, each depicting the way in which the several activities of software development are related. No matter the model, there are certain activities necessary to any software project. For example, every project requires people to interact with the customers to determine what they want and by when they want it. Other project personnel design the system, and still others write or test the programs. Key project activities are likely to include

1. requirements analysis
2. system design
3. program design
4. program implementation
5. testing
6. training
7. maintenance
8. quality assurance

However, not every task is performed by the same person or group; the assignment of staff to tasks depends on project size, staff expertise, and staff experience. There is great advantage in assigning different responsibilities to different sets of people, offering "checks and balances" that can identify faults early in the development process. For example, suppose the test team is separate from those who design and code the system. Testing new or modified software involves a system test, where the developers demonstrate to the customer that the system works as specified. The test team must define and document the way in which this test will be conducted and the criteria for linking the demonstrated functionality and performance characteristics to the requirements specified by the customer. The test team can generate its test plan from the requirements documents without knowing how the internal pieces of the system are put together. Because the test team has no preconceptions about how the hardware and software will work, it can concentrate on system functionality. This approach makes it easier for the test team to catch errors and omissions made by the designers or programmers. It is in part for this reason that the cleanroom method is organized to use an independent test team, as we will see in later chapters (Mills, Dyer, and Linger 1987).

For similar reasons, it is useful for program designers to be different from system designers. Program designers become deeply involved with the details of the code, and they sometimes neglect the larger picture of how the system should work. We will see in later chapters that techniques such as walkthroughs, inspections, and reviews can bring the two types of designers together to double-check the design before it goes on to be coded, as well as to provide continuity in the development process.

We saw in Chapter 1 that there are many other roles for personnel on the development or maintenance team. As we study each of the major tasks of development in subsequent chapters, we will describe the project team members who perform those tasks.

Once we have decided on the roles of project team members, we must decide which kinds of people we need in each role. Project personnel may differ in many ways, and it is not enough to say that a project needs an analyst, two designers, and five programmers, for example. Two people with the same job title may differ in at least one of the following ways:

- ability to perform the work
- interest in the work
- experience with similar applications
- experience with similar tools or languages
- experience with similar techniques
- experience with similar development environment
- training
- ability to communicate with others
- ability to share responsibility with others
- management skills

Each of these characteristics can affect an individual's ability to perform productively. These variations help to explain why one programmer can write a particular routine in a day, whereas another requires a week. The differences can be critical, not only to schedule estimation, but also to the success of the project.

To understand each worker's performance, we must know his or her ability to perform the work at hand. Some are good at viewing "the big picture," but may not enjoy focusing on detail if asked to work on a small part of a large project. Such people may be better suited to system design or testing than to program design or coding. Sometimes, ability is related to comfort. In classes or on projects, you may have worked with people who are more comfortable programming in one language than another. Indeed, some developers feel more confident about their design abilities than their coding prowess. This feeling of comfort is important; people are usually more productive when they have confidence in their ability to perform.

Interest in the work can also determine someone's success on a project. Although very good at doing a particular job, an employee may be more interested in trying something new than in repeating something done many times before. Thus, the novelty of the work is sometimes a factor in generating interest in it. On the other hand, there are always people who prefer doing what they know and do best, rather than venturing

into new territory. It is important that whoever is chosen for a task be excited about performing it, no matter what the reason.

Given equal ability and interest, two people may still differ in the amount of experience or training they have had with similar applications, tools, or techniques. The person who has already been successful at using C to write a communications controller is more likely to write another communications controller in C faster (but not necessarily more clearly or efficiently) than someone who has neither experience with C nor knowledge of what a communications controller does. Thus, selection of project personnel involves not only individual ability and skill, but also experience and training.

On every software development or maintenance project, members of the development team communicate with one another, with users, and with the customer. The project's progress is affected not only by the degree of communication, but also by the ability of individuals to communicate their ideas. Software failures can result from a breakdown in communication and understanding, so the number of people who need to communicate with one another can affect the quality of the resulting product. Figure 3.9 shows us how quickly the lines of communication can grow. Increasing a work team from two to three people triples the number of possible lines of communication. In general, if a project has n workers, then there are $n(n-1)/2$ pairs of people who might need to communicate, and $2^n - 1$ possible teams that can be created to work on smaller pieces of the project. Thus, a project involving only 10 people can use 45 lines of communication, and there are 1023 possible committees or teams that can be formed to handle subsystem development!

Many projects involve several people who must share responsibility for completing one or more activities. Those working on one aspect of project development must trust other team members to do their parts. In classes, you are usually in total control of the projects you do. You begin with the requirements (usually prescribed by your instructor), design a solution to the problem, outline the code, write the actual lines of code, and test the resulting programs. However, when working in a team, either in school or for an employer or customer, you must be able to share the workload. Not only does this require verbal communication of ideas and results, but it also requires written documentation of what you plan to do and what you have done. You must

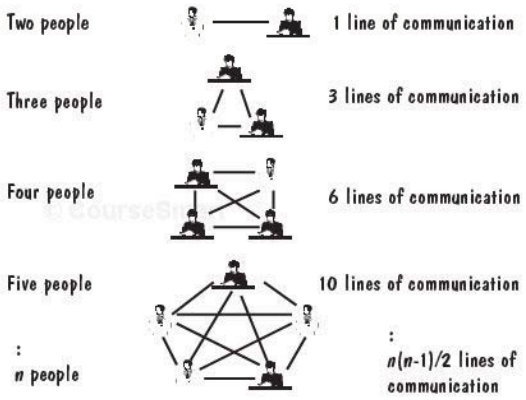


FIGURE 3.9 Communication paths on a project.

accept the results of others without redoing their work. Many people have difficulty in sharing control in this way.

Control is an issue in managing the project, too. Some people are good at directing the work of others. This aspect of personnel interaction is also related to the comfort people feel with the jobs they have. Those who feel uncomfortable with the idea of pushing their colleagues to stay on schedule, to document their code, or to meet with the customer are not good candidates for development jobs involving the management of other workers.

Thus, several aspects of a worker's background can affect the quality of the project team. A project manager should know each person's interests and abilities when choosing who will work together. Sidebar 3.1 explains how meetings and their organization can enhance or impede project progress. As we will see later in this chapter, employee background and communication can also have dramatic effects on the project's cost and schedule.

SIDEBAR 3.1 MAKE MEETINGS ENHANCE PROJECT PROGRESS

Some of the communication on a software project takes place in meetings, either in person or as teleconferences or electronic conversations. However, meetings may take up a great deal of time without accomplishing much. Dressler (1995) tells us that "running bad meetings can be expensive . . . a meeting of eight people who earn $40,000 a year could cost $320 an hour, including salary and benefit costs. That's nearly $6 a minute." Common complaints about meetings include

- The purpose of the meeting is unclear.
- The attendees are unprepared.
- Essential people are absent or late.
- The conversation veers away from its purpose.
- Some meeting participants do not discuss substantive issues. Instead, they argue, dominate the conversation, or do not participate.
- Decisions made at the meeting are never enacted afterward.

Good project management involves planning all software development activities, including meetings. There are several ways to ensure that a meeting is productive. First, the manager should make clear to others on the project team who should be at the meeting, when it will start and end, and what the meeting will accomplish. Second, every meeting should have a written agenda, distributed in advance if possible. Third, someone should take responsibility for keeping discussion on track and for resolving conflicts. Fourth, someone should be responsible for ensuring that each action item decided at the meeting is actually put into practice. Most importantly, minimize the number of meetings, as well as the number of people who must attend them.

Work Styles

Different people have different preferred styles for interacting with others on the job and for understanding problems that arise in the course of their work. For example, you may prefer to do a detailed analysis of all possible information before making a decision, whereas your colleague may rely on "gut feeling" for most of his important decisions. You can think of your preferred work style in terms of two components: the way in which your thoughts are communicated and ideas gathered, and the degree to which your emotions affect decision making. When communicating ideas, some people tell others their thoughts, and some people ask for suggestions from others before forming an opinion. Jung (1959) calls the former **extroverts** and the latter **introverts**. Clearly, your communication style affects the way you interact with others on a project. Similarly, **intuitive** people base their decisions on feelings about and emotional reactions to a problem. Others are **rational**, deciding primarily by examining the facts and carefully considering all options.

We can describe the variety of work styles by considering the graph of Figure 3.10, where communication style forms the horizontal axis and decision style the vertical one. The more extroverted you are, the farther to the right your work style falls on the graph. Similarly, the more emotions play a part in your decisions, the higher up you go. Thus, we can define four basic work styles, corresponding to the four quadrants of the graph. The **rational extroverts** tend to assert their ideas and not let "gut feeling" affect their decision making. They tell their colleagues what they want them to know, but they rarely ask for more information before doing so. When reasoning, they rely on logic, not emotion. The **rational introverts** also avoid emotional decisions, but they are willing to take time to consider all possible courses of action. Rational introverts are information gatherers; they do not feel comfortable making a decision unless they are convinced that all the facts are at hand.

In contrast, **intuitive extroverts** base many decisions on emotional reactions, tending to want to tell others about them rather than ask for input. They use their intuition to be creative, and they often suggest unusual approaches to solving a problem. The **intuitive introvert** is creative, too, but applies creativity only after having gathered

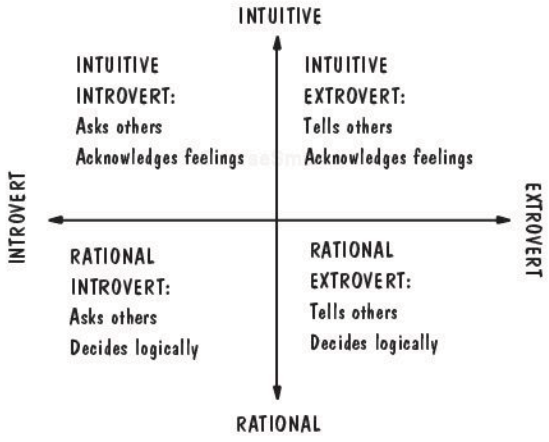


FIGURE 3.10 Work styles.

sufficient information on which to base a decision. Winston Churchill was an intuitive introvert; when he wanted to learn about an issue, he read every bit of material available that addressed it. He often made his decisions based on how he felt about what he had learned (Manchester 1983).

To see how work styles affect interactions on a project, consider several typical staff profiles. Kai, a rational extrovert, judges her colleagues by the results they produce. When making a decision, her top priority is efficiency. Thus, she wants to know only the bottom line. She examines her options and their probable effects, but she does not need to see documents or hear explanations supporting each option. If her time is wasted or her efficiency is hampered in some way, she asserts her authority to regain control of the situation. Thus, Kai is good at making sound decisions quickly.

Marcel, a rational introvert, is very different from his colleague Kai. He judges his peers by how busy they are, and he has little tolerance for those who appear not to be working hard all the time. He is a good worker, admired for the energy he devotes to his work. His reputation as a good worker is very important to him, and he prides himself on being accurate and thorough. He does not like to make decisions without complete information. When asked to make a presentation, Marcel does so only after gathering all relevant information on the subject.

Marcel shares an office with David, an intuitive extrovert. Whereas Marcel will not make a decision without complete knowledge of the situation, David prefers to follow his feelings. Often, he will trust his intuition about a problem, basing his decision on professional judgment rather than a slow, careful analysis of the information at hand. Since he is assertive, David tends to tell the others on his project about his new ideas. He is creative, and he enjoys when others recognize his ideas. David likes to work in an environment where there is a great deal of interaction among the staff members.

Ying, an intuitive introvert, also thrives on her colleagues' attention. She is sensitive and aware of her emotional reactions to people and problems; it is very important that she be liked by her peers. Because she is a good listener, Ying is the project member to whom others turn to express their feelings. Ying takes a lot of time to make a decision, not only because she needs complete information, but also because she wants to make the right decision. She is sensitive to what others think about her ability and ideas. She analyzes situations much as Marcel does, but with a different focus; Marcel looks at all the facts and figures, but Ying examines relational dependencies and emotional involvements, too.

Clearly, not everyone fits neatly into one of the four categories. Different people have different tendencies, and we can use the framework of Figure 3.10 to describe those tendencies and preferences.

Communication is critical to project success, and work style determines communication style. For example, if you are responsible for a part of the project that is behind schedule, Kai and David are likely to tell you when your work must be ready. David may offer several ideas to get the work back on track, and Kai will give you a new schedule to follow. However, Marcel and Ying will probably ask when the results will be ready. Marcel, in analyzing his options, will want to know why it is not ready; Ying will ask if there is anything she can do to help.

Understanding work styles can help you to be flexible in your approach to other project team members and to customers and users. In particular, work styles give you

information about the priorities of others. If a colleague's priorities and interests are different from yours, you can present information to her in terms of what she deems important. For example, suppose Claude is your customer and you are preparing a presentation for him on the status of the project. If Claude is an introvert, you know that he prefers gathering information to giving it. Thus, you may organize your presentation so that it tells him a great deal about how the project is structured and how it is progressing. However, if Claude is an extrovert, you can include questions to allow him to tell you what he wants or needs. Similarly, if Claude is intuitive, you can take advantage of his creativity by soliciting new ideas from him; if he is rational, your presentation can include facts or figures rather than judgments or feelings. Thus, work styles affect interactions among customers, developers, and users.

Work styles can also involve choice of worker for a given task. For instance, intuitive employees may prefer design and development (requiring new ideas) to maintenance programming and design (requiring attention to detail and analysis of complex results).

Project Organization

Software development and maintenance project teams do not consist of people working independently or without coordination. Instead, team members are organized in ways that enhance the swift completion of quality products. The choice of an appropriate structure for your project depends on several things:

- the backgrounds and work styles of the team members
- the number of people on the team
- the management styles of the customers and developers

Good project managers are aware of these issues, and they seek team members who are flexible enough to interact with all players, regardless of work style.

One popular organizational structure is the chief programmer team, first used at IBM (Baker 1972). On a **chief programmer team**, one person is totally responsible for a system's design and development. All other team members report to the chief programmer, who has the final say on every decision. The chief programmer supervises all others, designs all programs, and assigns the code development to the other team members. Assisting the chief is an understudy, whose principal job is substituting for the chief programmer when necessary. A librarian assists the team, responsible for maintaining all project documentation. The librarian also compiles and links the code, and performs preliminary testing of all modules submitted to the library. This division of labor allows the programmers to focus on what they do best: programming.

The organization of the chief programmer team is illustrated in Figure 3.11. By placing all responsibility for all decisions with the chief programmer, the team structure minimizes the amount of communication needed during the project. Each team member must communicate often with the chief, but not necessarily with other team members. Thus, if the team consists of $n - 1$ programmers plus the chief, the team can establish only $n - 1$ paths of communication (one path for each team member's interaction with the chief) out of a potential $n(n - 1)/2$ paths. For example, rather than working out a problem themselves, the programmers can simply approach the chief for an answer. Similarly, the chief reviews all design and code, removing the need for peer reviews.

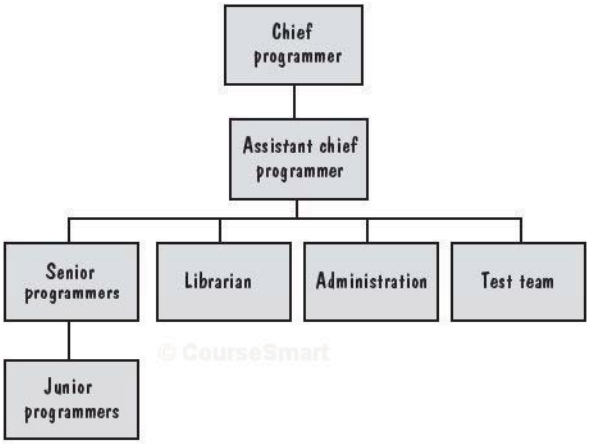


FIGURE 3.11 Chief programmer team organization.

