Microsoft

CODE COMPLETE

2

Second Edition

A practical handbook of software construction

Steve McConnell

Two-time winner of the *Software Development* Magazine Jolt Award

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
McConnell, Steve
Code Complete / Steve McConnell.--2nd ed.
p. cm.
Includes index.
ISBN 0-7356-1967-0
1. Computer Software--Development--Handbooks, manuals, etc. I. Title.

QA76.76.D47M39 2004
005.1--dc22 2004049981

Printed and bound in the United States of America.

ISBN: 978-0-7356-1967-8

Twenty-fourth Printing: February 2015

Distributed in Canada by H.B. Fenn and Company Ltd. A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to *mspinput@ microsoft.com*.

Acquisitions Editors: Linda Engelman and Robin Van Steenburgh
Project Editor: Devon Musgrave
Indexer: Bill Myers
Principal Desktop Publisher: Carl Diltz

Body Part No. X10-53130

[2014-03-21]

Microsoft®

Code Complete, Second Edition

Steve McConnell

Contents at a Glance

Table of Contents

Part II Creating High-Quality Code

Part V Code Improvements

Preface

The gap between the best software engineering practice and the average practice is very wide—perhaps wider than in any other engineering discipline. A tool that disseminates good practice would be important.
—Fred Brooks

My primary concern in writing this book has been to narrow the gap between the knowledge of industry gurus and professors on the one hand and common commercial practice on the other. Many powerful programming techniques hide in journals and academic papers for years before trickling down to the programming public.

Although leading-edge software-development practice has advanced rapidly in recent years, common practice hasn't. Many programs are still buggy, late, and over budget, and many fail to satisfy the needs of their users. Researchers in both the software industry and academic settings have discovered effective practices that eliminate most of the programming problems that have been prevalent since the 1970s. Because these practices aren't often reported outside the pages of highly specialized technical journals, however, most programming organizations aren't yet using them today. Studies have found that it typically takes 5 to 15 years or more for a research development to make its way into commercial practice (Raghavan and Chand 1989, Rogers 1995, Parnas 1999). This handbook shortcuts the process, making key discoveries available to the average programmer now.

Who Should Read This Book?

The research and programming experience collected in this handbook will help you to create higher-quality software and to do your work more quickly and with fewer problems. This book will give you insight into why you've had problems in the past and will show you how to avoid problems in the future. The programming practices described here will help you keep big projects under control and help you maintain and modify software successfully as the demands of your projects change.

Experienced Programmers

This handbook serves experienced programmers who want a comprehensive, easy-to-use guide to software development. Because this book focuses on construction, the most familiar part of the software life cycle, it makes powerful software development techniques understandable to self-taught programmers as well as to programmers with formal training.

Technical Leads

Many technical leads have used *Code Complete* to educate less-experienced programmers on their teams. You can also use it to fill your own knowledge gaps. If you're an experienced programmer, you might not agree with all my conclusions (and I would be surprised if you did), but if you read this book and think about each issue, only rarely will someone bring up a construction issue that you haven't previously considered.

Self-Taught Programmers

If you haven't had much formal training, you're in good company. About 50,000 new developers enter the profession each year (BLS 2004, Hecker 2004), but only about 35,000 software-related degrees are awarded each year (NCES 2002). From these figures it's a short hop to the conclusion that many programmers don't receive a formal education in software development. Self-taught programmers are found in the emerging group of professionals—engineers, accountants, scientists, teachers, and small-business owners—who program as part of their jobs but who do not necessarily view themselves as programmers. Regardless of the extent of your programming education, this handbook can give you insight into effective programming practices.

Students

The counterpoint to the programmer with experience but little formal training is the fresh college graduate. The recent graduate is often rich in theoretical knowledge but poor in the practical know-how that goes into building production programs. The practical lore of good coding is often passed down slowly in the ritualistic tribal dances of software architects, project leads, analysts, and more-experienced programmers. Even more often, it's the product of the individual programmer's trials and errors. This book is an alternative to the slow workings of the traditional intellectual potlatch. It pulls together the helpful tips and effective development strategies previously available mainly by hunting and gathering from other people's experience. It's a hand up for the student making the transition from an academic environment to a professional one.

Where Else Can You Find This Information?

This book synthesizes construction techniques from a variety of sources. In addition to being widely scattered, much of the accumulated wisdom about construction has resided outside written sources for years (Hildebrand 1989, McConnell 1997a). There is nothing mysterious about the effective, high-powered programming techniques used by expert programmers. In the day-to-day rush of grinding out the latest project, however, few experts take the time to share what they have learned. Consequently, programmers may have difficulty finding a good source of programming information.

The techniques described in this book fill the void after introductory and advanced programming texts. After you have read *Introduction to Java*, *Advanced Java*, and *Advanced Advanced Java*, what book do you read to learn more about programming? You could read books about the details of Intel or Motorola hardware, Microsoft Windows or Linux operating-system functions, or another programming language—you can't use a language or program in an environment without a good reference to such details. But this is one of the few books that discusses programming per se. Some of the most beneficial programming aids are practices that you can use regardless of the environment or language you're working in. Other books generally neglect such practices, which is why this book concentrates on them.

The information in this book is distilled from many sources, as shown below. The only other way to obtain the information you'll find in this handbook would be to plow through a mountain of books and a few hundred technical journals and then add a significant amount of real-world experience. If you've already done all that, you can still benefit from this book's collecting the information in one place for easy reference.

Key Benefits of This Handbook

Whatever your background, this handbook can help you write better programs in less time and with fewer headaches.

Complete software-construction reference This handbook discusses general aspects of construction such as software quality and ways to think about programming. It gets into nitty-gritty construction details such as steps in building classes, ins and outs of using data and control structures, debugging, refactoring, and code-tuning techniques and strategies. You don't need to read it cover to cover to learn about these topics. The book is designed to make it easy to find the specific information that interests you.

Ready-to-use checklists This book includes dozens of checklists you can use to assess your software architecture, design approach, class and routine quality, variable names, control structures, layout, test cases, and much more.

State-of-the-art information This handbook describes some of the most up-to-date techniques available, many of which have not yet made it into common use. Because this book draws from both practice and research, the techniques it describes will remain useful for years.

Larger perspective on software development This book will give you a chance to rise above the fray of day-to-day fire fighting and figure out what works and what doesn't. Few practicing programmers have the time to read through the hundreds of books and journal articles that have been distilled into this handbook. The research and real-world experience gathered into this handbook will inform and stimulate your thinking about your projects, enabling you to take strategic action so that you don't have to fight the same battles again and again.

Absence of hype Some software books contain 1 gram of insight swathed in 10 grams of hype. This book presents balanced discussions of each technique's strengths and weaknesses. You know the demands of your particular project better than anyone else. This book provides the objective information you need to make good decisions about your specific circumstances.

Concepts applicable to most common languages This book describes techniques you can use to get the most out of whatever language you're using, whether it's C++, C#, Java, Microsoft Visual Basic, or other similar languages.

Numerous code examples The book contains almost 500 examples of good and bad code. I've included so many examples because, personally, I learn best from examples. I think other programmers learn best that way too.

The examples are in multiple languages because mastering more than one language is often a watershed in the career of a professional programmer. Once a programmer realizes that programming principles transcend the syntax of any specific language, the doors swing open to knowledge that truly makes a difference in quality and productivity.

To make the multiple-language burden as light as possible, I've avoided esoteric language features except where they're specifically discussed. You don't need to understand every nuance of the code fragments to understand the points they're making. If you focus on the point being illustrated, you'll find that you can read the code regardless of the language. I've tried to make your job even easier by annotating the significant parts of the examples.