Although a chief programmer team is a hierarchy, groups of workers may be formed to accomplish a specialized task. For instance, one or more team members may form an administrative group to provide a status report on the project's current cost and schedule.

Clearly, the chief programmer must be good at making decisions quickly, so the chief is likely to be an extrovert. However, if most of the team members are introverts, the chief programmer team may not be the best structure for the project. An alternative is based on the idea of "egoless" programming, as described by Weinberg (1971). Instead of a single point of responsibility, an **egoless approach** holds everyone equally responsible. Moreover, the process is separated from the individuals; criticism is made of the product or the result, not the people involved. The egoless team structure is democratic, and all team members vote on a decision, whether it concerns design considerations or testing techniques.

Of course, there are many other ways to organize a development or maintenance project, and the two described above represent extremes. Which structure is preferable? The more people on the project, the more need there is for a formal structure. Certainly, a development team with only three or four members does not always need an elaborate organizational structure. However, a team of several dozen workers must have a well-defined organization. In fact, your company or your customer may impose a structure on the development team, based on past success, on the need to track progress in a certain way, or on the desire to minimize points of contact. For example, your customer may insist that the test team be totally independent of program design and development.

Researchers continue to investigate how project team structure affects the resulting product and how to choose the most appropriate organization in a given situation. A National Science Foundation (1983) investigation found that projects with a high degree of certainty, stability, uniformity, and repetition can be accomplished more effectively by a hierarchical organizational structure such as the chief programmer team. These projects require little communication among project members, so they are well-suited to an organization that stresses rules, specialization, formality, and a clear definition of organizational hierarchy.

TABLE 3.5 Comparison of Organizational Structures

Highly Structured	*Loosely Structured*
High certainty	Uncertainty
Repetition	New techniques or technology
Large projects	Small projects

On the other hand, when there is much uncertainty involved in a project, a more democratic approach may be better. For example, if the requirements may change as development proceeds, the project has a degree of uncertainty. Likewise, suppose your customer is building a new piece of hardware to interface with a system; if the exact specification of the hardware is not yet known, then the level of uncertainty is high. Here, participation in decision making, a loosely defined hierarchy, and the encouragement of open communication can be effective.

Table 3.5 summarizes the characteristics of projects and the suggested organizational structure to address them. A large project with high certainty and repetition probably needs a highly structured organization, whereas a small project with new techniques and a high degree of certainty needs a looser structure. Sidebar 3.2 describes the need to balance structure with creativity.

SIDEBAR 3.2 STRUCTURE VS. CREATIVITY

Kunde (1997) reports the results of experiments by Sally Philipp, a developer of software training materials. When Philipp teaches a management seminar, she divides her class into two groups. Each group is assigned the same task: to build a hotel with construction paper and glue. Some teams are structured, and the team members have clearly defined responsibilities. Others are left alone, given no direction or structure other than to build the hotel. Philipp claims that the results are always the same. "The unstructured teams always do incredibly creative, multistoried Taj Mahals and never complete one on time. The structured teams do a Day's Inn [a bland but functional small hotel], but they're finished and putting chairs around the pool when I call time," she says.

One way she places structure on a team is by encouraging team members to set deadlines. The overall task is broken into small subtasks, and individual team members are responsible for time estimates. The deadlines help to prevent "scope creep," the injection of unnecessary functionality into a product.

The experts in Kunde's article claim that good project management means finding a balance between structure and creativity. Left to their own devices, the software developers will focus only on functionality and creativity, disregarding deadlines and the scope of the specification. Many software project management experts made similar claims. Unfortunately, much of this information is based on anecdote, not on solid empirical investigation.

The two types of organizational structure can be combined, where appropriate. For instance, programmers may be asked to develop a subsystem on their own, using an egoless approach within a hierarchical structure. Or the test team of a loosely structured project may impose a hierarchical structure on itself and designate one person to be responsible for all major testing decisions.

3.3 EFFORT ESTIMATION

One of the crucial aspects of project planning and management is understanding how much the project is likely to cost. Cost overruns can cause customers to cancel projects, and cost underestimates can force a project team to invest much of its time without financial compensation. As described in Sidebar 3.3, there are many reasons for inaccurate estimates. A good cost estimate early in the project's life helps the project manager to know how many developers will be required and to arrange for the appropriate staff to be available when they are needed.

The project budget pays for several types of costs: facilities, staff, methods, and tools. The facilities costs include hardware, space, furniture, telephones, modems, heating and air conditioning, cables, disks, paper, pens, photocopiers, and all other items that provide the physical environment in which the developers will work. For some projects, this environment may already exist, so the costs are well-understood and easy to estimate. But for other projects, the environment may have to be created. For example, a new project may require a security vault, a raised floor, temperature or humidity controls, or special furniture. Here, the costs can be estimated, but they may vary from initial estimates as the environment is built or changed. For instance, installing cabling in a building may seem straightforward until the builders discover that the building is of special historical significance, so that the cables must be routed around the walls instead of through them.

There are sometimes hidden costs that are not apparent to the managers and developers. For example, studies indicate that a programmer needs a minimum amount of space and quiet to be able to work effectively. McCue (1978) reported to his colleagues at IBM that the minimum standard for programmer work space should be 100 square feet of dedicated floor space with 30 square feet of horizontal work surface. The space also needs a floor-to-ceiling enclosure for noise protection. DeMarco and Lister's (1987) work suggests that programmers free from telephone calls and uninvited visitors are more efficient and produce a better product than those who are subject to repeated interruption.

Other project costs involve purchasing software and tools to support development efforts. In addition to tools for designing and coding the system, the project may buy software to capture requirements, organize documentation, test the code, keep track of changes, generate test data, support group meetings, and more. These tools, sometimes called **Computer-Aided Software Engineering** (or **CASE**) **tools**, are sometimes required by the customer or are part of a company's standard software development process.

For most projects, the biggest component of cost is effort. We must determine how many staff-days of effort will be required to complete the project. Effort is certainly

SIDEBAR 3.3 CAUSES OF INACCURATE ESTIMATES

Lederer and Prasad (1992) investigated the cost-estimation practices of 115 different organizations. Thirty-five percent of the managers surveyed on a five-point Likert scale indicated that their current estimates were "moderately unsatisfactory" or "very unsatisfactory." The key causes identified by the respondents included

- frequent requests for changes by users
- overlooked tasks
- users' lack of understanding of their own requirements
- insufficient analysis when developing an estimate
- lack of coordination of systems development, technical services, operations, data administration, and other functions during development
- lack of an adequate method or guidelines for estimating

Several aspects of the project were noted as key influences on the estimate:

- complexity of the proposed application system
- required integration with existing systems
- complexity of the programs in the system
- size of the system expressed as number of functions or programs
- capabilities of the project team members
- project team's experience with the application
- anticipated frequency or extent of potential changes in user requirements
- project team's experience with the programming language
- database management system
- number of project team members
- extent of programming or documentation standards
- availability of tools such as application generators
- team's experience with the hardware

the cost component with the greatest degree of uncertainty. We have seen how work style, project organization, ability, interest, experience, training, and other employee characteristics can affect the time it takes to complete a task. Moreover, when a group of workers must communicate and consult with one another, the effort needed is increased by the time required for meetings, documentation, and training.

Cost, schedule, and effort estimation must be done as early as possible during the project's life cycle, since it affects resource allocation and project feasibility. (If it costs too much, the customer may cancel the project.) But estimation should be done repeatedly throughout the life cycle; as aspects of the project change, the estimate can

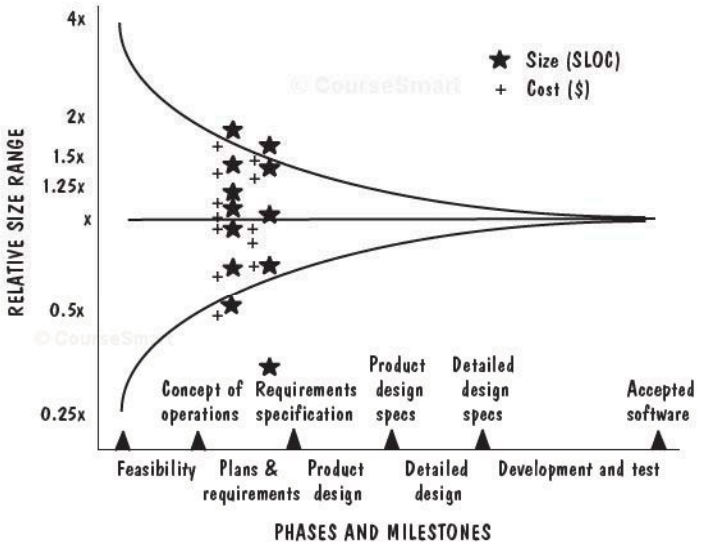


FIGURE 3.12 Changes in estimation accuracy as project progresses (Boehm et al. 1995).

be refined, based on more complete information about the project's characteristics. Figure 3.12 illustrates how uncertainty early in the project can affect the accuracy of cost and size estimates (Boehm et al. 1995).

The stars represent size estimates from actual projects, and the pluses are cost estimates. The funnel-shaped lines narrowing to the right represent Boehm's sense of how our estimates get more accurate as we learn more about a project. Notice that when the specifics of the project are not yet known, the estimate can differ from the eventual actual cost by a factor of 4. As decisions are made about the product and the process, the factor decreases. Many experts aim for estimates that are within 10 percent of the actual value, but Boehm's data indicate that such estimates typically occur only when the project is almost done—too late to be useful for project management.

To address the need for producing accurate estimates, software engineers have developed techniques for capturing the relationships among effort and staff characteristics, project requirements, and other factors that can affect the time, effort, and cost of developing a software system. For the rest of this chapter, we focus on effort-estimation techniques.

Expert Judgment

Many effort-estimation methods rely on expert judgment. Some are informal techniques, based on a manager's experience with similar projects. Thus, the accuracy of the prediction is based on the competence, experience, objectivity, and perception of the estimator. In its simplest form, such an estimate makes an educated guess about the effort needed to build an entire system or its subsystems. The complete estimate can be computed from either a top-down or a bottom-up analysis of what is needed.

Many times analogies are used to estimate effort. If we have already built a system much like the one proposed, then we can use the similarity as the basis for our estimates. For example, if system A is similar to system B, then the cost to produce system A should be very much like the cost to produce B. We can extend the analogy to say that if A is about half the size or complexity of B, then A should cost about half as much as B.

The analogy process can be formalized by asking several experts to make three predictions: a pessimistic one (x), an optimistic one (z), and a most likely guess (y). Then our estimate is the mean of the beta probability distribution determined by these numbers: $(x + 4y + z)/6$. By using this technique, we produce an estimate that "normalizes" the individual estimates.

The Delphi technique makes use of expert judgment in a different way. Experts are asked to make individual predictions secretly, based on their expertise and using whatever process they choose. Then, the average estimate is calculated and presented to the group. Each expert has the opportunity to revise his or her estimate, if desired. The process is repeated until no expert wants to revise. Some users of the Delphi technique discuss the average before new estimates are made; at other times, the users allow no discussion. And in another variation, the justifications of each expert are circulated anonymously among the experts.

Wolverton (1974) built one of the first models of software development effort. His software cost matrix captures his experience with project cost at TRW, a U.S. software development company. As shown in Table 3.6, the row name represents the type of software, and the column designates its difficulty. Difficulty depends on two factors: whether the problem is old (O) or new (N) and whether it is easy (E), moderate (M), or hard (H). The matrix elements are the cost per line of code, as calibrated from historical data at TRW. To use the matrix, you partition the proposed software system into modules. Then, you estimate the size of each module in terms of lines of code. Using the matrix, you calculate the cost per module, and then sum over all the modules. For instance, suppose you have a system with three modules: one input/output module that is old and easy, one algorithm module that is new and hard, and one data management module that is old and moderate. If the modules are likely to have 100, 200, and 100 lines of code, respectively, then the Wolverton model estimates the cost to be $(100 \times 17) + (200 \times 35) + (100 \times 31) = \$11{,}800$.

TABLE 3.6 Wolverton Model Cost Matrix

	Difficulty					
Type of software	OE	OM	OH	NE	NM	NH
Control	21	27	30	33	40	49
Input/output	17	24	27	28	35	43
Pre/post processor	16	23	26	28	34	42
Algorithm	15	20	22	25	30	35
Data management	24	31	35	37	46	57
Time-critical	75	75	75	75	75	75

Since the model is based on TRW data and uses 1974 dollars, it is not applicable to today's software development projects. But the technique is useful and can be transported easily to your own development or maintenance environment.

In general, experiential models, by relying mostly on expert judgment, are subject to all its inaccuracies. They rely on the expert's ability to determine which projects are similar and in what ways. However, projects that appear to be very similar can in fact be quite different. For example, fast runners today can run a mile in 4 minutes. A marathon race requires a runner to run 26 miles and 365 yards. If we extrapolate the 4-minute time, we might expect a runner to run a marathon in 1 hour and 45 minutes. Yet a marathon has never been run in under 2 hours. Consequently, there must be characteristics of running a marathon that are very different from those of running a mile. Likewise, there are often characteristics of one project that make it very different from another project, but the characteristics are not always apparent.

Even when we know how one project differs from another, we do not always know how the differences affect the cost. A proportional strategy is unreliable, because project costs are not always linear: Two people cannot produce code twice as fast as one. Extra time may be needed for communication and coordination, or to accommodate differences in interest, ability, and experience. Sackman, Erikson, and Grant (1968) found that the productivity ratio between best and worst programmers averaged 10 to 1, with no easily definable relationship between experience and performance. Likewise, a more recent study by Hughes (1996) found great variety in the way software is designed and developed, so a model that may work in one organization may not apply to another. Hughes also noted that past experience and knowledge of available resources are major factors in determining cost.

Expert judgment suffers not only from variability and subjectivity, but also from dependence on current data. The data on which an expert judgment model is based must reflect current practices, so they must be updated often. Moreover, most expert judgment techniques are simplistic, neglecting to incorporate a large number of factors that can affect the effort needed on a project. For this reason, practitioners and researchers have turned to algorithmic methods to estimate effort.

Algorithmic Methods

Researchers have created models that express the relationship between effort and the factors that influence it. The models are usually described using equations, where effort is the dependent variable, and several factors (such as experience, size, and application type) are the independent variables. Most of these models acknowledge that project size is the most influential factor in the equation by expressing effort as

$$E = (a + bS^c)m(\mathbf{X})$$

where S is the estimated size of the system, and a, b, and c are constants. $\mathbf{X}$ is a vector of cost factors, x_1 through x_n, and m is an adjustment multiplier based on these factors. In other words, the effort is determined mostly by the size of the proposed system, adjusted by the effects of several other project, process, product, or resource characteristics.

Walston and Felix (1977) developed one of the first models of this type, finding that IBM data from 60 projects yielded an equation of the form

$$E = 5.25S^{0.91}$$

The projects that supplied data built systems with sizes ranging from 4000 to 467,000 lines of code, written in 28 different high-level languages on 66 computers, and representing from 12 to 11,758 person-months of effort. Size was measured as lines of code, including comments as long as they did not exceed 50 percent of the total lines in the program.

The basic equation was supplemented with a productivity index that reflected 29 factors that can affect productivity, shown in Table 3.7. Notice that the factors are tied to a very specific type of development, including two platforms: an operational computer and a development computer. The model reflects the particular development style of the IBM Federal Systems organizations that provided the data.

TABLE 3.7 Walston and Felix Model Productivity Factors

1. Customer interface complexity	16. Use of design and code inspections
2. User participation in requirements definition	17. Use of top-down development
3. Customer-originated program design changes	18. Use of a chief programmer team
4. Customer experience with the application area	19. Overall complexity of code
5. Overall personnel experience	20. Complexity of application processing
6. Percentage of development programmers who participated in the design of functional specifications	21. Complexity of program flow
7. Previous experience with the operational computer	22. Overall constraints on program's design
8. Previous experience with the programming language	23. Design constraints on the program's main storage
9. Previous experience with applications of similar size and complexity	24. Design constraints on the program's timing
10. Ratio of average staff size to project duration (people per month)	25. Code for real-time or interactive operation or for execution under severe time constraints
11. Hardware under concurrent development	26. Percentage of code for delivery
12. Access to development computer open under special request	27. Code classified as nonmathematical application and input/output formatting programs
13. Access to development computer closed	28. Number of classes of items in the database per 1000 lines of code
14. Classified security environment for computer and at least 25% of programs and data	29. Number of pages of delivered documentation per 1000 lines of code
15. Use of structured programming	

Each of the 29 factors was weighted by 1 if the factor increases productivity, 0 if it has no effect on productivity, and −1 if it decreases productivity. A weighted sum of the 29 factors was then used to generate an effort estimate from the basic equation.

Bailey and Basili (1981) suggested a modeling technique, called a meta-model, for building an estimation equation that reflects your own organization's characteristics. They demonstrated their technique using a database of 18 scientific projects written in Fortran at NASA's Goddard Space Flight Center. First, they minimized the standard error estimate and produced an equation that was very accurate:

$$E = 5.5 + 0.73S^{1.16}$$

Then, they adjusted this initial estimate based on the ratio of errors. If R is the ratio between the actual effort, E, and the predicted effort, E', then the effort adjustment is defined as

$$ER_{adj} = \begin{cases} R - 1 & \text{if } R \geq 1 \\ 1 - 1/R & \text{if } R < 1 \end{cases}$$

They then adjusted the initial effort estimate E_{adj} this way:

$$E_{adj} = \begin{cases} (1 + ER_{adj})E & \text{if } R \geq 1 \\ E/(1 + ER_{adj}) & \text{if } R < 1 \end{cases}$$

Finally, Bailey and Basili (1981) accounted for other factors that affect effort, shown in Table 3.8. For each entry in the table, the project is scored from 0 (not present) to 5 (very important), depending on the judgment of the project manager. Thus, the total

TABLE 3.8 Bailey–Basili Effort Modifiers

Total Methodology (METH)	*Cumulative Complexity* (CPLX)	*Cumulative Experience* (EXP)
Tree charts	Customer interface complexity	Programmer qualifications
Top-down design	Application complexity	Programmer machine experience
Formal documentation	Program flow complexity	Programmer language experience
Chief programmer teams	Internal communication complexity	Programmer application experience
Formal training	Database complexity	Team experience
Formal test plans	External communication complexity	
Design formalisms	Customer-initiated program design changes	
Code reading		
Unit development folders		

score for METH can be as high as 45, for CPLX as high as 35, and for EXP as high as 25. Their model describes a procedure, based on multilinear least-square regression, for using these scores to further modify the effort estimate.

Clearly, one of the problems with models of this type is their dependence on size as a key variable. Estimates are usually required early, well before accurate size information is available, and certainly before the system is expressed as lines of code. So the models simply translate the effort-estimation problem to a size-estimation problem. Boehm's Constructive Cost Model (COCOMO) acknowledges this problem and incorporates three sizing techniques in the latest version, COCOMO II.

Boehm (1981) developed the original COCOMO model in the 1970s, using an extensive database of information from projects at TRW, an American company that built software for many different clients. Considering software development from both an engineering and an economics viewpoint, Boehm used size as the primary determinant of cost and then adjusted the initial estimate using over a dozen cost drivers, including attributes of the staff, the project, the product, and the development environment. In the 1990s, Boehm updated the original COCOMO model, creating COCOMO II to reflect the ways in which software development had matured.

The COCOMO II estimation process reflects three major stages of any development project. Whereas the original COCOMO model used delivered source lines of code as its key input, the new model acknowledges that lines of code are impossible to know early in the development cycle. At stage 1, projects usually build prototypes to resolve high-risk issues involving user interfaces, software and system interaction, performance, or technological maturity. Here, little is known about the likely size of the final product under consideration, so COCOMO II estimates size in what its creators call application points. As we shall see, this technique captures size in terms of high-level effort generators, such as the number of screens and reports, and the number of third-generation language components.

At stage 2, the early design stage, a decision has been made to move forward with development, but the designers must explore alternative architectures and concepts of operation. Again, there is not enough information to support fine-grained effort and duration estimation, but far more is known than at stage 1. For stage 2, COCOMO II employs function points as a size measure. Function points, a technique explored in depth in IFPUG (1994a and b), estimate the functionality captured in the requirements, so they offer a richer system description than application points.

By stage 3, the postarchitecture stage, development has begun, and far more information is known. In this stage, sizing can be done in terms of function points or lines of code, and many cost factors can be estimated with some degree of comfort.

COCOMO II also includes models of reuse, takes into account maintenance and breakage (i.e., the change in requirements over time), and more. As with the original COCOMO, the model includes cost factors to adjust the initial effort estimate. A research group at the University of Southern California is assessing and improving its accuracy.

Let us look at COCOMO II in more detail. The basic model is of the form

$$E = bS^c m(\mathbf{X})$$

where the initial size-based estimate, bS^c, is adjusted by the vector of cost driver information, $m(\mathbf{X})$. Table 3.9 describes the cost drivers at each stage, as well as the use of other models to modify the estimate.

TABLE 3.9 Three Stages of COCOMO II

Model Aspect	Stage 1: Application Composition	Stage 2: Early Design	Stage 3: Postarchitecture
Size	Application points	Function points (FPs) and language	FP and language or source lines of code (SLOC)
Reuse	Implict in model	Equivalent SLOC as function of other variables	Equivalent SLOC as function of other variables
Requirements change	Implicit in model	% change expressed as a cost factor	% change expressed as a cost factor
Maintenance	Application points, annual change traffic (ACT)	Function of ACT, software understanding, unfmiliarity	Function of ACT, software understanding, unfamiliarity
Scale (c) in nominal effort equation	1.0	0.91 to 1.23, depending on precedentedness, conformity, early architecture, risk resolution, team cohesion, and SEI process maturity	0.91 to 1.23, depending on precedentedness, conformity, early architecture, risk resolution, team cohesion, and SEI process maturity
Product cost drivers	None	Complexity, required reusability	Reliability, database size, documentation needs, required reuse, and product complexity
Platform cost drivers	None	Platform difficulty	Execution time constraints, main storage constraints, and virtual machine volatility
Personnel cost drivers	None	Personnel capability and experience	Analyst capability, applications experience, programmer capability, programmer experience, language and tool experience, and personnel continuity
Project cost drivers	None	Required development schedule, development environment	Use of software tools, required development schedule, and multisite development

At stage 1, application points supply the size measure. This size measure is an extension of the object-point approach suggested by Kauffman and Kumar (1993) and productivity data reported by Banker, Kauffman, and Kumar (1992). To compute application points, you first count the number of screens, reports, and third-generation language components that will be involved in the application. It is assumed that these elements are defined in a standard way as part of an integrated computer-aided software engineering environment. Next, you classify each application element as simple, medium, or difficult. Table 3.10 contains guidelines for this classification.

TABLE 3.10 Application Point Complexity Levels

For Screens				For Reports			
	Number and source of data tables				Number and source of data tables		
Number of views contained	Total <4 (<2 servers, <3 clients)	Total <8 (2–3 servers, 3–5 clients)	Total 8+ (>3 servers, >5 clients)	*Number of sections contained*	Total <4 (<2 servers, <3 clients)	Total <8 (2–3 servers, 3–5 clients)	Total 8+ (>3 servers, >5 clients)
<3	Simple	Simple	Medium	0 or 1	Simple	Simple	Medium
3–7	Simple	Medium	Difficult	2 or 3	Simple	Medium	Difficult
8+	Medium	Difficult	Difficult	4+	Medium	Difficult	Difficult

The number to be used for simple, medium, or difficult application points is a complexity weight found in Table 3.11. The weights reflect the relative effort required to implement a report or screen of that complexity level.

Then, you sum the weighted reports and screens to obtain a single application-point number. If r percent of the objects will be reused from previous projects, the number of new application points is calculated to be

$$\text{New application points} = (\text{application points}) \times (100 - r)/100$$

To use this number for effort estimation, you use an adjustment factor, called a productivity rate, based on developer experience and capability, coupled with CASE maturity and capability. For example, if the developer experience and capability are rated low, and the CASE maturity and capability are rated low, then Table 3.12 tells us that the productivity factor is 7, so the number of person-months required is the number of new application points divided by 7. When the developers' experience is low but CASE maturity is high, the productivity estimate is the mean of the two values: 16. Likewise, when a team of developers has experience levels that vary, the productivity estimate can use the mean of the experience and capability weights.

At stage 1, the cost drivers are not applied to this effort estimate. However, at stage 2, the effort estimate, based on a function-point calculation, is adjusted for degree of reuse, requirements change, and maintenance. The scale (i.e., the value for c in the effort equation) had been set to 1.0 in stage 1; for stage 2, the scale ranges from 0.91 to

TABLE 3.11 Complexity Weights for Application Points

Element Type	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component	—	—	10

TABLE 3.12 Productivity Estimate Calculation

Developers' experience and capability	Very low	Low	Nominal	High	Very high
CASE maturity and capability	Very low	Low	Nominal	High	Very high
Productivity factor	4	7	13	25	50

1.23, depending on the degree of novelty of the system, conformity, early architecture and risk resolution, team cohesion, and process maturity.

The cost drivers in stages 2 and 3 are adjustment factors expressed as effort multipliers based on rating your project from "extra low" to "extra high," depending on its characteristics. For example, a development team's experience with an application type is considered to be

- *extra low* if it has fewer than 3 months of experience
- *very low* if it has at least 3 but fewer than 5 months of experience
- *low* if it has at least 5 but fewer than 9 months of experience
- *nominal* if it has at least 9 months but less than one year of experience
- *high* if it has at least 1 year but fewer than 2 years of experience
- *very high* if it has at least 2 years but fewer than 4 years of experience
- *extra high* if it has at least 4 years of experience

Similarly, analyst capability is measured on an ordinal scale based on percentile ranges. For instance, the rating is "very high" if the analyst is in the ninetieth percentile and "nominal" for the fifty-fifth percentile. Correspondingly, COCOMO II assigns an effort multiplier ranging from 1.42 for very low to 0.71 for very high. These multipliers reflect the notion that an analyst with very low capability expends 1.42 times as much effort as a nominal or average analyst, while one with very high capability needs about three-quarters the effort of an average analyst. Similarly, Table 3.13 lists the cost driver categories for tool use, and the multipliers range from 1.17 for very low to 0.78 for very high.

TABLE 3.13 Tool Use Categories

Category	*Meaning*
Very low	Edit, code, debug
Low	Simple front-end, back-end CASE, little integration
Nominal	Basic life-cycle tools, moderately integrated
High	Strong, mature life-cycle tools, moderately integrated
Very high	Strong, mature, proactive life-cycle tools, well-integrated with processes, methods, reuse

Notice that stage 2 of COCOMO II is intended for use during the early stages of design. The set of cost drivers in this stage is smaller than the set used in stage 3, reflecting lesser understanding of the project's parameters at stage 2.

The various components of the COCOMO model are intended to be tailored to fit the characteristics of your own organization. Tools are available that implement COCOMO II and compute the estimates from the project characteristics that you supply. Later in this chapter, we will apply COCOMO to our information system example.

Machine-Learning Methods

In the past, most effort- and cost-modeling techniques have relied on algorithmic methods. That is, researchers have examined data from past projects and generated equations from them that are used to predict effort and cost on future projects. However, some researchers are looking to machine learning for assistance in producing good estimates. For example, neural networks can represent a number of interconnected, interdependent units, so they are a promising tool for representing the various activities involved in producing a software product. In a neural network, each unit (called a neuron and represented by network node) represents an activity; each activity has inputs and outputs. Each unit of the network has associated software that performs an accounting of its inputs, computing a weighted sum; if the sum exceeds a threshold value, the unit produces an output. The output, in turn, becomes input to other related units in the network, until a final output value is produced by the network. The neural network is, in a sense, an extension of the activity graphs we examined earlier in this chapter.

There are many ways for a neural network to produce its outputs. Some techniques involve looking back to what has happened at other nodes; these are called *back-propagation* techniques. They are similar to the method we used with activity graphs to look back and determine the slack on a path. Other techniques look forward, to anticipate what is about to happen.

Neural networks are developed by "training" them with data from past projects. Relevant data are supplied to the network, and the network uses forward and backward algorithms to "learn" by identifying patterns in the data. For example, historical data about past projects might contain information about developer experience; the network may identify relationships between level of experience and the amount of effort required to complete a project.

Figure 3.13 illustrates how Shepperd (1997) used a neural network to produce an effort estimate. There are three layers in the network, and the network has no cycles. The four inputs are factors that can affect effort on a project; the network uses them to produce effort as the single output. To begin, the network is initialized with random weights. Then, new weights, calculated as a "training set" of inputs and outputs based on past history, are fed to the network. The user of the model specifies a training algorithm that explains how the training data are to be used; this algorithm is also based on past history, and it commonly involves back-propagation. Once the network is trained (i.e., once the network values are adjusted to reflect past experience), it can then be used to estimate effort on new projects.

Several researchers have used back-propagation algorithms on similar neural networks to predict development effort, including estimation for projects using

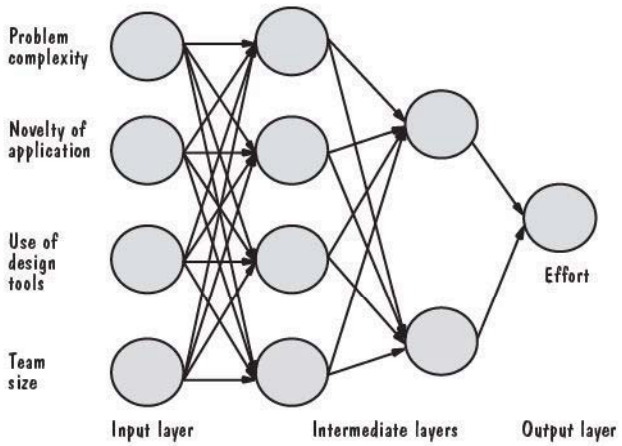


FIGURE 3.13 Shepperd's feed-forward neural network.

fourth-generation languages (Wittig and Finnie 1994; Srinivasan and Fisher 1995; Samson, Ellison, and Dugard 1997). Shepperd (1997) reports that the accuracy of this type of model seems to be sensitive to decisions about the topology of the neural network, the number of learning stages, and the initial random weights of the neurons within the network. The networks also seem to require large training sets in order to give good predictions. In other words, they must be based on a great deal of experience rather than a few representative projects. Data of this type are sometimes difficult to obtain, especially collected consistently and in large quantity, so the paucity of data limits this technique's usefulness. Moreover, users tend to have difficulty understanding neural networks. However, if the technique produces more accurate estimates, organizations may be more willing to collect data for the networks.