Access to other sources of information This book collects much of the available information on software construction, but it's hardly the last word. Throughout the chapters, "Additional Resources" sections describe other books and articles you can read as you pursue the topics you find most interesting.

cc2e.com/1234

Book website Updated checklists, books, magazine articles, Web links, and other content are provided on a companion website at *cc2e.com*. To access information related to *Code Complete*, 2d ed., enter *cc2e.com/* followed by a four-digit code, an example of which is shown here in the left margin. These website references appear throughout the book.

Why This Handbook Was Written

The need for development handbooks that capture knowledge about effective development practices is well recognized in the software-engineering community. A report of the Computer Science and Technology Board stated that the biggest gains in software-development quality and productivity will come from codifying, unifying, and distributing existing knowledge about effective software-development practices (CSTB 1990, McConnell 1997a). The board concluded that the strategy for spreading that knowledge should be built on the concept of software-engineering handbooks.

The Topic of Construction Has Been Neglected

At one time, software development and coding were thought to be one and the same. But as distinct activities in the software-development life cycle have been identified, some of the best minds in the field have spent their time analyzing and debating methods of project management, requirements, design, and testing. The rush to study these newly identified areas has left code construction as the ignorant cousin of software development.

Discussions about construction have also been hobbled by the suggestion that treating construction as a distinct software development *activity* implies that construction must also be treated as a distinct *phase*. In reality, software activities and phases don't have to be set up in any particular relationship to each other, and it's useful to discuss the activity of construction regardless of whether other software activities are performed in phases, in iterations, or in some other way.

Construction Is Important

Another reason construction has been neglected by researchers and writers is the mistaken idea that, compared to other software-development activities, construction is a relatively mechanical process that presents little opportunity for improvement. Nothing could be further from the truth.

Code construction typically makes up about 65 percent of the effort on small projects and 50 percent on medium projects. Construction accounts for about 75 percent of the errors on small projects and 50 to 75 percent on medium and large projects. Any activity that accounts for 50 to 75 percent of the errors presents a clear opportunity for improvement. (Chapter 27 contains more details on these statistics.)

Some commentators have pointed out that although construction errors account for a high percentage of total errors, construction errors tend to be less expensive to fix than those caused by requirements and architecture, the suggestion being that they are therefore less important. The claim that construction errors cost less to fix is true but misleading because the cost of not fixing them can be incredibly high. Researchers have found that small-scale coding errors account for some of the most expensive software errors of all time, with costs running into hundreds of millions of dollars (Weinberg 1983, SEN 1990). An inexpensive cost to fix obviously does not imply that fixing them should be a low priority.

The irony of the shift in focus away from construction is that construction is the only activity that's guaranteed to be done. Requirements can be assumed rather than developed; architecture can be shortchanged rather than designed; and testing can be abbreviated or skipped rather than fully planned and executed. But if there's going to be a program, there has to be construction, and that makes construction a uniquely fruitful area in which to improve development practices.

No Comparable Book Is Available

In light of construction's obvious importance, I was sure when I conceived this book that someone else would already have written a book on effective construction practices. The need for a book about how to program effectively seemed obvious. But I found that only a few books had been written about construction and then only on parts of the topic. Some had been written 15 years or more earlier and employed relatively esoteric languages such as ALGOL, PL/I, Ratfor, and Smalltalk. Some were written by professors who were not working on production code. The professors wrote about techniques that worked for student projects, but they often had little idea of how the techniques would play out in full-scale development environments. Still other books trumpeted the authors' newest favorite methodologies but ignored the huge repository of mature practices that have proven their effectiveness over time.

When art critics get together they talk about Form and Structure and Meaning. When artists get together they talk about where you can buy cheap turpentine.
—Pablo Picasso

In short, I couldn't find any book that had even attempted to capture the body of practical techniques available from professional experience, industry research, and academic work. The discussion needed to be brought up to date for current programming languages, object-oriented programming, and leading-edge development practices. It seemed clear that a book about programming needed to be written by someone who was knowledgeable about the theoretical state of the art but who was also building enough production code to appreciate the state of the practice. I conceived this book as a full discussion of code construction—from one programmer to another.

Author Note

I welcome your inquiries about the topics discussed in this book, your error reports, or other related subjects. Please contact me at *stevemcc@construx.com*, or visit my website at *www.stevemcconnell.com*.

E-mail:

mspinput@microsoft.com

Acknowledgments

A book is never really written by one person (at least none of my books are). A second edition is even more a collective undertaking.

I'd like to thank the people who contributed review comments on significant portions of the book: Hákon Ágústsson, Scott Ambler, Will Barns, William D. Bartholomew, Lars Bergstrom, Ian Brockbank, Bruce Butler, Jay Cincotta, Alan Cooper, Bob Corrick, Al Corwin, Jerry Deville, Jon Eaves, Edward Estrada, Steve Gouldstone, Owain Griffiths, Matthew Harris, Michael Howard, Andy Hunt, Kevin Hutchison, Rob Jasper, Stephen Jenkins, Ralph Johnson and his Software Architecture Group at the University of Illinois, Marek Konopka, Jeff Langr, Andy Lester, Mitica Manu, Steve Mattingly, Gareth McCaughan, Robert McGovern, Scott Meyers, Gareth Morgan, Matt Peloquin, Bryan Pflug, Jeffrey Richter, Steve Rinn, Doug Rosenberg, Brian St. Pierre, Diomidis Spinellis, Matt Stephens, Dave Thomas, Andy Thomas-Cramer, John Vlissides, Pavel Vozenilek, Denny Williford, Jack Woolley, and Dee Zsombor.

Hundreds of readers sent comments about the first edition, and many more sent individual comments about the second edition. Thanks to everyone who took time to share their reactions to the book in its various forms.

Special thanks to the Construx Software reviewers who formally inspected the entire manuscript: Jason Hills, Bradey Honsinger, Abdul Nizar, Tom Reed, and Pamela Perrott. I was truly amazed at how thorough their review was, especially considering how many eyes had scrutinized the book before they began working on it. Thanks also to Bradey, Jason, and Pamela for their contributions to the *cc2e.com* website.

Working with Devon Musgrave, project editor for this book, has been a special treat. I've worked with numerous excellent editors on other projects, and Devon stands out as especially conscientious and easy to work with. Thanks, Devon! Thanks to Linda Engleman who championed the second edition; this book wouldn't have happened without her. Thanks also to the rest of the Microsoft Press staff, including Robin Van Steenburgh, Elden Nelson, Carl Diltz, Joel Panchot, Patricia Masserman, Bill Myers, Sandi Resnick, Barbara Norfleet, James Kramer, and Prescott Klassen.

I'd like to remember the Microsoft Press staff that published the first edition: Alice Smith, Arlene Myers, Barbara Runyan, Carol Luke, Connie Little, Dean Holmes, Eric Stroo, Erin O'Connor, Jeannie McGivern, Jeff Carey, Jennifer Harris, Jennifer Vick, Judith Bloch, Katherine Erickson, Kim Eggleston, Lisa Sandburg, Lisa Theobald, Margarite Hargrave, Mike Halvorson, Pat Forgette, Peggy Herman, Ruth Pettis, Sally Brunsman, Shawn Peck, Steve Murray, Wallis Bolz, and Zaafar Hasnain.

Thanks to the reviewers who contributed so significantly to the first edition: Al Corwin, Bill Kiestler, Brian Daugherty, Dave Moore, Greg Hitchcock, Hank Meuret, Jack Woolley, Joey Wyrick, Margot Page, Mike Klein, Mike Zevenbergen, Pat Forman, Peter Pathe, Robert L. Glass, Tammy Forman, Tony Pisculli, and Wayne Beardsley. Special thanks to Tony Garland for his exhaustive review: with 12 years' hindsight, I appreciate more than ever how exceptional Tony's several thousand review comments really were.