In general, this "learning" approach has been tried in different ways by other researchers. Srinivasan and Fisher (1995) used Kemerer's data (Kemerer 1989) with a statistical technique called a regression tree; they produced predictions more accurate than those of the original COCOMO model and SLIM, a proprietary commercial model. However, their results were not as good as those produced by a neural network or a model based on function points. Briand, Basili, and Thomas (1992) obtained better results from using a tree induction technique, using the Kemerer and COCOMO datasets. Porter and Selby (1990) also used a tree-based approach; they constructed a decision tree that identifies which project, process, and product characteristics may be useful in predicting likely effort. They also used the technique to predict which modules are likely to be fault-prone.

A machine-learning technique called *Case-Based Reasoning* (CBR) can be applied to analogy-based estimates. Used by the artificial intelligence community, CBR builds a decision algorithm based on the several combinations of inputs that might be encountered on a project. Like the other techniques described here, CBR requires information about past projects. Shepperd (1997) points out that CBR offers two clear advantages over many of the other techniques. First, CBR deals only with events that actually occur, rather than with the much larger set of all possible occurrences. This

same feature also allows CBR to deal with poorly understood domains. Second, it is easier for users to understand particular cases than to depict events as chains of rules or as neural networks.

Estimation using CBR involves four steps:

1. The user identifies a new problem as a case.
2. The system retrieves similar cases from a repository of historical information.
3. The system reuses knowledge from previous cases.
4. The system suggests a solution for the new case.

The solution may be revised, depending on actual events, and the outcome is placed in the repository, building up the collection of completed cases. However, there are two big hurdles in creating a successful CBR system: characterizing cases and determining similarity.

Cases are characterized based on the information that happens to be available. Usually, experts are asked to supply a list of features that are significant in describing cases and, in particular, in determining when two cases are similar. In practice, similarity is usually measured using an n-dimensional vector of n features. Shepperd, Schofield, and Kitchenham (1996) found a CBR approach to be more accurate than traditional regression analysis-based algorithmic methods.

Finding the Model for Your Situation

There are many effort and cost models being used today: commercial tools based on past experience or intricate models of development, and home-grown tools that access databases of historical information about past projects. Validating these models (i.e., making sure the models reflect actual practice) is difficult, because a large amount of data is needed for the validation exercise. Moreover, if a model is to apply to a large and varied set of situations, the supporting database must include measures from a very large and varied set of development environments.

Even when you find models that are designed for your development environment, you must be able to evaluate which are the most accurate on your projects. There are two statistics that are often used to help you in assessing the accuracy, PRED and MMRE. **PRED(x/100)** is the percentage of projects for which the estimate is within x% of the actual value. For most effort, cost, and schedule models, managers evaluate PRED(0.25), that is, those models whose estimates are within 25% of the actual value; a model is considered to function well if PRED(0.25) is greater than 75%. **MMRE** is the mean magnitude of relative error, so we hope that the MMRE for a particular model is very small. Some researchers consider an MMRE of 0.25 to be fairly good, and Boehm (1981) suggests that MMRE should be 0.10 or less. Table 3.14 lists the best values for PRED and MMRE reported in the literature for a variety of models. As you can see, the statistics for most models are disappointing, indicating that no model appears to have captured the essential characteristics and their relationships for all types of development. However, the relationships among cost factors are not simple, and the models must be flexible enough to handle changing use of tools and methods.

TABLE 3.14 Summary of Model Performance

Model	*PRED(0.25)*	*MMRE*
Walston–Felix	0.30	0.48
Basic COCOMO	0.27	0.60
Intermediate COCOMO	0.63	0.22
Intermediate COCOMO (variation)	0.76	0.19
Bailey–Basili	0.78	0.18
Pfleeger	0.50	0.29
SLIM	0.06–0.24	0.78–1.04
Jensen	0.06–0.33	0.70–1.01
COPMO	0.38–0.63	0.23–5.7
General COPMO	0.78	0.25

Moreover, Kitchenham, MacDonell, Pickard, and Shepperd (2000) point out that the MMRE and PRED statistics are not direct measures of estimation accuracy. They suggest that you use the simple ratio of estimate to actual: estimate/actual. This measure has a distribution that directly reflects estimation accuracy. By contrast, MMRE and PRED are measures of the spread (standard deviation) and peakedness (kurtosis) of the ratio, so they tell us only characteristics of the distribution.

Even when estimation models produce reasonably accurate estimates, we must be able to understand which types of effort are needed during development. For example, designers may not be needed until the requirements analysts have finished developing the specification. Some effort and cost models use formulas based on past experience to apportion the effort across the software development life cycle. For instance, the original COCOMO model suggested effort required by development activity, based on percentages allotted to key process activities. But, as Figure 3.14 illustrates, researchers report conflicting values for these percentages (Brooks 1995; Yourdon 1982). Thus, when you are building your own database to support estimation in your organization, it is important to record not only how much effort is expended on a project, but also who is doing it and for what activity.

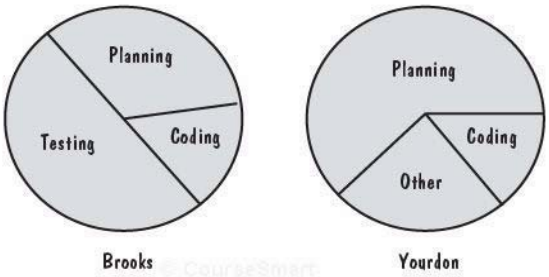


FIGURE 3.14 Different reports of effort distribution.

3.4 RISK MANAGEMENT

As we have seen, many software project managers take steps to ensure that their projects are done on time and within effort and cost constraints. However, project management involves far more than tracking effort and schedule. Managers must determine whether any unwelcome events may occur during development or maintenance and make plans to avoid these events or, if they are inevitable, minimize their negative consequences. A **risk** is an unwanted event that has negative consequences. Project managers must engage in **risk management** to understand and control the risks on their projects.

What Is a Risk?

Many events occur during software development; Sidebar 3.4 lists Boehm's view of some of the riskiest ones. We distinguish risks from other project events by looking for three things (Rook 1993):

1. *A loss associated with the event.* The event must create a situation where something negative happens to the project: a loss of time, quality, money, control, understanding, and so on. For example, if requirements change dramatically after

SIDEBAR 3.4 BOEHM'S TOP TEN RISK ITEMS

Boehm (1991) identifies 10 risk items and recommends risk management techniques to address them.

1. *Personnel shortfalls.* Staffing with top talent; job matching; team building; morale building; cross-training; prescheduling key people.
2. *Unrealistic schedules and budgets.* Detailed multisource cost and schedule estimation; design to cost; incremental development; software reuse; requirements scrubbing.
3. *Developing the wrong software functions.* Organizational analysis; mission analysis; operational concept formulation; user surveys; prototyping; early user's manuals.
4. *Developing the wrong user interface.* Prototyping; scenarios; task analysis.
5. *Gold plating.* Requirements scrubbing; prototyping; cost-benefit analysis; design to cost.
6. *Continuing stream of requirements changes.* High change threshold; information hiding; incremental development (defer changes to later increments).
7. *Shortfalls in externally performed tasks.* Reference checking; preaward audits; award-fee contracts; competitive design or prototyping; team building.
8. *Shortfalls in externally furnished components.* Benchmarking; inspections; reference checking; compatibility analysis.
9. *Real-time performance shortfalls.* Simulation; benchmarking; modeling; prototyping; instrumentation; tuning.
10. *Straining computer science capabilities.* Technical analysis; cost-benefit analysis; prototyping; reference checking.

the design is done, then the project can suffer from loss of control and understanding if the new requirements are for functions or features with which the design team is unfamiliar. And a radical change in requirements is likely to lead to losses of time and money if the design is not flexible enough to be changed quickly and easily. The loss associated with a risk is called the **risk impact**.

2. *The likelihood that the event will occur.* We must have some idea of the probability that the event will occur. For example, suppose a project is being developed on one machine and will be ported to another when the system is fully tested. If the second machine is a new model to be delivered by the vendor, we must estimate the likelihood that it will not be ready on time. The likelihood of the risk, measured from 0 (impossible) to 1 (certainty) is called the **risk probability**. When the risk probability is 1, then the risk is called a **problem**, since it is certain to happen.
3. *The degree to which we can change the outcome.* For each risk, we must determine what we can do to minimize or avoid the impact of the event. **Risk control** involves a set of actions taken to reduce or eliminate a risk. For example, if the requirements may change after design, we can minimize the impact of the change by creating a flexible design. If the second machine is not ready when the software is tested, we may be able to identify other models or brands that have the same functionality and performance and can run our new software until the new model is delivered.

We can quantify the effects of the risks we identify by multiplying the risk impact by the risk probability, to yield the **risk exposure**. For example, if the likelihood that the requirements will change after design is 0.3, and the cost to redesign to new requirements is $50,000, then the risk exposure is $15,000. Clearly, the risk probability can change over time, as can the impact, so part of a project manager's job is to track these values over time and plan for the events accordingly.

There are two major sources of risk: generic risks and project-specific risks. **Generic risks** are those common to all software projects, such as misunderstanding the requirements, losing key personnel, or allowing insufficient time for testing. **Project-specific risks** are threats that result from the particular vulnerabilities of the given project. For example, a vendor may be promising network software by a particular date, but there is some risk that the network software will not be ready on time.

Risk Management Activities

Risk management involves several important steps, each of which is illustrated in Figure 3.15. First, we assess the risks on a project, so that we understand what may occur during the course of development or maintenance. The assessment consists of three activities: identifying the risks, analyzing them, and assigning priorities to each of them. To identify them, we may use many different techniques.

If the system we are building is similar in some way to a system we have built before, we may have a checklist of problems that may occur; we can review the checklist to determine if the new project is likely to be subject to the risks listed. For systems that are new in some way, we may augment the checklist with an analysis of each of the activities in the development cycle; by decomposing the process into small pieces, we may be able to anticipate problems that may arise. For example, we may decide that

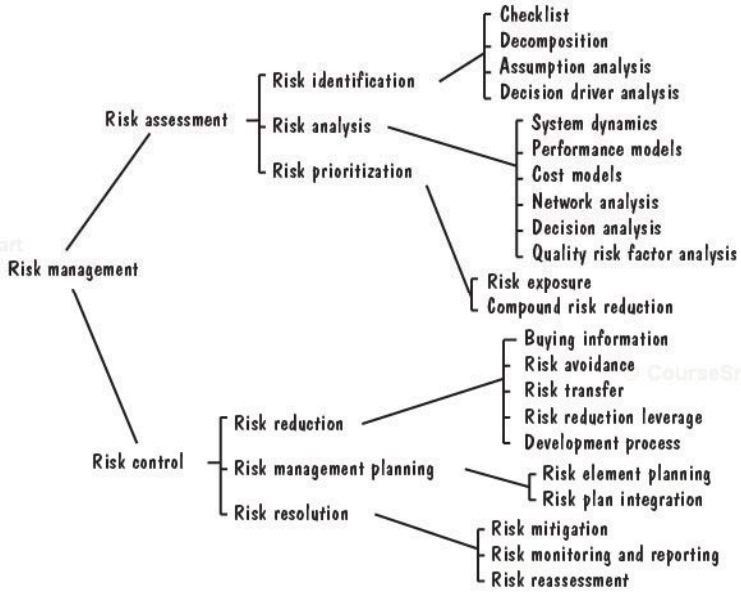


FIGURE 3.15 Steps in risk management (Rook 1993).

there is a risk of the chief designer leaving during the design process. Similarly, we may analyze the assumptions or decisions we make about how the project will be done, who will do it, and with what resources. Then, each assumption is assessed to determine the risks involved.

Finally, we analyze the risks we have identified, so that we can understand as much as possible about when, why, and where they might occur. There are many techniques we can use to enhance our understanding, including system dynamics models, cost models, performance models, network analysis, and more.

Once we have itemized all the risks, we use our understanding to assign priorities them. A priority scheme enables us to devote our limited resources only to the most threatening risks. Usually, priorities are based on the risk exposure, which takes into account not only likely impact, but also the probability of occurrence.

The risk exposure is computed from the risk impact and the risk probability, so we must estimate each of these risk aspects. To see how the quantification is done, consider the analysis depicted in Figure 3.16. Suppose we have analyzed the system development process and we know we are working under tight deadlines for delivery. We have decided to build the system in a series of releases, where each release has more functionality than the one that preceded it. Because the system is designed so that functions are relatively independent, we consider testing only the new functions for a release, and we assume that the existing functions still work as they did before. However, we may worry that there are risks associated with not performing **regression testing**: the assurance that existing functionality still works correctly.

For each possible outcome, we estimate two quantities: the probability of an unwanted outcome, P(UO), and the loss associated with the unwanted outcome,

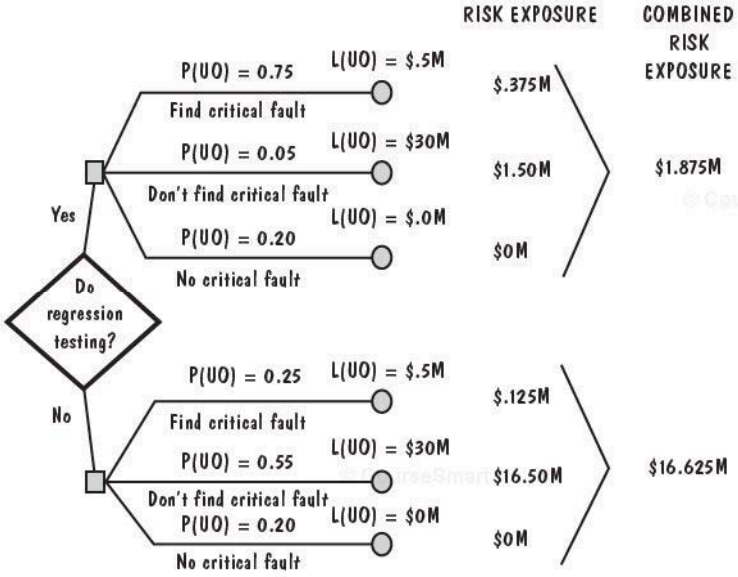


FIGURE 3.16 Example of risk exposure calculation.

L(UO). For instance, there are three possible consequences of performing regression testing: finding a critical fault if one exists, not finding the critical fault (even though it exists), or deciding (correctly) that there is no critical fault. As the figure illustrates, we have estimated the probability of the first case to be 0.75, of the second to be 0.05, and of the third to be 0.20. The loss associated with an unwanted outcome is estimated to be $500,000 if a critical fault is found, so that the risk exposure is $375,000. Similarly, we calculate the risk exposure for the other branches of this decision tree, and we find that our risk exposure if we perform regression testing is almost $2 million. However, the same kind of analysis shows us that the risk exposure if we do not perform regression testing is almost $17 million. Thus, we say (loosely) that more is at risk if we do not perform regression testing.

Risk exposure helps us to list the risks in priority order, with the risks of most concern given the highest priority. Next, we must take steps to control the risks. The notion of control acknowledges that we may not be able to eliminate all risks. Instead, we may be able to minimize the risk or mitigate it by taking action to handle the unwanted outcome in an acceptable way. Therefore, risk control involves risk reduction, risk planning, and risk resolution.

There are three strategies for risk reduction:

- avoiding the risk, by changing requirements for performance or functionality
- transferring the risk, by allocating risks to other systems or by buying insurance to cover any financial loss should the risk become a reality
- assuming the risk, by accepting it and controlling it with the project's resources

To aid decision making about risk reduction, we must take into account the cost of reducing the risk. We call **risk leverage** the difference in risk exposure divided by the cost of reducing the risk. In other words, risk reduction leverage is

$$\frac{(\text{Risk exposure before reduction} - \text{risk exposure after reduction})}{(\text{cost of risk reduction})}$$

If the leverage value is not high enough to justify the action, then we can look for other less costly or more effective reduction techniques.

In some cases, we can choose a development process to help reduce the risk. For example, we saw in Chapter 2 that prototyping can improve understanding of the requirements and design, so selecting a prototyping process can reduce many project risks.

It is useful to record decisions in a **risk management plan**, so that both customer and development team can review how problems are to be avoided, as well as how they are to be handled should they arise. Then, we should monitor the project as development progresses, periodically reevaluating the risks, their probability, and their likely impact.

3.5 THE PROJECT PLAN

To communicate risk analysis and management, project cost estimates, schedule, and organization to our customers, we usually write a document called a **project plan**. The plan puts in writing the customer's needs, as well as what we hope to do to meet them. The customer can refer to the plan for information about activities in the development process, making it easy to follow the project's progress during development. We can also use the plan to confirm with the customer any assumptions we are making, especially about cost and schedule.

A good project plan includes the following items:

1. project scope
2. project schedule
3. project team organization
4. technical description of the proposed system
5. project standards, procedures, and proposed techniques and tools
6. quality assurance plan
7. configuration management plan
8. documentation plan
9. data management plan
10. resource management plan
11. test plan
12. training plan
13. security plan
14. risk management plan
15. maintenance plan

The scope defines the system boundary, explaining what will be included in the system and what will not be included. It assures the customer that we understand what is wanted. The schedule can be expressed using a work breakdown structure, the deliverables, and a timeline to show what will be happening at each point during the project life cycle. A Gantt chart can be useful in illustrating the parallel nature of some of the development tasks.

The project plan also lists the people on the development team, how they are organized, and what they will be doing. As we have seen, not everyone is needed all the time during the project, so the plan usually contains a resource allocation chart to show staffing levels at different times.

Writing a technical description forces us to answer questions and address issues as we anticipate how development will proceed. This description lists hardware and software, including compilers, interfaces, and special-purpose equipment or software. Any special restrictions on cabling, execution time, response time, security, or other aspects of functionality or performance are documented in the plan. The plan also lists any standards or methods that must be used, such as

- algorithms
- tools
- review or inspection techniques
- design languages or representations
- coding languages
- testing techniques

For large projects, it may be appropriate to include a separate quality assurance plan, to describe how reviews, inspections, testing, and other techniques will help to evaluate quality and ensure that it meets the customer's needs. Similarly, large projects need a configuration management plan, especially when there are to be multiple versions and releases of the system. As we will see in Chapter 10, configuration management helps to control multiple copies of the software. The configuration management plan tells the customer how we will track changes to the requirements, design, code, test plans, and documents.

Many documents are produced during development, especially for large projects where information about the design must be made available to project team members. The project plan lists the documents that will be produced, explains who will write them and when, and, in concert with the configuration management plan, describes how documents will be changed.

Because every software system involves data for input, calculation, and output, the project plan must explain how data will be gathered, stored, manipulated, and archived. The plan should also explain how resources will be used. For example, if the hardware configuration includes removable disks, then the resource management part of the project plan should explain what data are on each disk and how the disk packs or diskettes will be allocated and backed up.

Testing requires a great deal of planning to be effective, and the project plan describes the project's overall approach to testing. In particular, the plan should state

how test data will be generated, how each program module will be tested (e.g., by testing all paths or all statements), how program modules will be integrated with each other and tested, how the entire system will be tested, and who will perform each type of testing. Sometimes, systems are produced in stages or phases, and the test plan should explain how each stage will be tested. When new functionality is added to a system in stages, as we saw in Chapter 2, then the test plan must address regression testing, ensuring that the existing functionality still works correctly.

Training classes and documents are usually prepared during development, rather than after the system is complete, so that training can begin as soon as the system is ready (and sometimes before). The project plan explains how training will occur, describing each class, supporting software and documents, and the expertise needed by each student.

When a system has security requirements, a separate security plan is sometimes needed. The security plan addresses the way that the system will protect data, users, and hardware. Since security involves confidentiality, availability, and integrity, the plan must explain how each facet of security affects system development. For example, if access to the system will be limited by using passwords, then the plan must describe who issues and maintains the passwords, who develops the password-handling software, and what the password encryption scheme will be.

Finally, if the project team will maintain the system after it is delivered to the user, the project plan should discuss responsibilities for changing the code, repairing the hardware, and updating supporting documentation and training materials.

3.6 PROCESS MODELS AND PROJECT MANAGEMENT

We have seen how different aspects of a project can affect the effort, cost, and schedule required, as well as the risks involved. Managers most successful at building quality products on time and within budget are those who tailor the project management techniques to the particular characteristics of the resources needed, the chosen process, and the people assigned.

To understand what to do on your next project, it is useful to examine project management techniques used by successful projects from the recent past. In this section, we look at two projects: Digital's Alpha AXP program and the F-16 aircraft software. We also investigate the merging of process and project management.

Enrollment Management

Digital Equipment Corporation spent many years developing its Alpha AXP system, a new system architecture and associated products that formed the largest project in Digital's history. The software portion of the effort involved four operating systems and 22 software engineering groups, whose roles included designing migration tools, network systems, compilers, databases, integration frameworks, and applications. Unlike many other development projects, the major problems with Alpha involved reaching milestones too early! Thus, it is instructive to look at how the project was managed and what effects the management process had on the final product.

During the course of development, the project managers developed a model that incorporated four tenets, called the Enrollment Management model:

1. establishing an appropriately large shared vision
2. delegating completely and eliciting specific commitments from participants
3. inspecting vigorously and providing supportive feedback
4. acknowledging every advance and learning as the program progressed (Conklin 1996)

Figure 3.17 illustrates the model. Vision was used to "enroll" the related programs, so they all shared common goals. Each group or subgroup of the project defined its own objectives in terms of the global ones stated for the project, including the company's business goals. Next, as managers developed plans, they delegated tasks to groups, soliciting comments and commitments about the content of each task and the schedule constraints imposed. Each required result was measurable and identified with a particular owner who was held accountable for delivery. The owner may not have been the person doing the actual work; rather, he or she was the person responsible for getting the work done.

Managers continually inspected the project to make sure that delivery would be on time. Project team members were asked to identify risks, and when a risk threatened to keep the team from meeting its commitments, the project manager declared the project to be a "cusp": a critical event. Such a declaration meant that team members were ready to make substantial changes to help move the project forward. For each project step, the managers acknowledged progress both personally and publicly. They recorded what had been learned and they asked team members how things could be improved the next time.

Coordinating all the hardware and software groups was difficult, and managers realized that they had to oversee both technical and project events. That is, the technical focus involved technical design and strategy, whereas the project focus addressed

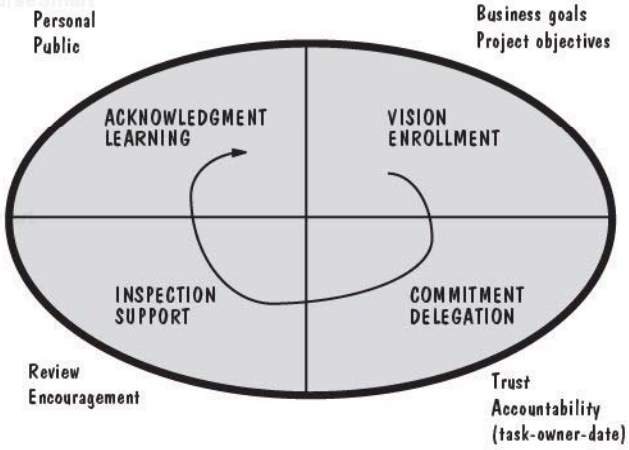


FIGURE 3.17 Enrollment Management model (Conklin 1996).

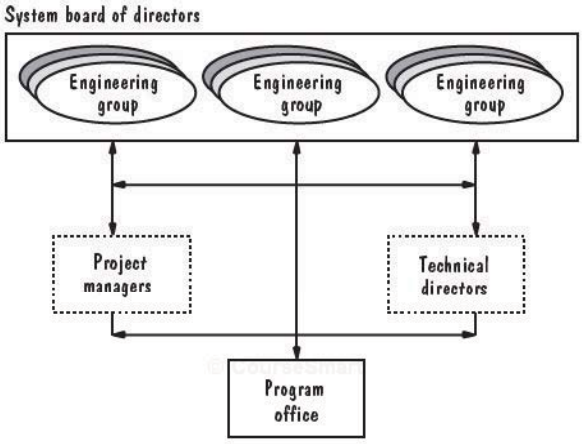


FIGURE 3.18 Alpha project organization (Conklin 1996).

commitments and deliverables. Figure 3.18 illustrates the organization that allowed both foci to contribute to the overall program.

The simplicity of the model and organization does not mean that managing the Alpha program was simple. Several cusps threatened the project and were dealt with in a variety of ways. For example, management was unable to produce an overall plan, and project managers had difficulty coping. At the same time, technical leaders were generating unacceptably large design documents that were difficult to understand. To gain control, the Alpha program managers needed a programwide work plan that illustrated the order in which each contributing task was to be done and how it coordinated with the other tasks. They created a master plan based only on the critical program components—those things that were critical to business success. The plan was restricted to a single page, so that the participants could see the "big picture," without complexity or detail. Similarly, one-page descriptions of designs, schedules, and other key items enabled project participants to have a global picture of what to do, and when and how to do it.

Another cusp occurred when a critical task was announced to be several months behind schedule. The management addressed this problem by instituting regular operational inspections of progress so there would be no more surprises. The inspection involved presentation of a one-page report, itemizing key points about the project:

- schedule
- milestones
- critical path events in the past month
- activities along the critical path in the next month
- issues and dependencies resolved
- issues and dependencies not resolved (with ownership and due dates)

An important aspect of Alpha's success was the managers' realization that engineers are usually motivated more by recognition than by financial gain. Instead of rewarding participants with money, they focused on announcing progress and on making sure that the public knew how much the managers appreciated the engineers' work.

The result of Alpha's flexible and focused management was a program that met its schedule to the month, despite setbacks along the way. Enrollment management enabled small groups to recognize their potential problems early and take steps to handle them while the problems were small and localized. Constancy of purpose was combined with continual learning to produce an exceptional product. Alpha met its performance goals, and its quality was reported to be very high.

Accountability Modeling

The U.S. Air Force and Lockheed Martin formed an Integrated Product Development Team to build a modular software system designed to increase capacity, provide needed functionality, and reduce the cost and schedule of future software changes to the F-16 aircraft. The resulting software included more than four million lines of code, a quarter of which met real-time deadlines in flight. F-16 development also involved building device drivers, real-time extensions to the Ada run-time system, a software engineering workstation network, an Ada compiler for the modular mission computer, software build and configuration management tools, simulation and test software, and interfaces for loading software into the airplane (Parris 1996).

The flight software's capability requirements were well-understood and stable, even though about a million lines of code were expected to be needed from the 250 developers organized as eight product teams, a chief engineer, plus a program manager and staff. However, the familiar capabilities were to be implemented in an unfamiliar way: modular software using Ada and object-oriented design and analysis, plus a transition from mainframes to workstations. Project management constraints included rigid "need dates" and commitment to developing three releases of equal task size, called tapes. The approach was high risk, because the first tape included little time for learning the new methods and tools, including concurrent development (Parris 1996).

Pressure on the project increased because funding levels were cut and schedule deadlines were considered to be extremely unrealistic. In addition, the project was organized in a way unfamiliar to most of the engineers. The participants were used to working in a **matrix organization**, so that each engineer belonged to a functional unit based on a type of skill (such as the design group or the test group) but was assigned to one or more projects as that skill was needed. In other words, an employee could be identified by his or her place in a matrix, with functional skills as one dimension and project names as the other dimension. Decisions were made by the functional unit hierarchy in this traditional organization. However, the contract for the F-16 required the project to be organized as an **integrated product development** team: combining individuals from different functional groups into an interdisciplinary work unit empowered with separate channels of accountability.

To enable the project members to handle the culture change associated with the new organization, the F-16 project used the accountability model shown in Figure 3.19. In the model, a team is any collection of people responsible for producing a given result. A stakeholder is anyone affected by that result or the way in which the result is achieved. The process involves a continuing exchange of accountings (a report of what you have done, are doing, or plan to do) and consequences, with the goal of doing only

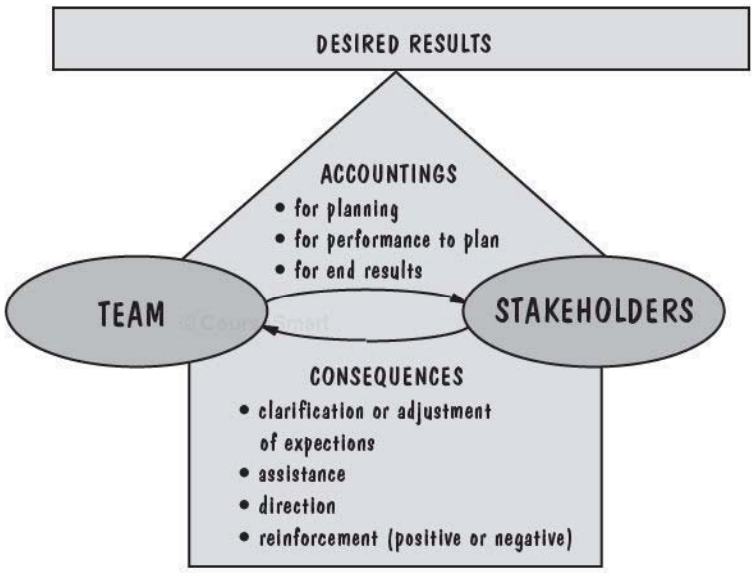


FIGURE 3.19 Accountability model (Parris 1996).

what makes sense for both the team and the stakeholders. The model was applied to the design of management systems and to team operating procedures, replacing independent behaviors with interdependence, emphasizing "being good rather than looking good" (Parris 1996).

As a result, several practices were required, including a weekly, one-hour team status review. To reinforce the notions of responsibility and accountability, each personal action item had explicit closure criteria and was tracked to completion. An action item could be assigned to a team member or a stakeholder, and often involved clarifying issues or requirements, providing missing information or reconciling conflicts.

Because the teams had multiple, overlapping activities, an activity map was used to illustrate progress on each activity in the overall context of the project. Figure 3.20 shows part of an activity map. You can see how each bar represents an activity, and each activity is assigned a method for reporting progress. The point on a bar indicates when detailed planning should be in place to guide activities. The "today" line shows current status, and an activity map was used during the weekly reviews as an overview of the progress to be discussed.

For each activity, progress was tracked using an appropriate evaluation or performance method. Sometimes the method included cost estimation, critical path analysis, or schedule tracking. *Earned value* was used as a common measure for comparing progress on different activities: a scheme for comparing activities determined how much of the project had been completed by each activity. The earned-value calculation included weights to represent what percent of the total process each step constituted, relative to overall effort. Similarly, each component was assigned a size value that

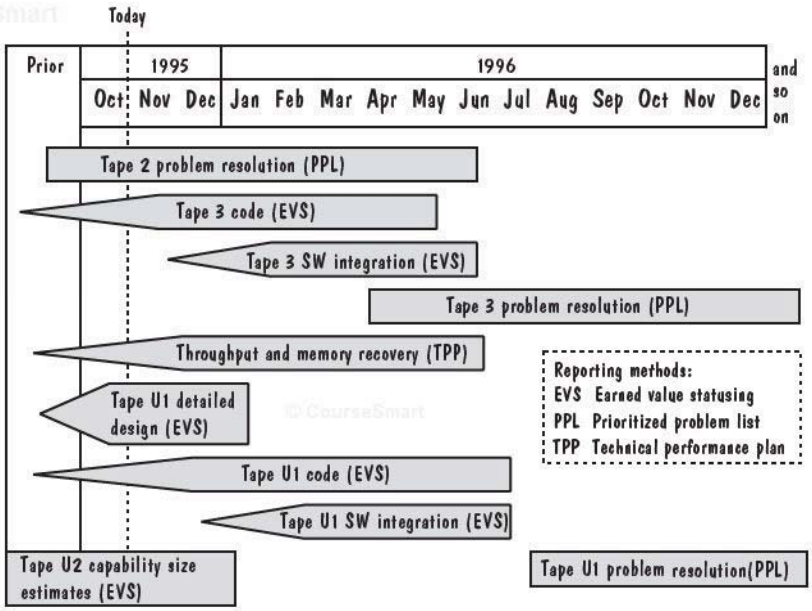


FIGURE 3.20 Sample activity roadmap (adapted from Parris 1996).

represented its proportion of the total product, so that progress relative to the final size could be tracked, too. Then, an earned-value summary chart, similar to Figure 3.21, was presented at each review meeting.