Checklists

Tables

Figures

Part I
Laying the Foundation

Chapter 1

Welcome to Software Construction

cc2e.com/0178

Contents

Related Topics

- Who should read this book: Preface
- Benefits of reading the book: Preface
- Why the book was written: Preface

You know what "construction" means when it's used outside software development. "Construction" is the work "construction workers" do when they build a house, a school, or a skyscraper. When you were younger, you built things out of "construction paper." In common usage, "construction" refers to the process of building. The construction process might include some aspects of planning, designing, and checking your work, but mostly "construction" refers to the hands-on part of creating something.

1.1 What Is Software Construction?

Developing computer software can be a complicated process, and in the last 25 years, researchers have identified numerous distinct activities that go into software development. They include

- Problem definition
- Requirements development
- Construction planning
- Software architecture, or high-level design
- Detailed design
- Coding and debugging
- Unit testing

- Integration testing
- Integration
- System testing
- Corrective maintenance

If you've worked on informal projects, you might think that this list represents a lot of red tape. If you've worked on projects that are too formal, you *know* that this list represents a lot of red tape! It's hard to strike a balance between too little and too much formality, and that's discussed later in the book.

If you've taught yourself to program or worked mainly on informal projects, you might not have made distinctions among the many activities that go into creating a software product. Mentally, you might have grouped all of these activities together as "programming." If you work on informal projects, the main activity you think of when you think about creating software is probably the activity the researchers refer to as "construction."

This intuitive notion of "construction" is fairly accurate, but it suffers from a lack of perspective. Putting construction in its context with other activities helps keep the focus on the right tasks during construction and appropriately emphasizes important nonconstruction activities. Figure 1-1 illustrates construction's place related to other software-development activities.

Figure 1-1 Construction activities are shown inside the gray circle. Construction focuses on coding and debugging but also includes detailed design, unit testing, integration testing, and other activities.

KEY POINT

As the figure indicates, construction is mostly coding and debugging but also involves detailed design, construction planning, unit testing, integration, integration testing, and other activities. If this were a book about all aspects of software development, it would feature nicely balanced discussions of all activities in the development process. Because this is a handbook of construction techniques, however, it places a lopsided emphasis on construction and only touches on related topics. If this book were a dog, it would nuzzle up to construction, wag its tail at design and testing, and bark at the other development activities.

Construction is also sometimes known as "coding" or "programming." "Coding" isn't really the best word because it implies the mechanical translation of a preexisting design into a computer language; construction is not at all mechanical and involves substantial creativity and judgment. Throughout the book, I use "programming" interchangeably with "construction."

In contrast to Figure 1-1's flat-earth view of software development, Figure 1-2 shows the round-earth perspective of this book.

Figure 1-2 This book focuses on coding and debugging, detailed design, construction planning, unit testing, integration, integration testing, and other activities in roughly these proportions.

Figure 1-1 and Figure 1-2 are high-level views of construction activities, but what about the details? Here are some of the specific tasks involved in construction:

- Verifying that the groundwork has been laid so that construction can proceed successfully
- Determining how your code will be tested

- Designing and writing classes and routines
- Creating and naming variables and named constants
- Selecting control structures and organizing blocks of statements
- Unit testing, integration testing, and debugging your own code
- Reviewing other team members' low-level designs and code and having them review yours
- Polishing code by carefully formatting and commenting it
- Integrating software components that were created separately
- Tuning code to make it faster and use fewer resources

For an even fuller list of construction activities, look through the chapter titles in the table of contents.

With so many activities at work in construction, you might say, "OK, Jack, what activities are *not* part of construction?" That's a fair question. Important nonconstruction activities include management, requirements development, software architecture, user-interface design, system testing, and maintenance. Each of these activities affects the ultimate success of a project as much as construction—at least the success of any project that calls for more than one or two people and lasts longer than a few weeks. You can find good books on each activity; many are listed in the "Additional Resources" sections throughout the book and in Chapter 35, "Where to Find More Information," at the end of the book.

1.2 Why Is Software Construction Important?

Since you're reading this book, you probably agree that improving software quality and developer productivity is important. Many of today's most exciting projects use software extensively. The Internet, movie special effects, medical life-support systems, space programs, aeronautics, high-speed financial analysis, and scientific research are a few examples. These projects and more conventional projects can all benefit from improved practices because many of the fundamentals are the same.

If you agree that improving software development is important in general, the question for you as a reader of this book becomes, Why is construction an important focus?

Here's why:

Cross-Reference For details on the relationship between project size and the percentage of time consumed by construction, see "Activity Proportions and Size" in Section 27.5.

Construction is a large part of software development Depending on the size of the project, construction typically takes 30 to 80 percent of the total time spent on a project. Anything that takes up that much project time is bound to affect the success of the project.

Construction is the central activity in software development Requirements and architecture are done before construction so that you can do construction effectively. System testing (in the strict sense of independent testing) is done after construction to verify that construction has been done correctly. Construction is at the center of the software-development process.

Cross-Reference For data on variations among programmers, see "Individual Variation" in Section 28.5.

With a focus on construction, the individual programmer's productivity can improve enormously A classic study by Sackman, Erikson, and Grant showed that the productivity of individual programmers varied by a factor of 10 to 20 during construction (1968). Since their study, their results have been confirmed by numerous other studies (Curtis 1981, Mills 1983, Curtis et al. 1986, Card 1987, Valett and McGarry 1989, DeMarco and Lister 1999, Boehm et al. 2000). This book helps all programmers learn techniques that are already used by the best programmers.

Construction's product, the source code, is often the only accurate description of the software In many projects, the only documentation available to programmers is the code itself. Requirements specifications and design documents can go out of date, but the source code is always up to date. Consequently, it's imperative that the source code be of the highest possible quality. Consistent application of techniques for source-code improvement makes the difference between a Rube Goldberg contraption and a detailed, correct, and therefore informative program. Such techniques are most effectively applied during construction.

KEY POINT

Construction is the only activity that's guaranteed to be done The ideal software project goes through careful requirements development and architectural design before construction begins. The ideal project undergoes comprehensive, statistically controlled system testing after construction. Imperfect, real-world projects, however, often skip requirements and design to jump into construction. They drop testing because they have too many errors to fix and they've run out of time. But no matter how rushed or poorly planned a project is, you can't drop construction; it's where the rubber meets the road. Improving construction is thus a way of improving any software-development effort, no matter how abbreviated.

1.3 How to Read This Book

This book is designed to be read either cover to cover or by topic. If you like to read books cover to cover, you might simply dive into Chapter 2, "Metaphors for a Richer Understanding of Software Development." If you want to get to specific programming tips, you might begin with Chapter 6, "Working Classes," and then follow the cross references to other topics you find interesting. If you're not sure whether any of this applies to you, begin with Section 3.2, "Determine the Kind of Software You're Working On."

Key Points

- Software construction is the central activity in software development; construction is the only activity that's guaranteed to happen on every project.
- The main activities in construction are detailed design, coding, debugging, integration, and developer testing (unit testing and integration testing).
- Other common terms for construction are "coding" and "programming."
- The quality of the construction substantially affects the quality of the software.
- In the final analysis, your understanding of how to do construction determines how good a programmer you are, and that's the subject of the rest of the book.

Chapter 2

Metaphors for a Richer Understanding of Software Development

cc2e.com/0278

Contents

Related Topic

- Heuristics in design: "Design Is a Heuristic Process" in Section 5.1

Computer science has some of the most colorful language of any field. In what other field can you walk into a sterile room, carefully controlled at 68°F, and find viruses, Trojan horses, worms, bugs, bombs, crashes, flames, twisted sex changers, and fatal errors?

These graphic metaphors describe specific software phenomena. Equally vivid metaphors describe broader phenomena, and you can use them to improve your understanding of the software-development process.