Once part of a product was completed, its progress was no longer tracked. Instead, its performance was tracked, and problems were recorded. Each problem was assigned a priority by the stakeholders, and a snapshot of the top five problems on each product team's list was presented at the weekly review meeting for discussion. The priority lists generated discussion about why the problems occurred, what work-arounds could be put in place, and how similar problems could be prevented in the future.

The project managers found a major problem with the accountability model: it told them nothing about coordination among different teams. As a result, they built software to catalog and track the hand-offs from one team to another, so that every team could understand who was waiting for action or products from them. A model of the hand-offs was used for planning, so that undesirable patterns or scenarios could be eliminated. Thus, an examination of the hand-off model became part of the review process.

It is easy to see how the accountability model, coupled with the hand-off model, addressed several aspects of project management. First, it provided a mechanism for communication and coordination. Second, it encouraged risk management, especially by forcing team members to examine problems in review meetings. And third, it integrated progress reporting with problem solving. Thus, the model actually prescribes a project management process that was followed on the F-16 project.

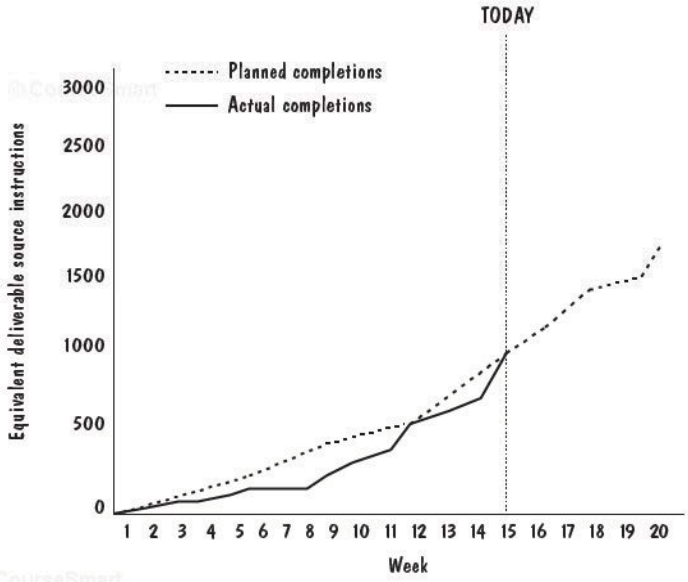


FIGURE 3.21 Example earned-value summary chart (Parris 1996).

Anchoring Milestones

In Chapter 2, we examined many process models that described how the technical activities of software development should progress. Then, in this chapter, we looked at several methods to organize projects to perform those activities. The Alpha AXP and F-16 examples have shown us that project management must be tightly integrated with the development process, not just for tracking progress, but, more importantly, for effective planning and decision making to prevent major problems from derailing the project. Boehm (1996) has identified three milestones common to all software development processes that can serve as a basis for both technical process and project management:

- life-cycle objectives
- life-cycle architecture
- initial operational capability

We can examine each milestone in more detail.

The purpose of the life-cycle objectives milestone is to make sure the stakeholders agree with the system's goals. The key stakeholders act as a team to determine the system boundary, the environment in which the system will operate, and the external systems with which the system must interact. Then, the stakeholders work through scenarios of how the system will be used. The scenarios can be expressed in terms of prototypes, screen layouts, data flows, or other representations, some of which we will learn about in later chapters. If the system is business- or safety-critical, the scenarios should also include instances where the system fails, so that designers can determine how the

system is supposed to react to or even avoid a critical failure. Similarly, other essential features of the system are derived and agreed upon. The result is an initial life-cycle plan that lays out (Boehm 1996):

- *Objectives:* Why is the system being developed?
- *Milestones and schedules:* What will be done by when?
- *Responsibilities:* Who is responsible for a function?
- *Approach:* How will the job be done, technically and managerially?
- *Resources:* How much of each resource is needed?
- *Feasibility:* Can this be done, and is there a good business reason for doing it?

The life-cycle architecture is coordinated with the life-cycle objectives. The purpose of the life-cycle architecture milestone is defining both the system and the software architectures, the components of which we will study in Chapters 5, 6, and 7. The architectural choices must address the project risks addressed by the risk management plan, focusing on system evolution in the long term as well as system requirements in the short term.

The key elements of the initial operational capability are the readiness of the software itself, the site at which the system will be used, and the selection and training of the team that will use it. Boehm notes that different processes can be used to implement the initial operational capability, and different estimating techniques can be applied at different stages.

To supplement these milestones, Boehm suggests using the Win–Win spiral model, illustrated in Figure 3.22 and intended to be an extension of the spiral model we examined in Chapter 2. The model encourages participants to converge on a common understanding of the system's next-level objectives, alternatives, and constraints.

Boehm applied Win–Win, called the **Theory W** approach, to the U.S. Department of Defense's STARS program, whose focus was developing a set of prototype software

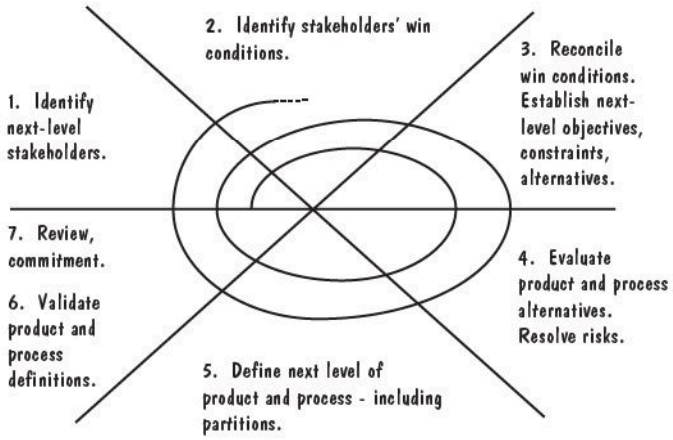


FIGURE 3.22 Win–Win spiral model (Boehm 1996).

engineering environments. The project was a good candidate for Theory W, because there was a great mismatch between what the government was planning to build and what the potential users needed and wanted. The Win–Win model led to several key compromises, including negotiation of a set of common, open interface specifications to enable tool vendors to reach a larger marketplace at reduced cost, and the inclusion of three demonstration projects to reduce risk. Boehm reports that Air Force costs on the project were reduced from $140 to $57 per delivered line of code and that quality improved from 3 to 0.035 faults per thousand delivered lines of code. Several other projects report similar success. TRW developed over half a million lines of code for complex distributed software within budget and schedule using Boehm's milestones with five increments. The first increment included distributed kernel software as part of the life-cycle architecture milestone; the project was required to demonstrate its ability to meet projections that the number of requirements would grow over time (Royce 1990).

3.7 INFORMATION SYSTEMS EXAMPLE

Let us return to the Piccadilly Television airtime sales system to see how we might estimate the amount of effort required to build the software. Because we are in the preliminary stages of understanding just what the software is to do, we can use COCOMO II's initial effort model to suggest the number of person-months needed. A **person-month** is the amount of time one person spends working on a software development project for one month. The COCOMO model assumes that the number of person-months does not include holidays and vacations, nor time off at weekends. The number of person-months is not the same as the time needed to finish building the system. For instance, a system may require 100 person-months, but it can be finished in one month by having ten people work in parallel for one month, or in two months by having five people work in parallel (assuming that the tasks can be accomplished in that manner).

The first COCOMO II model, application composition, is designed to be used in the earliest stages of development. Here, we compute application points to help us determine the likely size of the project. The application point count is determined from three calculations: the number of server data tables used with a screen or report, the number of client data tables used with a screen or report, and the percentage of screens, reports, and modules reused from previous applications. Let us assume that we are not reusing any code in building the Piccadilly system. Then we must begin our estimation process by predicting how many screens and reports we will be using in this application. Suppose our initial estimate is that we need three screens and one report:

- a booking screen to record a new advertising sales booking
- a ratecard screen showing the advertising rates for each day and hour
- an availability screen showing which time slots are available
- a sales report showing total sales for the month and year, and comparing them with previous months and years

For each screen or report, we use the guidance in Table 3.10 and an estimate of the number of data tables needed to produce a description of the screen or report. For example, the booking screen may require the use of three data tables: a table of available

TABLE 3.15 Ratings for Piccadilly Screens and Reports

Name	Screen or Report	Complexity	Weight
Booking	Screen	Simple	1
Ratecard	Screen	Simple	1
Availability	Screen	Medium	2
Sales	Report	Medium	5

time slots, a table of past usage by this customer, and a table of the contact information for this customer (such as name, address, tax number, and sales representative handling the sale). Thus, the number of data tables is fewer than four, so we must decide whether we need more than eight views. Since we are likely to need fewer than eight views, we rate the booking screen as "simple" according to the application point table. Similarly, we may rate the ratecard screen as "simple," the availability screen as "medium," and the sales report as "medium." Next, we use Table 3.11 to assign a complexity rate of 1 to simple screens, 2 to medium screens, and 5 to medium reports; a summary of our ratings is shown in Table 3.15.

We add all the weights in the rightmost column to generate a count of new application points (NOPS): 9. Suppose our developers have low experience and low CASE maturity. Table 3.12 tells us that the productivity rate for this circumstance is 7. Then the COCOMO model tells us that the estimated effort to build the Piccadilly system is NOP divided by the productivity rate, or 1.29 person-months.

As we understand more about the requirements for Piccadilly, we can use the other parts of COCOMO: the early design model and the postarchitecture model, based on nominal effort estimates derived from lines of code or function points. These models use a scale exponent computed from the project's scale factors, listed in Table 3.16.

TABLE 3.16 Scale Factors for COCOMO II Early Design and Postarchitecture Models

Scale Factors	Very Low	Low	Nominal	High	Very High	Extra High
Precedentedness	Thoroughly unprecedented	Largely unprecedented	Somewhat unprecedented	Generally familiar	Largely familiar	Thoroughly familiar
Flexibility	Rigorous	Occasional relaxation	Some relaxation	General conformity	Some conformity	General goals
Significant risks eliminated	Little (20%)	Some (40%)	Often (60%)	Generally (75%)	Mostly (90%)	Full (100%)
Team interaction process	Very difficult interactions	Some difficult interactions	Basically cooperative interactions	Largely cooperative	Highly cooperative	Seamless interactions
Process maturity	Determined by questionnaire	Determined by questionnaire	Determined by questionnaire	Determined by questionnaire	Determined by questionnaire	Determined by questionnaire

"Extra high" is equivalent to a rating of zero, "very high" to 1, "high" to 2, "nominal" to 3, "low" to 4, and "very low" to 5. Each of the scale factors is rated, and the sum of all ratings is used to weight the initial effort estimate. For example, suppose we know that the type of application we are building for Piccadilly is generally familiar to the development team; we can rate the first scale factor as "high." Similarly, we may rate flexibility as "very high," risk resolution as "nominal," team interaction as "high," and the maturity rating may turn out to be "low." We sum the ratings (2 + 1 + 3 + 2 + 4) to get a scale factor of 12. Then, we compute the scale exponent to be

$$1.01 + 0.01(12)$$

or 1.13. This scale exponent tells us that if our initial effort estimate is 100 person-months, then our new estimate, relative to the characteristics reflected in Table 3.16, is $100^{1.13}$, or 182 person-months. In a similar way, the cost drivers adjust this estimate based on characteristics such as tool usage, analyst expertise, and reliability requirements. Once we calculate the adjustment factor, we multiply by our 182 person-months estimate to yield an adjusted effort estimate.

3.8 REAL-TIME EXAMPLE

The board investigating the Ariane-5 failure examined the software, the documentation, and the data captured before and during flight to determine what caused the failure (Lions et al. 1996). Its report notes that the launcher began to disintegrate 39 seconds after takeoff because the angle of attack exceeded 20 degrees, causing the boosters to separate from the main stage of the rocket; this separation triggered the launcher's self-destruction. The angle of attack was determined by software in the on-board computer on the basis of data transmitted by the active inertial reference system, SRI2. As the report notes, SRI2 was supposed to contain valid flight data, but instead it contained a diagnostic bit pattern that was interpreted erroneously as flight data. The erroneous data had been declared a failure, and the SRI2 had been shut off. Normally, the on-board computer would have switched to the other inertial reference system, SRI1, but that, too, had been shut down for the same reason.

The error occurred in a software module that computed meaningful results only before lift-off. As soon as the launcher lifted off, the function performed by this module served no useful purpose, so it was no longer needed by the rest of the system. However, the module continued its computations for approximately 40 seconds of flight based on a requirement for the Ariane-4 that was not needed for Ariane-5.

The internal events that led to the failure were reproduced by simulation calculations supported by memory readouts and examination of the software itself. Thus, the Ariane-5 destruction might have been prevented had the project managers developed a risk management plan, reviewed it, and developed risk avoidance or mitigation plans for each identified risk. To see how, consider again the steps of Figure 3.15. The first stage of risk assessment is risk identification. The possible problem with reuse of the Ariane-4 software might have been identified by a decomposition of the functions; someone might have recognized early on that the requirements for Ariane-5 were

different from Ariane-4. Or an assumption analysis might have revealed that the assumptions for the SRI in Ariane-4 were different from those for Ariane-5.

Once the risks were identified, the analysis phase might have included simulations, which probably would have highlighted the problem that eventually caused the rocket's destruction. And prioritization would have identified the risk exposure if the SRI did not work as planned; the high exposure might have prompted the project team to examine the SRI and its workings more carefully before implementation.

Risk control involves risk reduction, management planning, and risk resolution. Even if the risk assessment activities had missed the problems inherent in reusing the SRI from Ariane-4, risk reduction techniques including risk avoidance analysis might have noted that both SRIs could have been shut down for the same underlying cause. Risk avoidance might have involved using SRIs with two different designs, so that the design error would have shut down one but not the other. Or the fact that the SRI calculations were not needed after lift-off might have prompted the designers or implementers to shut down the SRI earlier, before it corrupted the data for the angle calculations. Similarly, risk resolution includes plans for mitigation and continual reassessment of risk. Even if the risk of SRI failure had not been caught earlier, a risk reassessment during design or even during unit testing might have revealed the problem in the middle of development. A redesign or development at that stage would have been costly, but not as costly as the complete loss of Ariane-5 on its maiden voyage.

3.9 WHAT THIS CHAPTER MEANS FOR YOU

This chapter has introduced you to some of the key concepts in project management, including project planning, cost and schedule estimation, risk management, and team organization. You can make use of this information in many ways, even if you are not a manager. Project planning involves input from all team members, including you, and understanding the planning process and estimation techniques gives you a good idea of how your input will be used to make decisions for the whole team. Also, we have seen how the number of possible communication paths grows as the size of the team increases. You can take communication into account when you are planning your work and estimating the time it will take you to complete your next task.