The rest of the book doesn't directly depend on the discussion of metaphors in this chapter. Skip it if you want to get to the practical suggestions. Read it if you want to think about software development more clearly.

2.1 The Importance of Metaphors

Important developments often arise out of analogies. By comparing a topic you understand poorly to something similar you understand better, you can come up with insights that result in a better understanding of the less-familiar topic. This use of metaphor is called "modeling."

The history of science is full of discoveries based on exploiting the power of metaphors. The chemist Kekulé had a dream in which he saw a snake grasp its tail in its mouth. When he awoke, he realized that a molecular structure based on a similar ring shape would account for the properties of benzene. Further experimentation confirmed the hypothesis (Barbour 1966).

The kinetic theory of gases was based on a "billiard-ball" model. Gas molecules were thought to have mass and to collide elastically, as billiard balls do, and many useful theorems were developed from this model.

The wave theory of light was developed largely by exploring similarities between light and sound. Light and sound have amplitude (brightness, loudness), frequency (color, pitch), and other properties in common. The comparison between the wave theories of sound and light was so productive that scientists spent a great deal of effort looking for a medium that would propagate light the way air propagates sound. They even gave it a name —"ether"—but they never found the medium. The analogy that had been so fruitful in some ways proved to be misleading in this case.

In general, the power of models is that they're vivid and can be grasped as conceptual wholes. They suggest properties, relationships, and additional areas of inquiry. Sometimes a model suggests areas of inquiry that are misleading, in which case the metaphor has been overextended. When the scientists looked for ether, they overextended their model.

As you might expect, some metaphors are better than others. A good metaphor is simple, relates well to other relevant metaphors, and explains much of the experimental evidence and other observed phenomena.

Consider the example of a heavy stone swinging back and forth on a string. Before Galileo, an Aristotelian looking at the swinging stone thought that a heavy object moved naturally from a higher position to a state of rest at a lower one. The Aristotelian would think that what the stone was really doing was falling with difficulty. When Galileo saw the swinging stone, he saw a pendulum. He thought that what the stone was really doing was repeating the same motion again and again, almost perfectly.

The suggestive powers of the two models are quite different. The Aristotelian who saw the swinging stone as an object falling would observe the stone's weight, the height to which it had been raised, and the time it took to come to rest. For Galileo's pendulum model, the prominent factors were different. Galileo observed the stone's weight, the radius of the pendulum's swing, the angular displacement, and the time per swing. Galileo discovered laws the Aristotelians could not discover because their model led them to look at different phenomena and ask different questions.

Metaphors contribute to a greater understanding of software-development issues in the same way that they contribute to a greater understanding of scientific questions. In his 1973 Turing Award lecture, Charles Bachman described the change from the prevailing earth-centered view of the universe to a sun-centered view. Ptolemy's earth-centered model had lasted without serious challenge for 1400 years. Then in 1543, Copernicus introduced a heliocentric theory, the idea that the sun rather than the earth was the center of the universe. This change in mental models led ultimately to the discovery of new planets, the reclassification of the moon as a satellite rather than as a planet, and a different understanding of humankind's place in the universe.

The value of metaphors should not be underestimated. Metaphors have the virtue of an expected behavior that is understood by all. Unnecessary communication and misunderstandings are reduced. Learning and education are quicker. In effect, metaphors are a way of internalizing and abstracting concepts, allowing one's thinking to be on a higher plane and low-level mistakes to be avoided.
—*Fernando J. Corbató*

Bachman compared the Ptolemaic-to-Copernican change in astronomy to the change in computer programming in the early 1970s. When Bachman made the comparison in 1973, data processing was changing from a computer-centered view of information systems to a database-centered view. Bachman pointed out that the ancients of data processing wanted to view all data as a sequential stream of cards flowing through a computer (the computer-centered view). The change was to focus on a pool of data on which the computer happened to act (a database-oriented view).

Today it's difficult to imagine anyone thinking that the sun moves around the earth. Similarly, it's difficult to imagine a programmer thinking that all data could be viewed as a sequential stream of cards. In both cases, once the old theory has been discarded, it seems incredible that anyone ever believed it at all. More fantastically, people who believed the old theory thought the new theory was just as ridiculous then as you think the old theory is now.

The earth-centered view of the universe hobbled astronomers who clung to it after a better theory was available. Similarly, the computer-centered view of the computing universe hobbled computer scientists who held on to it after the database-centered theory was available.

It's tempting to trivialize the power of metaphors. To each of the earlier examples, the natural response is to say, "Well, of course the right metaphor is more useful. The other metaphor was wrong!" Though that's a natural reaction, it's simplistic. The history of science isn't a series of switches from the "wrong" metaphor to the "right" one. It's a series of changes from "worse" metaphors to "better" ones, from less inclusive to more inclusive, from suggestive in one area to suggestive in another.

In fact, many models that have been replaced by better models are still useful. Engineers still solve most engineering problems by using Newtonian dynamics even though, theoretically, Newtonian dynamics have been supplanted by Einsteinian theory.

Software development is a younger field than most other sciences. It's not yet mature enough to have a set of standard metaphors. Consequently, it has a profusion of complementary and conflicting metaphors. Some are better than others. Some are worse. How well you understand the metaphors determines how well you understand software development.

2.2 How to Use Software Metaphors

KEY POINT

A software metaphor is more like a searchlight than a road map. It doesn't tell you where to find the answer; it tells you how to look for it. A metaphor serves more as a heuristic than it does as an algorithm.

An algorithm is a set of well-defined instructions for carrying out a particular task. An algorithm is predictable, deterministic, and not subject to chance. An algorithm tells you how to go from point A to point B with no detours, no side trips to points D, E, and F, and no stopping to smell the roses or have a cup of joe.

A heuristic is a technique that helps you look for an answer. Its results are subject to chance because a heuristic tells you only how to look, not what to find. It doesn't tell you how to get directly from point A to point B; it might not even know where point A and point B are. In effect, a heuristic is an algorithm in a clown suit. It's less predictable, it's more fun, and it comes without a 30-day, money-back guarantee.

Here is an algorithm for driving to someone's house: Take Highway 167 south to Puyallup. Take the South Hill Mall exit and drive 4.5 miles up the hill. Turn right at the light by the grocery store, and then take the first left. Turn into the driveway of the large tan house on the left, at 714 North Cedar.

Cross-Reference For details on how to use heuristics in designing software, see "Design Is a Heuristic Process" in Section 5.1.

Here's a heuristic for getting to someone's house: Find the last letter we mailed you. Drive to the town in the return address. When you get to town, ask someone where our house is. Everyone knows us—someone will be glad to help you. If you can't find anyone, call us from a public phone, and we'll come get you.

The difference between an algorithm and a heuristic is subtle, and the two terms overlap somewhat. For the purposes of this book, the main difference between the two is the level of indirection from the solution. An algorithm gives you the instructions directly. A heuristic tells you how to discover the instructions for yourself, or at least where to look for them.

Having directions that told you exactly how to solve your programming problems would certainly make programming easier and the results more predictable. But programming science isn't yet that advanced and may never be. The most challenging part of programming is conceptualizing the problem, and many errors in programming are conceptual errors. Because each program is conceptually unique, it's difficult or impossible to create a general set of directions that lead to a solution in every case. Thus, knowing how to approach problems in general is at least as valuable as knowing specific solutions for specific problems.

How do you use software metaphors? Use them to give you insight into your programming problems and processes. Use them to help you think about your programming activities and to help you imagine better ways of doing things. You won't be able to look at a line of code and say that it violates one of the metaphors described in this chapter. Over time, though, the person who uses metaphors to illuminate the software-development process will be perceived as someone who has a better understanding of programming and produces better code faster than people who don't use them.

2.3 Common Software Metaphors

A confusing abundance of metaphors has grown up around software development. David Gries says writing software is a science (1981). Donald Knuth says it's an art (1998). Watts Humphrey says it's a process (1989). P. J. Plauger and Kent Beck say it's like driving a car, although they draw nearly opposite conclusions (Plauger 1993, Beck 2000). Alistair Cockburn says it's a game (2002). Eric Raymond says it's like a bazaar (2000). Andy Hunt and Dave Thomas say it's like gardening. Paul Heckel says it's like filming *Snow White and the Seven Dwarfs* (1994). Fred Brooks says that it's like farming, hunting werewolves, or drowning with dinosaurs in a tar pit (1995). Which are the best metaphors?

Software Penmanship: Writing Code

The most primitive metaphor for software development grows out of the expression "writing code." The writing metaphor suggests that developing a program is like writing a casual letter—you sit down with pen, ink, and paper and write it from start to finish. It doesn't require any formal planning, and you figure out what you want to say as you go.

Many ideas derive from the writing metaphor. Jon Bentley says you should be able to sit down by the fire with a glass of brandy, a good cigar, and your favorite hunting dog to enjoy a "literate program" the way you would a good novel. Brian Kernighan and P. J. Plauger named their programming-style book *The Elements of Programming Style* (1978) after the writing-style book *The Elements of Style* (Strunk and White 2000). Programmers often talk about "program readability."

HARD DATA

For an individual's work or for small-scale projects, the letter-writing metaphor works adequately, but for other purposes it leaves the party early—it doesn't describe software development fully or adequately. Writing is usually a one-person activity, whereas a software project will most likely involve many people with many different responsibilities. When you finish writing a letter, you stuff it into an envelope and mail it. You can't change it anymore, and for all intents and purposes it's complete. Software isn't as difficult to change and is hardly ever fully complete. As much as 90 percent of the development effort on a typical software system comes after its initial release, with two-thirds being typical (Pigoski 1997). In writing, a high premium is placed on originality. In software construction, trying to create truly original work is often less effective than focusing on the reuse of design ideas, code, and test cases from previous projects. In short, the writing metaphor implies a software-development process that's too simple and rigid to be healthy.

Plan to throw one away; you will, anyhow.
—*Fred Brooks*

If you plan to throw one away, you will throw away two.
—*Craig Zerouni*

Unfortunately, the letter-writing metaphor has been perpetuated by one of the most popular software books on the planet, Fred Brooks's *The Mythical Man-Month* (Brooks 1995). Brooks says, "Plan to throw one away; you will, anyhow." This conjures up an image of a pile of half-written drafts thrown into a wastebasket, as shown in Figure 2-1.

Figure 2-1 The letter-writing metaphor suggests that the software process relies on expensive trial and error rather than careful planning and design.

Planning to throw one away might be practical when you're writing a polite how-do-you-do to your aunt. But extending the metaphor of "writing" software to a plan to throw one away is poor advice for software development, where a major system already costs as much as a 10-story office building or an ocean liner. It's easy to grab the brass ring if you can afford to sit on your favorite wooden pony for an unlimited number of spins around the carousel. The trick is to get it the first time around–or to take several chances when they're cheapest. Other metaphors better illuminate ways of attaining such goals.

Software Farming: Growing a System

In contrast to the rigid writing metaphor, some software developers say you should envision creating software as something like planting seeds and growing crops. You design a piece, code a piece, test a piece, and add it to the system a little bit at a time. By taking small steps, you minimize the trouble you can get into at any one time.

KEY POINT

Sometimes a good technique is described with a bad metaphor. In such cases, try to keep the technique and come up with a better metaphor. In this case, the incremental technique is valuable, but the farming metaphor is terrible.

Further Reading For an illustration of a different farming metaphor, one that's applied to software maintenance, see the chapter "On the Origins of Designer Intuition" in *Rethinking Systems Analysis and Design* (Weinberg 1988).

The idea of doing a little bit at a time might bear some resemblance to the way crops grow, but the farming analogy is weak and uninformative, and it's easy to replace with the better metaphors described in the following sections. It's hard to extend the farming metaphor beyond the simple idea of doing things a little bit at a time. If you buy into the farming metaphor, imagined in Figure 2-2, you might find yourself talking about fertilizing the system plan, thinning the detailed design, increasing code yields through effective land management, and harvesting the code itself. You'll talk about rotating in a crop of C++ instead of barley, of letting the land rest for a year to increase the supply of nitrogen in the hard disk.

The weakness in the software-farming metaphor is its suggestion that you don't have any direct control over how the software develops. You plant the code seeds in the spring. *Farmer's Almanac* and the Great Pumpkin willing, you'll have a bumper crop of code in the fall.

Figure 2-2 It's hard to extend the farming metaphor to software development appropriately.

Software Oyster Farming: System Accretion

Sometimes people talk about growing software when they really mean software accretion. The two metaphors are closely related, but software accretion is the more insightful image. "Accretion," in case you don't have a dictionary handy, means any growth or increase in size by a gradual external addition or inclusion. Accretion describes the way an oyster makes a pearl, by gradually adding small amounts of calcium carbonate. In geology, "accretion" means a slow addition to land by the deposit of waterborne sediment. In legal terms, "accretion" means an increase of land along the shores of a body of water by the deposit of waterborne sediment.

Cross-Reference For details on how to apply incremental strategies to system integration, see Section 29.2, "Integration Frequency—Phased or Incremental?"

This doesn't mean that you have to learn how to make code out of waterborne sediment; it means that you have to learn how to add to your software systems a small amount at a time. Other words closely related to accretion are "incremental," "iterative," "adaptive," and "evolutionary." Incremental designing, building, and testing are some of the most powerful software-development concepts available.

In incremental development, you first make the simplest possible version of the system that will run. It doesn't have to accept realistic input, it doesn't have to perform realistic manipulations on data, it doesn't have to produce realistic output–it just has to be a skeleton strong enough to hold the real system as it's developed. It might call dummy classes for each of the basic functions you have identified. This basic beginning is like the oyster's beginning a pearl with a small grain of sand.

After you've formed the skeleton, little by little you lay on the muscle and skin. You change each of the dummy classes to real classes. Instead of having your program pretend to accept input, you drop in code that accepts real input. Instead of having your program pretend to produce output, you drop in code that produces real output. You add a little bit of code at a time until you have a fully working system.

The anecdotal evidence in favor of this approach is impressive. Fred Brooks, who in 1975 advised building one to throw away, said that nothing in the decade after he wrote his landmark book *The Mythical Man-Month* so radically changed his own practice or its effectiveness as incremental development (1995). Tom Gilb made the same point in his breakthrough book, *Principles of Software Engineering Management* (1988), which introduced Evolutionary Delivery and laid the groundwork for much of today's Agile programming approach. Numerous current methodologies are based on this idea (Beck 2000, Cockburn 2002, Highsmith 2002, Reifer 2002, Martin 2003, Larman 2004).

As a metaphor, the strength of the incremental metaphor is that it doesn't overpromise. It's harder than the farming metaphor to extend inappropriately. The image of an oyster forming a pearl is a good way to visualize incremental development, or accretion.

Software Construction: Building Software

KEY POINT

The image of "building" software is more useful than that of "writing" or "growing" software. It's compatible with the idea of software accretion and provides more detailed guidance. Building software implies various stages of planning, preparation, and execution that vary in kind and degree depending on what's being built. When you explore the metaphor, you find many other parallels.

Building a four-foot tower requires a steady hand, a level surface, and 10 undamaged beer cans. Building a tower 100 times that size doesn't merely require 100 times as many beer cans. It requires a different kind of planning and construction altogether.

If you're building a simple structure—a doghouse, say—you can drive to the lumber store and buy some wood and nails. By the end of the afternoon, you'll have a new house for Fido. If you forget to provide for a door, as shown in Figure 2-3, or make some other mistake, it's not a big problem; you can fix it or even start over from the beginning. All you've wasted is part of an afternoon. This loose approach is appropriate for small software projects too. If you use the wrong design for 1000 lines of code, you can refactor or start over completely without losing much.