We have also seen how communication styles differ and how they affect the way we interact with each other on the job. By understanding your teammates' styles, you can create reports and presentations for them that match their expectations and needs. You can prepare summary information for people with a bottom-line style and offer complete analytical information to those who are rational.

3.10 WHAT THIS CHAPTER MEANS FOR YOUR DEVELOPMENT TEAM

At the same time, you have learned how to organize a development team so that team interaction helps produce a better product. There are several choices for team structure, from a hierarchical chief programmer team to a loose, egoless approach. Each has its benefits, and each depends to some degree on the uncertainty and size of the project.

We have also seen how the team can work to anticipate and reduce risk from the project's beginning. Redundant functionality, team reviews, and other techniques can help us catch errors early, before they become embedded in the code as faults waiting to cause failures.

Similarly, cost estimation should be done early and often, including input from team members about progress in specifying, designing, coding, and testing the system. Cost estimation and risk management can work hand in hand; as cost estimates raise concerns about finishing on time and within budget, risk management techniques can be used to mitigate or even eliminate risks.

3.11 WHAT THIS CHAPTER MEANS FOR RESEARCHERS

This chapter has described many techniques that still require a great deal of research. Little is known about which team organizations work best in which situations. Likewise, cost- and schedule-estimation models are not as accurate as we would like them to be, and improvements can be made as we learn more about how project, process, product, and resource characteristics affect our efficiency and productivity. Some methods, such as machine learning, look promising but require a great deal of historical data to make them accurate. Researchers can help us to understand how to balance practicality with accuracy when using estimation techniques.

Similarly, a great deal of research is needed in making risk management techniques practical. The calculation of risk exposure is currently more an art than a science, and we need methods to help us make our risk calculations more relevant and our mitigation techniques more effective.

3.12 TERM PROJECT

Often, a company or organization must estimate the effort and time required to complete a project, even before detailed requirements are prepared. Using the approaches described in this chapter, or a tool of your choosing from other sources, estimate the effort required to build the Loan Arranger system. How many people will be required? What kinds of skills should they have? How much experience? What kinds of tools or techniques can you use to shorten the amount of time that development will take?

You may want to use more than one approach to generate your estimates. If you do, then compare and contrast the results. Examine each approach (its models and assumptions) to see what accounts for any substantial differences among estimates.

Once you have your estimates, evaluate them and their underlying assumptions to see how much uncertainty exists in them. Then, perform a risk analysis. Save your results; you can examine them at the end of the project to see which risks turned into real problems, and which ones were mitigated by your chosen risk strategies.

3.13 KEY REFERENCES

A great deal of information about COCOMO is available from the Center for Software Engineering at the University of Southern California. The Web site, http://sunset.usc.

edu/csse/research/COCOMOII/cocomo_main.html, points to current research on COCOMO, including a Java implementation of COCOMO II. It is at this site that you can also find out about COCOMO user-group meetings and obtain a copy of the COCOMO II user's manual. Related information about function points is available from IFPUG, the International Function Point User Group, in Westerville, Ohio.

The Center for Software Engineering also performs research on risk management. You can ftp a copy of its Software Risk Technical Advisor at ftp://usc.edu/pub/soft_engineering/demos/stra.tar.z and read about current research at http://sunset.usc.edu.

PC-based tools to support estimation are described and available from the Web site for Bournemouth University's Empirical Software Engineering Research Group: http://dec.bournemouth.ac.uk/ESERG.

Several companies producing commercial project management and cost-estimation tools have information available on their Web sites. Quantitative Software Management, producers of the SLIM cost-estimation package, is located at http://www.qsm.com. Likewise, Software Productivity Research offers a package called Checkpoint. Information can be found at http://www.spr.com. Computer Associates has developed a large suite of project management tools, including Estimacs for cost estimation and Planmacs for planning. A full description of its products is at http://www.cai.com/products.

The Software Technology Support Center at Hill Air Force Base in Ogden, Utah, produces a newsletter called *CrossTalk* that reports on method and tool evaluation. Its guidelines for successful acquisition and management can be found at http://stsc.hill.af.mil/stscdocs.html. The Center's Web pages also contain pointers to several technology areas, including project management and cost estimation; you can find the listing at http://stsc.hill.af.mil.

Team building and team interaction are essential on good software projects. Weinberg (1993) discusses work styles and their application to team building in the second volume of his series on software quality. Scholtes (1995) includes material on how to handle difficult team members.

Project management for small projects is necessarily different from that for large projects. The October 1999 issue of *IEEE Computer* addresses software engineering in the small, with articles about small projects, Internet time pressures, and extreme programming.

Project management for Web applications is somewhat different from more traditional software engineering. Mendes and Moseley (2006) explore the differences. In particular, they address estimation for Web applications in their book on "Web engineering."

3.14 EXERCISES

1. You are about to bake a two-layer birthday cake with icing. Describe the cake-baking project as a work breakdown structure. Generate an activity graph from that structure. What is the critical path?
2. Figure 3.23 is an activity graph for a software development project. The number corresponding to each edge of the graph indicates the number of days required to complete the activity represented by that branch. For example, it will take four days to complete the

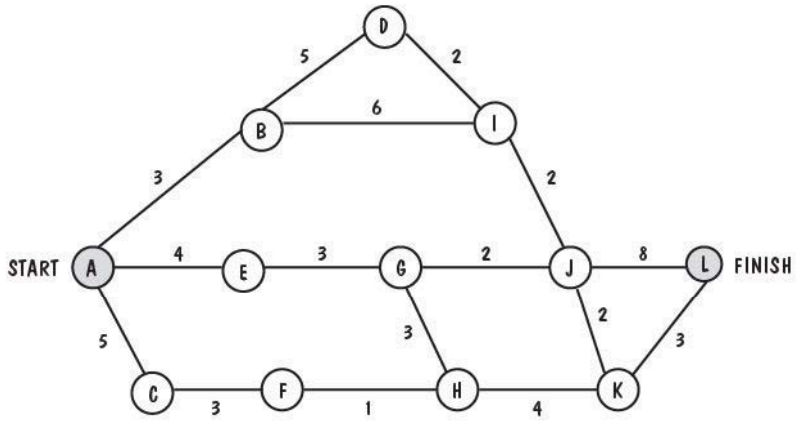


FIGURE 3.23 Activity graph for Exercise 2.

activity that ends in milestone E. For each activity, list its precursors and compute the earliest start time, the latest start time, and the slack. Then, identify the critical path.

3. Figure 3.24 is an activity graph. Find the critical path.

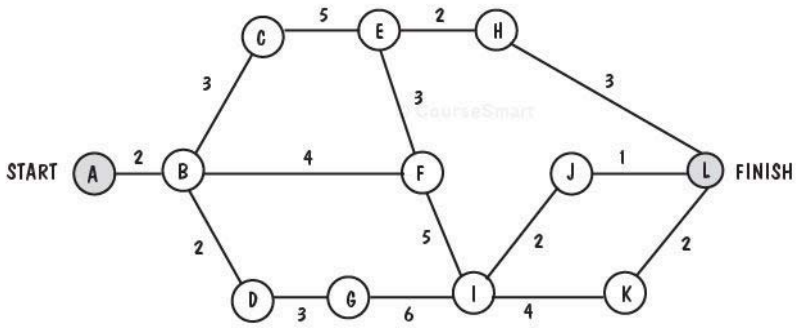


FIGURE 3.24 Activity graph for Exercise 3.

4. On a software development project, what kinds of activities can be performed in parallel? Explain why the activity graph sometimes hides the interdependencies of these activities.
5. Describe how adding personnel to a project that is behind schedule might make the project completion date even later.
6. A large government agency wants to contract with a software development firm for a project involving 20,000 lines of code. The Hardand Software Company uses Walston and Felix's estimating technique for determining the number of people required for the time needed to write that much code. How many person-months does Hardand estimate will be needed? If the government's estimate of size is 10% too low (i.e., 20,000 lines of code represent only 90% of the actual size), how many additional person-months will be needed? In general, if the government's size estimate is k% too low, by how much must the person-month estimate change?

7. Explain why it takes longer to develop a utility program than an applications program and longer still to develop a system program.
8. Manny's Manufacturing must decide whether to build or buy a software package to keep track of its inventory. Manny's computer experts estimate that it will cost $325,000 to buy the necessary programs. To build the programs in-house, programmers will cost $5000 each per month. What factors should Manny consider in making his decision? When is it better to build? To buy?
9. Brooks says that adding people to a late project makes it even later (Brooks 1975). Some schedule-estimation techniques seem to indicate that adding people to a project can shorten development time. Is this a contradiction? Why or why not?
10. Many studies indicate that two of the major reasons that a project is late are changing requirements (called requirements volatility or instability) and employee turnover. Review the cost models discussed in this chapter, plus any you may use on your job, and determine which models have cost factors that reflect the effects of these reasons.
11. Even on your student projects, there are significant risks to your finishing your project on time. Analyze a student software development project and list the risks. What is the risk exposure? What techniques can you use to mitigate each risk?
12. Many project managers plan their schedules based on programmer productivity on past projects. This productivity is often measured in terms of a unit of size per unit of time. For example, an organization may produce 300 lines of code per day or 1200 application points per month. Is it appropriate to measure productivity in this way? Discuss the measurement of productivity in terms of the following issues:
    - Different languages can produce different numbers of lines of code for implementation of the same design.
    - Productivity in lines of code cannot be measured until implementation begins.
    - Programmers may structure code to meet productivity goals.

4 Capturing the Requirements

In this chapter, we look at
- eliciting requirements from our customers
- modeling requirements
- reviewing requirements to ensure their quality
- documenting requirements for use by the design and test teams

In earlier chapters, when looking at various process models, we noted several key steps for successful software development. In particular, each proposed model of the software-development process includes activities aimed at capturing requirements: understanding our customers' fundamental problems and goals. Thus, our understanding of system intent and function starts with an examination of requirements. In this chapter, we look at the various types of requirements and their different sources, and we discuss how to resolve conflicting requirements. We detail a variety of modeling notations and requirements-specification methods, with examples of both automated and manual techniques. These models help us understand the requirements and document the relationships among them. Once the requirements are well understood, we learn how to review them for correctness and completeness. At the end of the chapter, we learn how to choose a requirements-specification method that is appropriate to the project under consideration, based on the project's size, scope, and the criticality of its mission.

Analyzing requirements involves much more than merely writing down what the customer wants. As we shall see, we must find requirements on which both we and the customer can agree and on which we can build our test procedures. First, let us examine exactly what requirements are, why they are so important (see Sidebar 4.1), and how we work with users and customers to define and document them.

SIDEBAR 4.1 WHY ARE REQUIREMENTS IMPORTANT?

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

(Brooks 1987)

In 1994, the Standish Group surveyed over 350 companies about their over 8000 software projects to find out how well they were faring. The results are sobering. Thirty-one percent of the software projects were canceled before they were completed. Moreover, in large companies, only 9% of the projects were delivered on time and within budget; 16% met those criteria in small companies (Standish 1994). Similar results have been reported since then; the bottom line is that developers have trouble delivering the right system on time and within budget.

To understand why, Standish (1995) asked the survey respondents to explain the causes of the failed projects. The top factors were reported to be

1. Incomplete requirements (13.1%)
2. Lack of user involvement (12.4%)
3. Lack of resources (10.6%)
4. Unrealistic expectations (9.9%)
5. Lack of executive support (9.3%)
6. Changing requirements and specifications (8.7%)
7. Lack of planning (8.1%)
8. System no longer needed (7.5%)

Notice that some part of the requirements elicitation, definition, and management process is involved in almost all of these causes. Lack of care in understanding, documenting, and managing requirements can lead to a myriad of problems: building a system that solves the wrong problem, that doesn't function as expected, or that is difficult for the users to understand and use.

Furthermore, requirements errors can be expensive if they are not detected and fixed early in the development process. Boehm and Papaccio (1988) report that if it costs $1 to find and fix a requirements-based problem during the requirements definition process, it can cost $5 to repair it during design, $10 during coding, $20 during unit testing, and as much as $200 after delivery of the system! So it pays to take time to understand the problem and its context, and to get the requirements right the first time.

4.1 THE REQUIREMENTS PROCESS

A customer who asks us to build a new system has some notion of what the system should do. Often, the customer wants to automate a manual task, such as paying bills electronically rather than with handwritten checks. Sometimes, the customer wants to

enhance or extend a current manual or automated system. For example, a telephone billing system that charged customers only for local telephone service and long-distance calls may be updated to bill for call forwarding, call waiting, and other new features. More and more frequently, a customer wants products that do things that have never been done before: tailoring electronic news to a user's interests, changing the shape of an airplane wing in mid-flight, or monitoring a diabetic's blood sugar and automatically controlling insulin dosage. No matter whether its functionality is old or new, a proposed software system has a purpose, usually expressed in terms of goals or desired behavior.

A **requirement** is an expression of desired behavior. A requirement deals with objects or entities, the states they can be in, and the functions that are performed to change states or object characteristics. For example, suppose we are building a system to generate paychecks for our customer's company. One requirement may be that the checks are to be issued every two weeks. Another may be that direct deposit of an employee's check is to be allowed for every employee at a certain salary level or higher. The customer may request access to the paycheck system from several different company locations. All of these requirements are specific descriptions of functions or characteristics that address the general purpose of the system: to generate paychecks. Thus, we look for requirements that identify key entities ("an employee is a person who is paid by the company"), limit entities ("an employee may be paid for no more than 40 hours per week"), or define relationships among entities ("employee X is supervised by employee Y if Y can authorize a change to X's salary").

Note that none of these requirements specify how the system is to be implemented. There is no mention of what database-management system to use, whether a client-server architecture will be employed, how much memory the computer is to have, or what programming language must be used to develop the system. These implementation-specific descriptions are not considered to be requirements (unless they are mandated by the customer). The goal of the requirements phase is to understand the customer's problems and needs. Thus, requirements focus on the customer and the problem, not on the solution or the implementation. We often say that requirements designate *what* behavior the customer wants, without saying *how* that behavior will be realized. Any discussion of a solution is premature until the problem is clearly defined.

It helps to describe requirements as interactions among real-world phenomena, without any reference to system phenomena. For example, billing requirements should refer to customers, services billed, billing periods and amounts—without mentioning system data or procedures. We take this approach partly to get at the heart of the customer's needs, because sometimes the stated needs are not the real needs. Moreover, the customer's problem is usually most easily stated in terms of the customer's business. Another reason we take this approach is to give the designer maximum flexibility in deciding how to carry out the requirements. During the **specification** phase, we will decide which requirements will be fulfilled by our software system (as opposed to requirements that are addressed by special-purpose hardware devices, by other software systems, or by human operators or users); during the **design** phase, we will devise a plan for how the specified behavior will be implemented.