Figure 2-3 The penalty for a mistake on a simple structure is only a little time and maybe some embarrassment.

If you're building a house, the building process is more complicated, and so are the consequences of poor design. First you have to decide what kind of house you want to build—analogous in software development to problem definition. Then you and an architect have to come up with a general design and get it approved. This is similar to software architectural design. You draw detailed blueprints and hire a contractor. This is similar to detailed software design. You prepare the building site, lay a foundation, frame the house, put siding and a roof on it, and plumb and wire it. This is similar to software construction. When most of the house is done, the landscapers, painters, and decorators come in to make the best of your property and the home you've built. This is similar to software optimization. Throughout the process, various inspectors come to check the site, foundation, frame, wiring, and other inspectables. This is similar to software reviews and inspections.

Greater complexity and size imply greater consequences in both activities. In building a house, materials are somewhat expensive, but the main expense is labor. Ripping out a wall and moving it six inches is expensive not because you waste a lot of nails but because you have to pay the people for the extra time it takes to move the wall. You have to make the design as good as possible, as suggested by Figure 2-4, so that you don't waste time fixing mistakes that could have been avoided. In building a software product, materials are even less expensive, but labor costs just as much. Changing a report format is just as expensive as moving a wall in a house because the main cost component in both cases is people's time.

Figure 2-4 More complicated structures require more careful planning.

What other parallels do the two activities share? In building a house, you won't try to build things you can buy already built. You'll buy a washer and dryer, dishwasher, refrigerator, and freezer. Unless you're a mechanical wizard, you won't consider building them yourself. You'll also buy prefabricated cabinets, counters, windows, doors, and bathroom fixtures. If you're building a software system, you'll do the same thing. You'll make extensive use of high-level language features rather than writing your own operating-system-level code. You might also use prebuilt libraries of container classes, scientific functions, user interface classes, and database-manipulation classes. It generally doesn't make sense to code things you can buy ready-made.

If you're building a fancy house with first-class furnishings, however, you might have your cabinets custom-made. You might have a dishwasher, refrigerator, and freezer built in to look like the rest of your cabinets. You might have windows custom-made in unusual shapes and sizes. This customization has parallels in software development. If you're building a first-class software product, you might build your own scientific functions for better speed or accuracy. You might build your own container classes, user interface classes, and database classes to give your system a seamless, perfectly consistent look and feel.

Both building construction and software construction benefit from appropriate levels of planning. If you build software in the wrong order, it's hard to code, hard to test, and hard to debug. It can take longer to complete, or the project can fall apart because everyone's work is too complex and therefore too confusing when it's all combined.

Careful planning doesn't necessarily mean exhaustive planning or over-planning. You can plan out the structural supports and decide later whether to put in hardwood floors or carpeting, what color to paint the walls, what roofing material to use, and so on. A well-planned project improves your ability to change your mind later about details. The more experience you have with the kind of software you're building, the more details you can take for granted. You just want to be sure that you plan enough so that lack of planning doesn't create major problems later.

The construction analogy also helps explain why different software projects benefit from different development approaches. In building, you'd use different levels of planning, design, and quality assurance if you're building a warehouse or a toolshed than if you're building a medical center or a nuclear reactor. You'd use still different approaches for building a school, a skyscraper, or a three-bedroom home. Likewise, in software you might generally use flexible, lightweight development approaches, but sometimes you'll need rigid, heavyweight approaches to achieve safety goals and other goals.

Making changes in the software brings up another parallel with building construction. To move a wall six inches costs more if the wall is load-bearing than if it's merely a partition between rooms. Similarly, making structural changes in a program costs more than adding or deleting peripheral features.

Finally, the construction analogy provides insight into extremely large software projects. Because the penalty for failure in an extremely large structure is severe, the structure has to be over-engineered. Builders make and inspect their plans carefully. They build in margins of safety; it's better to pay 10 percent more for stronger material than to have a skyscraper fall over. A great deal of attention is paid to timing. When the Empire State Building was built, each delivery truck had a 15-minute margin in which to make its delivery. If a truck wasn't in place at the right time, the whole project was delayed.

Likewise, for extremely large software projects, planning of a higher order is needed than for projects that are merely large. Capers Jones reports that a software system with one million lines of code requires an average of 69 *kinds* of documentation (1998). The requirements specification for such a system would typically be about 4000–5000 pages long, and the design documentation can easily be two or three times as extensive as the requirements. It's unlikely that an individual would be able to understand the complete design for a project of this size—or even read it. A greater degree of preparation is appropriate.

We build software projects comparable in economic size to the Empire State Building, and technical and managerial controls of similar stature are needed.

Further Reading For some good comments about extending the construction metaphor, see "What Supports the Roof?" (Starr 2003).

The building-construction metaphor could be extended in a variety of other directions, which is why the metaphor is so powerful. Many terms common in software development derive from the building metaphor: software architecture, scaffolding, construction, foundation classes, and tearing code apart. You'll probably hear many more.

Applying Software Techniques: The Intellectual Toolbox

KEY POINT

People who are effective at developing high-quality software have spent years accumulating dozens of techniques, tricks, and magic incantations. The techniques are not rules; they are analytical tools. A good craftsman knows the right tool for the job and knows how to use it correctly. Programmers do, too. The more you learn about programming, the more you fill your mental toolbox with analytical tools and the knowledge of when to use them and how to use them correctly.

Cross-Reference For details on selecting and combining methods in design, see Section 5.3, "Design Building Blocks: Heuristics."

In software, consultants sometimes tell you to buy into certain software-development methods to the exclusion of other methods. That's unfortunate because if you buy into any single methodology 100 percent, you'll see the whole world in terms of that methodology. In some instances, you'll miss opportunities to use other methods better suited to your current problem. The toolbox metaphor helps to keep all the methods, techniques, and tips in perspective—ready for use when appropriate.

Combining Metaphors

KEY POINT

Because metaphors are heuristic rather than algorithmic, they are not mutually exclusive. You can use both the accretion and the construction metaphors. You can use writing if you want to, and you can combine writing with driving, hunting for werewolves, or drowning in a tar pit with dinosaurs. Use whatever metaphor or combination of metaphors stimulates your own thinking or communicates well with others on your team.

Using metaphors is a fuzzy business. You have to extend them to benefit from the heuristic insights they provide. But if you extend them too far or in the wrong direction, they'll mislead you. Just as you can misuse any powerful tool, you can misuse metaphors, but their power makes them a valuable part of your intellectual toolbox.

Additional Resources

cc2e.com/0285

Among general books on metaphors, models, and paradigms, the touchstone book is by Thomas Kuhn.

Kuhn, Thomas S. *The Structure of Scientific Revolutions*, 3d ed. Chicago, IL: The University of Chicago Press, 1996. Kuhn's book on how scientific theories emerge, evolve, and succumb to other theories in a Darwinian cycle set the philosophy of science on its ear when it was first published in 1962. It's clear and short, and it's loaded with interesting examples of the rise and fall of metaphors, models, and paradigms in science.

Floyd, Robert W. "The Paradigms of Programming." 1978 Turing Award Lecture. *Communications of the ACM*, August 1979, pp. 455–60. This is a fascinating discussion of models in software development, and Floyd applies Kuhn's ideas to the topic.

Chapter 3

Measure Twice, Cut Once: Upstream Prerequisites

cc2e.com/0309

Contents

Related Topics

- Key construction decisions: Chapter 4
- Effect of project size on construction and prerequisites: Chapter 27
- Relationship between quality goals and construction activities: Chapter 20
- Managing construction: Chapter 28
- Design: Chapter 5

Before beginning construction of a house, a builder reviews blueprints, checks that all permits have been obtained, and surveys the house's foundation. A builder prepares for building a skyscraper one way, a housing development a different way, and a doghouse a third way. No matter what the project, the preparation is tailored to the project's specific needs and done conscientiously before construction begins.

This chapter describes the work that must be done to prepare for software construction. As with building construction, much of the success or failure of the project has already been determined before construction begins. If the foundation hasn't been laid well or the planning is inadequate, the best you can do during construction is to keep damage to a minimum.

The carpenter's saying, "Measure twice, cut once" is highly relevant to the construction part of software development, which can account for as much as 65 percent of the total project costs. The worst software projects end up doing construction two or three times or more. Doing the most expensive part of the project twice is as bad an idea in software as it is in any other line of work.

Although this chapter lays the groundwork for successful software construction, it doesn't discuss construction directly. If you're feeling carnivorous or you're already well versed in the software-engineering life cycle, look for the construction meat beginning in Chapter 5, "Design in Construction." If you don't like the idea of prerequisites to construction, review Section 3.2, "Determine the Kind of Software You're Working On," to see how prerequisites apply to your situation, and then take a look at the data in Section 3.1, which describes the cost of not doing prerequisites.

3.1 Importance of Prerequisites

Cross-Reference Paying attention to quality is also the best way to improve productivity. For details, see Section 20.5, "The General Principle of Software Quality."

A common denominator of programmers who build high-quality software is their use of high-quality practices. Such practices emphasize quality at the beginning, middle, and end of a project.

If you emphasize quality at the end of a project, you emphasize system testing. Testing is what many people think of when they think of software quality assurance. Testing, however, is only one part of a complete quality-assurance strategy, and it's not the most influential part. Testing can't detect a flaw such as building the wrong product or building the right product in the wrong way. Such flaws must be worked out earlier than in testing—before construction begins.

If you emphasize quality in the middle of the project, you emphasize construction practices. Such practices are the focus of most of this book.

KEY POINT

If you emphasize quality at the beginning of the project, you plan for, require, and design a high-quality product. If you start the process with designs for a Pontiac Aztek, you can test it all you want to, and it will never turn into a Rolls-Royce. You might build the best possible Aztek, but if you want a Rolls-Royce, you have to plan from the beginning to build one. In software development, you do such planning when you define the problem, when you specify the solution, and when you design the solution.

Since construction is in the middle of a software project, by the time you get to construction, the earlier parts of the project have already laid some of the groundwork for success or failure. During construction, however, you should at least be able to determine how good your situation is and to back up if you see the black clouds of failure looming on the horizon. The rest of this chapter describes in detail why proper preparation is important and tells you how to determine whether you're really ready to begin construction.

Do Prerequisites Apply to Modern Software Projects?

The methodology used should be based on choice of the latest and best, and not based on ignorance. It should also be laced liberally with the old and dependable.
—*Harlan Mills*

Some people have asserted that upstream activities such as architecture, design, and project planning aren't useful on modern software projects. In the main, such assertions are not well supported by research, past or present, or by current data. (See the rest of this chapter for details.) Opponents of prerequisites typically show examples of prerequisites that have been done poorly and then point out that such work isn't effective. Upstream activities can be done well, however, and industry data from the 1970s to the present day indicates that projects will run best if appropriate preparation activities are done before construction begins in earnest.

KEY POINT

The overarching goal of preparation is risk reduction: a good project planner clears major risks out of the way as early as possible so that the bulk of the project can proceed as smoothly as possible. By far the most common project risks in software development are poor requirements and poor project planning, thus preparation tends to focus on improving requirements and project plans.

Preparation for construction is not an exact science, and the specific approach to risk reduction must be decided project by project. Details can vary greatly among projects. For more on this, see Section 3.2.

Causes of Incomplete Preparation

You might think that all professional programmers know about the importance of preparation and check that the prerequisites have been satisfied before jumping into construction. Unfortunately, that isn't so.

Further Reading For a description of a professional development program that cultivates these skills, see Chapter 16 of *Professional Software Development* (McConnell 2004).

A common cause of incomplete preparation is that the developers who are assigned to work on the upstream activities do not have the expertise to carry out their assignments. The skills needed to plan a project, create a compelling business case, develop comprehensive and accurate requirements, and create high-quality architectures are far from trivial, but most developers have not received training in how to perform these activities. When developers don't know how to do upstream work, the recommendation to "do more upstream work" sounds like nonsense: If the work isn't being done well in the first place, doing *more* of it will not be useful! Explaining how to perform these activities is beyond the scope of this book, but the "Additional Resources" sections at the end of this chapter provide numerous options for gaining that expertise.

cc2e.com/0316

Some programmers do know how to perform upstream activities, but they don't prepare because they can't resist the urge to begin coding as soon as possible. If you feed your horse at this trough, I have two suggestions. Suggestion 1: Read the argument in the next section. It may tell you a few things you haven't thought of. Suggestion 2: Pay attention to the problems you experience. It takes only a few large programs to learn that you can avoid a lot of stress by planning ahead. Let your own experience be your guide.

A final reason that programmers don't prepare is that managers are notoriously unsympathetic to programmers who spend time on construction prerequisites. People like Barry Boehm, Grady Booch, and Karl Wiegers have been banging the requirements and design drums for 25 years, and you'd expect that managers would have started to understand that software development is more than coding.

Further Reading For many entertaining variations on this theme, read Gerald Weinberg's classic, *The Psychology of Computer Programming* (Weinberg 1998).

A few years ago, however, I was working on a Department of Defense project that was focusing on requirements development when the Army general in charge of the project came for a visit. We told him that we were developing requirements and that we were mainly talking to our customer, capturing requirements, and outlining the design. He insisted on seeing code anyway. We told him there was no code, but he walked around a work bay of 100 people, determined to catch someone programming. Frustrated by seeing so many people away from their desks or working on requirements and design, the large, round man with the loud voice finally pointed to the engineer sitting next to me and bellowed, "What's he doing? He must be writing code!" In fact, the engineer was working on a document-formatting utility, but the general wanted to find code, thought it looked like code, and wanted the engineer to be working on code, so we told him it was code.

This phenomenon is known as the WISCA or WIMP syndrome: Why Isn't Sam Coding Anything? or Why Isn't Mary Programming?

If the manager of your project pretends to be a brigadier general and orders you to start coding right away, it's easy to say, "Yes, Sir!" (What's the harm? The old guy must know what he's talking about.) This is a bad response, and you have several better alternatives. First, you can flatly refuse to do work in an ineffective order. If your relationships with your boss and your bank account are healthy enough for you to be able to do this, good luck.

A second questionable alternative is pretending to be coding when you're not. Put an old program listing on the corner of your desk. Then go right ahead and develop your requirements and architecture, with or without your boss's approval. You'll do the project faster and with higher-quality results. Some people find this approach ethically objectionable, but from your boss's perspective, ignorance will be bliss.

Third, you can educate your boss in the nuances of technical projects. This is a good approach because it increases the number of enlightened bosses in the world. The next subsection presents an extended rationale for taking the time to do prerequisites before construction.

Finally, you can find another job. Despite economic ups and downs, good programmers are perennially in short supply (BLS 2002), and life is too short to work in an unenlightened programming shop when plenty of better alternatives are available.

Utterly Compelling and Foolproof Argument for Doing Prerequisites Before Construction

Suppose you've already been to the mountain of problem definition, walked a mile with the man of requirements, shed your soiled garments at the fountain of architecture, and bathed in the pure waters of preparedness. Then you know that before you implement a system, you need to understand what the system is supposed to do and how it's supposed to do it.