Figure 4.1 illustrates the process of determining the requirements for a proposed software system. The person performing these tasks usually goes by the title of **requirements analyst** or **systems analyst**. As a requirements analyst, we first work with

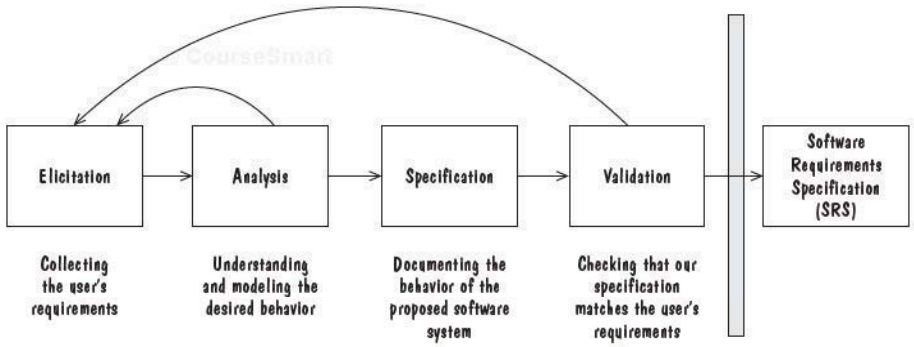


FIGURE 4.1 Process for capturing the requirements.

our customers to elicit the requirements by asking questions, examining current behavior, or demonstrating similar systems. Next, we capture the requirements in a model or a prototype. This exercise helps us to better understand the required behavior, and usually raises additional questions about what the customer wants to happen in certain situations (e.g., what if an employee leaves the company in the middle of a pay period?). Once the requirements are well understood, we progress to the specification phase, in which we decide which parts of the required behavior will be implemented in software. During validation, we check that our specification matches what the customer expects to see in the final product. Analysis and validation activities may expose problems or omissions in our models or specification that cause us to revisit the customer and revise our models and specification. The eventual outcome of the requirements process is a Software Requirements Specification (SRS), which is used to communicate to other software developers (designers, testers, maintainers) how the final product ought to behave. Sidebar 4.2 discusses how the use of agile methods affects the requirements process and the resulting requirements documents. The remainder of this chapter explores the requirements process in more detail.

4.2 REQUIREMENTS ELICITATION

Requirements elicitation is an especially critical part of the process. We must use a variety of techniques to determine what the users and customers really want. Sometimes, we are automating a manual system, so it is easy to examine what the system already does. But often, we must work with users and customers to understand a completely new problem. This task is rarely as simple as asking the right questions to pluck the requirements from the customer's head. At the early stage of a project, requirements are ill-formed and ill-understood by everyone. Customers are not always good at describing exactly what they want or need, and we are not always good at understanding someone else's business concerns. The customers know their business, but they cannot always describe their business problems to outsiders; their descriptions are full of jargon and

SIDEBAR 4.2 AGILE REQUIREMENTS MODELING

As we noted in Chapter 2, requirements analysis plays a large role in deciding whether to use agile methods as the basis for software development. If the requirements are tightly coupled and complex, or if future requirements and enhancements are likely to cause major changes to the system's architecture, then we may be better off with a "heavy" process that emphasizes up-front modeling. In a heavy process, developers put off coding until the requirements have been modeled and analyzed, an architecture is proposed that reflects the requirements, and a detailed design has been chosen. Each of these steps requires a model, and the models are related and coordinated so that the design fully implements the requirements. This approach is most appropriate for large-team development, where the documentation helps the developers to coordinate their work, and for safety-critical systems, where a system's correctness and safety are more important than its release date.

However, for problems where the requirements are uncertain, it can be cumbersome to employ a heavy process and have to update the models with every change to the requirements. As an alternative approach, agile methods gather and implement the requirements in increments. The initial release implements the most essential requirements, as defined by the stakeholders' business goals. As new requirements emerge with use of the system or with better understanding of the problem, they are implemented in subsequent releases of the system. This incremental development allows for "early and continuous delivery of valuable software" (Beck et al. 2004) and accommodates emergent and late-breaking requirements.

Extreme programming (XP) takes agile requirements processes to the extreme, in that the system is built to the requirements that happen to be defined at the time, with no planning or designing for possible future requirements. Moreover, XP forgoes traditional requirements documentation, and instead encodes the requirements as test cases that the eventual implementation must pass. Berry (2002a) points out that the trade-off for agile methods' flexibility is the difficulty of making changes to the system as requirements are added, deleted, or changed. But there can be additional problems: XP uses test cases to specify requirements, so a poorly written test case can lead to the kinds of misunderstandings described in this chapter.

assumptions with which we may not be familiar. Likewise, we as developers know about computer solutions, but not always about how possible solutions will affect our customers' business activities. We, too, have our jargon and assumptions, and sometimes we think everyone is speaking the same language, when in fact people have different meanings for the same words. It is only by discussing the requirements with everyone who has a stake in the system, coalescing these different views into a coherent set of requirements, and reviewing these documents with the stakeholders that we all come to an agreement about what the requirements are. (See Sidebar 4.3 for an alternative viewpoint.) If we cannot agree on what the requirements are, then the project is doomed to fail.

SIDEBAR 4.3 USING VIEWPOINTS TO MANAGE INCONSISTENCY

Although most software engineers strive for consistent requirements, Easterbrook and Nuseibeh (1996) argue that it is often desirable to tolerate and even encourage inconsistency during the requirements process. They claim that because the stakeholders' understanding of the domain and their needs evolve over time, it is pointless to try to resolve inconsistencies early in the requirements process. Early resolutions are expensive and often unnecessary (and can occur naturally as stakeholders revise their views). They can also be counter-productive if the resolution process focuses attention on how to come to agreement rather than on the underlying causes of the inconsistency (e.g., stakeholders' misunderstanding of the domain).

Instead, Easterbrook and Nuseibeh propose that stakeholders' views be documented and maintained as separate Viewpoints (Nuseibeh et al. 1994) throughout the software development process. The requirements analyst defines consistency rules that should apply between Viewpoints (e.g., how objects, states, or transitions in one Viewpoint correspond to similar entities in another Viewpoint; or how one Viewpoint refines another Viewpoint), and the Viewpoints are analyzed (possibly automatically) to see if they conform to the consistency rules. If the rules are violated, the inconsistencies are recorded as part of the Viewpoints, so that other software developers do not mistakenly implement a view that is being contested. The recorded inconsistencies are rechecked whenever an associated Viewpoint is modified, to see if the Viewpoints are still inconsistent; and the consistency rules are checked periodically, to see if any have been broken by evolving Viewpoints.

The outcome of this approach is a requirements document that accommodates all stakeholders' views at all times. Inconsistencies are highlighted but not addressed until there is sufficient information to make an informed decision. This way, we avoid committing ourselves prematurely to requirements or design decisions.

So who are the stakeholders? It turns out that there are many people who have something to contribute to the requirements of a new system:

- *Clients, who are the ones paying for the software to be developed:* By paying for the development, the clients are, in some sense, the ultimate stakeholders, and have the final say about what the product does (Robertson and Robertson 1999).
- *Customers, who buy the software after it is developed:* Sometimes the customer and the user are the same; other times, the customer is a business manager who is interested in improving the productivity of her employees. We have to understand the customers' needs well enough to build a product that they will buy and find useful.
- *Users, who are familiar with the current system and will use the future system:* These are the experts on how the current system works, which features are the most useful, and which aspects of the system need improving. We may want to consult also with special-interest groups of users, such as users with disabilities,

people who are unfamiliar with or uncomfortable using computers, expert users, and so on, to understand their particular needs.

- *Domain experts, who are familiar with the problem that the software must automate:* For example, we would consult a financial expert if we were building a financial package, or a meteorologist if our software were to model the weather. These people can contribute to the requirements, or will know about the kinds of environments to which the product will be exposed.
- *Market researchers, who have conducted surveys to determine future trends and potential customers' needs:* They may assume the role of the customer if our software is being developed for the mass market and no particular customer has been identified yet.
- *Lawyers or auditors, who are familiar with government, safety, or legal requirements:* For example, we might consult a tax expert to ensure that a payroll package adheres to the tax law. We may also consult with experts on standards that are relevant to the product's functions.
- *Software engineers or other technology experts:* These experts ensure that the product is technically and economically feasible. They can educate the customer about innovative hardware and software technologies, and can recommend new functionality that takes advantage of these technologies. They can also estimate the cost and development time of the product.

Each stakeholder has a particular view of the system and how it should work, and often these views conflict. One of the many skills of a requirements analyst is the ability to understand each view and capture the requirements in a way that reflects the concerns of each participant. For example, a customer may specify that a system perform a particular task, but the customer is not necessarily the user of the proposed system. The user may want the task to be performed in three modes: a learning mode, a novice mode, and an expert mode; this separation will allow the user to learn and master the system gradually. Some systems are implemented in this way, so that new users can adapt to the new system gradually. However, conflicts can arise when ease of use suggests a slower system than response-time requirements permit.

Also, different participants may expect differing levels of detail in the requirements documentation, in which case the requirements will need to be packaged in different ways for different people. In addition, users and developers may have preconceptions (right or wrong) about what the other group values and how it acts. Table 4.1 summarizes some of the common stereotypes. This table emphasizes the role that human interaction plays in the development of software systems; good requirements analysis requires excellent interpersonal skills as well as solid technical skills. The book's Web site contains suggestions for addressing each of these differences in perception.

In addition to interviewing stakeholders, other means of eliciting requirements include

- Reviewing available documentation, such as documented procedures of manual tasks, and specifications or user manuals of automated systems
- Observing the current system (if one exists), to gather objective information about how the users perform their tasks, and to better understand the system we are about to automate or to change; often, when a new computer system is developed,

TABLE 4.1 How Users and Developers View Each Other (Scharer 1990)

How Developers See Users	How Users See Developers
Users don't know what they want.	Developers don't understand operational needs.
Users can't articulate what they want.	Developers can't translate clearly stated needs into a successful system.
Users are unable to provide a usable statement of needs.	Developers set unrealistic standards for requirements definition.
Users have too many needs that are politically motivated.	Developers place too much emphasis on technicalities.
Users want everything right now.	Developers are always late.
Users can't remain on schedule.	Developers can't respond quickly to legitimately changing needs.
Users can't prioritize needs.	Developers are always over budget.
Users are unwilling to compromise.	Developers say "no" all the time.
Users refuse to take responsibility for the system.	Developers try to tell us how to do our jobs.
Users are not committed to development projects.	Developers ask users for time and effort, even to the detriment of the users' important primary duties.

the old system continues to be used because it provides some critical function that the designers of the new system overlooked

- Apprenticing with users (Beyer and Holtzblatt 1995), to learn about users' tasks in more detail, as the user performs them
- Interviewing users or stakeholders in groups, so that they will be inspired by one another's ideas
- Using domain-specific strategies, such as Joint Application Design (Wood and Silver 1995) or PIECES (Wetherbe 1984) for information systems, to ensure that stakeholders consider specific types of requirements that are relevant to their particular situations
- Brainstorming with current and potential users about how to improve the proposed product

The Volere requirements process model (Robertson and Robertson 1999), as shown in Figure 4.2, suggests some additional sources for requirements, such as templates and libraries of requirements from related systems that we have developed.

4.3 TYPES OF REQUIREMENTS

When most people think about requirements, they think about required *functionality*: What services should be provided? What operations should be performed? What should be the reaction to certain stimuli? How does required behavior change over time and in response to the history of events? A **functional requirement** describes required behavior in terms of required activities, such as reactions to inputs, and the state of each entity before and after an activity occurs. For instance, for a payroll system, the functional requirements state how often paychecks are issued, what input is necessary for a paycheck to be printed, under what conditions the amount of pay can be changed, and what causes the removal of an employee from the payroll list.

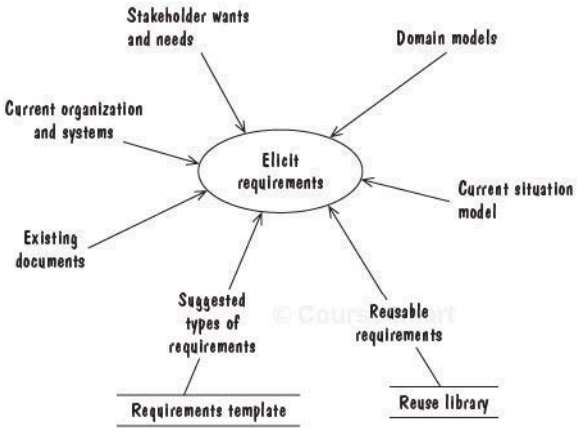


FIGURE 4.2 Sources of possible requirements (Robertson and Robertson 1999).

The functional requirements define the boundaries of the solution space for our problem. The solution space is the set of possible ways that software can be designed to implement the requirements, and initially that set can be very large. However, in practice it is usually not enough for a software product to compute correct outputs; there are other types of requirements that also distinguish between acceptable and unacceptable products. A **quality requirement**, or **nonfunctional requirement**, describes some quality characteristic that the software solution must possess, such as fast response time, ease of use, high reliability, or low maintenance costs. A **design constraint** is a design decision, such as choice of platform or interface components, that has already been made and that restricts the set of solutions to our problem. A **process constraint** is a restriction on the techniques or resources that can be used to build the system. For example, customers may insist that we use agile methods, so that they can use early versions of the system while we continue to add features. Thus, quality requirements, design constraints, and process constraints further restrict our solution space by differentiating acceptable, well-liked solutions from unused products. Table 4.2 gives examples of each kind of requirement.

Quality requirements sometimes sound like "motherhood" characteristics that all products ought to possess. After all, who is going to ask for a slow, unfriendly, unreliable, unmaintainable software system? It is better to think of quality requirements as design criteria that can be optimized and can be used to choose among alternative implementations of functional requirements. Given this approach, the question to be answered by the requirements is: To what extent must a product satisfy these quality requirements to be acceptable? Sidebar 4.4 explains how to express quality requirements such that we can test whether they are met.

Resolving Conflicts

In trying to elicit all types of requirements from all of the relevant stakeholders, we are bound to encounter conflicting ideas of what the requirements ought to be. It usually helps to ask the customer to prioritize requirements. This task forces the customer to

TABLE 4.2 Questions to Tease Out Different Types of Requirements

Functional Requirements

Functionality

- What will the system do?
- When will the system do it?
- Are there several modes of operation?
- What kinds of computations or data transformations must be performed?
- What are the appropriate reactions to possible stimuli?

Data

- For both input and output, what should be the format of the data?
- Must any data be retained for any period of time?

Design Constraints

Physical Environment

- Where is the equipment to be located?
- Is there one location or several?
- Are there any environmental restrictions, such as temperature, humidity, or magnetic interference?
- Are there any constraints on the size of the system?
- Are there any constraints on power, heating, or air conditioning?
- Are there constraints on the programming language because of existing software components?

Interfaces

- Is input coming from one or more other systems?
- Is output going to one or more other systems?
- Is there a prescribed way in which input/output data must be formatted?
- Is there a prescribed medium that the data must use?

Users

- Who will use the system?
- Will there be several types of users?
- What is the skill level of each user?

Process Constraints

Resources

- What materials, personnel, or other resources are needed to build the system?
- What skills must the developers have?

Documentation

- How much documentation is required?
- Should it be online, in book format, or both?
- To what audience should each type of documentation be addressed?

Standards

Quality Requirements

Performance

- Are there constraints on execution speed, response time, or throughput?
- What efficiency measures will apply to resource usage and response time?
- How much data will flow through the system?
- How often will data be received or sent?

Usability and Human Factors

- What kind of training will be required for each type of user?
- How easy should it be for a user to understand and use the system?
- How difficult should it be for a user to misuse the system?

Security

- Must access to the system or information be controlled?
- Should each user's data be isolated from the data of other users?
- Should user programs be isolated from other programs and from the operating system?
- Should precautions be taken against theft or vandalism?

Reliability and Availability

- Must the system detect and isolate faults?
- What is the prescribed mean time between failures?
- Is there a maximum time allowed for restarting the system after a failure?
- How often will the system be backed up?
- Must backup copies be stored at a different location?
- Should precautions be taken against fire or water damage?

Maintainability

- Will maintenance merely correct errors, or will it also include improving the system?
- When and in what ways might the system be changed in the future?
- How easy should it be to add features to the system?
- How easy should it be to port the system from one platform (computer, operating system) to another?

Precision and Accuracy

- How accurate must data calculations be?
- To what degree of precision must calculations be made?

Time to Delivery / Cost

- Is there a prescribed timetable for development?
- Is there a limit on the amount of money to be spent on development or on hardware or software?