KEY POINT

Part of your job as a technical employee is to educate the nontechnical people around you about the development process. This section will help you deal with managers and bosses who have not yet seen the light. It's an extended argument for doing requirements and architecture—getting the critical aspects right—before you begin coding, testing, and debugging. Learn the argument, and then sit down with your boss and have a heart-to-heart talk about the programming process.

Appeal to Logic

One of the key ideas in effective programming is that preparation is important. It makes sense that before you start working on a big project, you should plan the project. Big projects require more planning; small projects require less. From a management point of view, planning means determining the amount of time, number of people, and number of computers the project will need. From a technical point of view, planning means understanding what you want to build so that you don't waste money building the wrong thing. Sometimes users aren't entirely sure what they want at first, so it might take more effort than seems ideal to find out what they really want. But that's cheaper than building the wrong thing, throwing it away, and starting over.

It's also important to think about how to build the system before you begin to build it. You don't want to spend a lot of time and money going down blind alleys when there's no need to, especially when that increases costs.

Appeal to Analogy

Building a software system is like any other project that takes people and money. If you're building a house, you make architectural drawings and blueprints before you begin pounding nails. You'll have the blueprints reviewed and approved before you pour any concrete. Having a technical plan counts just as much in software.

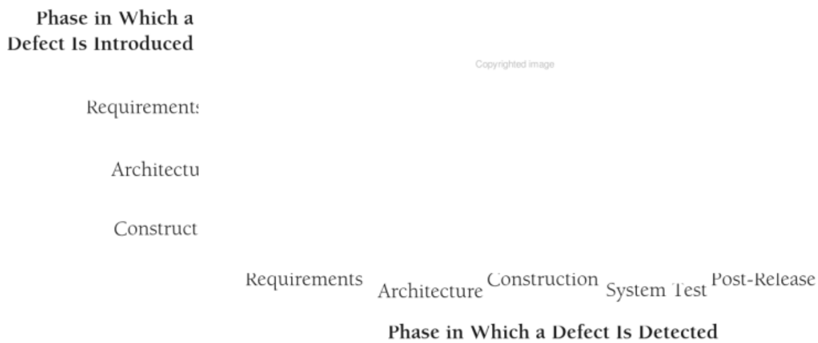

Figure 3-1 The cost to fix a defect rises dramatically as the time from when it's introduced to when it's detected increases. This remains true whether the project is highly sequential (doing 100 percent of requirements and design up front) or highly iterative (doing 5 percent of requirements and design up front).

HARD DATA

The average project still exerts most of its defect-correction effort on the right side of Figure 3-1, which means that debugging and associated rework takes about 50 percent of the time spent in a typical software development cycle (Mills 1983; Boehm 1987a; Cooper and Mullen 1993; Fishman 1996; Haley 1996; Wheeler, Brykczynski, and Meeson 1996; Jones 1998; Shull et al. 2002; Wiegers 2002). Dozens of companies have found that simply focusing on correcting defects earlier rather than later in a project can cut development costs and schedules by factors of two or more (McConnell 2004). This is a healthy incentive to find and fix your problems as early as you can.

Boss-Readiness Test

When you think your boss understands the importance of working on prerequisites before moving into construction, try the test below to be sure.

Which of these statements are self-fulfilling prophecies?

- We'd better start coding right away because we're going to have a lot of debugging to do.
- We haven't planned much time for testing because we're not going to find many defects.

- We've investigated requirements and design so much that I can't think of any major problems we'll run into during coding or debugging.

All of these statements are self-fulfilling prophecies. Aim for the last one.

If you're still not convinced that prerequisites apply to your project, the next section will help you decide.

3.2 Determine the Kind of Software You're Working On

Capers Jones, Chief Scientist at Software Productivity Research, summarized 20 years of software research by pointing out that he and his colleagues have seen 40 different methods for gathering requirements, 50 variations in working on software designs, and 30 kinds of testing applied to projects in more than 700 different programming languages (Jones 2003).

Different kinds of software projects call for different balances between preparation and construction. Every project is unique, but projects do tend to fall into general development styles. Table 3-2 shows three of the most common kinds of projects and lists the practices that are typically best suited to each kind of project.

Table 3-2 Typical Good Practices for Three Common Kinds of Software Projects

	Kind of Software		
	Business Systems	**Mission-Critical Systems**	**Embedded Life-Critical Systems**
Typical applications	Internet site Intranet site Inventory management Games Management information systems Payroll system	Embedded software Games Internet site Packaged software Software tools Web services	Avionics software Embedded software Medical devices Operating systems Packaged software
Life-cycle models	Agile development (Extreme Programming, Scrum, timebox development, and so on) Evolutionary prototyping	Staged delivery Evolutionary delivery Spiral development	Staged delivery Spiral development Evolutionary delivery

Table 3-2 Typical Good Practices for Three Common Kinds of Software Projects

	Kind of Software		
	Business Systems	**Mission-Critical Systems**	**Embedded Life-Critical Systems**
Planning and management	Incremental project planning As-needed test and QA planning Informal change control	Basic up-front planning Basic test planning As-needed QA planning Formal change control	Extensive up-front planning Extensive test planning Extensive QA planning Rigorous change control
Requirements	Informal require-ments specification	Semiformal require-ments specification As-needed require-ments reviews	Formal requirements specification Formal requirements inspections
Design	Design and coding are combined	Architectural design Informal detailed design As-needed design reviews	Architectural design Formal architecture inspections Formal detailed design Formal detailed design inspections
Construction	Pair programming or individual coding Informal check-in procedure or no check-in procedure	Pair programming or individual coding Informal check-in procedure As-needed code reviews	Pair programming or individual coding Formal check-in procedure Formal code inspections
Testing and QA	Developers test their own code Test-first development Little or no testing by a separate test group	Developers test their own code Test-first development Separate testing group	Developers test their own code Test-first development Separate testing group Separate QA group
Deployment	Informal deploy-ment procedure	Formal deployment procedure	Formal deployment procedure

On real projects, you'll find infinite variations on the three themes presented in this table; however, the generalities in the table are illuminating. Business systems projects tend to benefit from highly iterative approaches, in which planning, requirements, and architecture are interleaved with construction, system testing, and quality-assurance activities. Life-critical systems tend to require more sequential approaches—requirements stability is part of what's needed to ensure ultrahigh levels of reliability.

Iterative Approaches' Effect on Prerequisites

Some writers have asserted that projects that use iterative techniques don't need to focus on prerequisites much at all, but that point of view is misinformed. Iterative approaches tend to reduce the impact of inadequate upstream work, but they don't eliminate it. Consider the examples shown in Table 3-3 of projects that don't focus on prerequisites. One project is conducted sequentially and relies solely on testing to discover defects; the other is conducted iteratively and discovers defects as it progresses. The first approach delays most defect correction work to the end of the project, making the costs higher, as noted in Table 3-1. The iterative approach absorbs rework piecemeal over the course of the project, which makes the total cost lower. The data in this table and the next is for purposes of illustration only, but the relative costs of the two general approaches are well supported by the research described earlier in this chapter.

Table 3-3 Effect of Skipping Prerequisites on Sequential and Iterative Projects

	Approach #1: Sequential Approach Without Prerequisites		**Approach #2: Iterative Approach Without Prerequisites**	
Project Completion Status	Cost of Work	Cost of Rework	Cost of Work	Cost of Rework
20%	$100,000	$0	$100,000	$75,000
40%	$100,000	$0	$100,000	$75,000
60%	$100,000	$0	$100,000	$75,000
80%	$100,000	$0	$100,000	$75,000
100%	$100,000	$0	$100,000	$75,000
End-of-Project Rework	$0	$500,000	$0	$0
TOTAL	$500,000	$500,000	$500,000	$375,000
GRAND TOTAL		$1,000,000		$875,000

The iterative project that abbreviates or eliminates prerequisites will differ in two ways from a sequential project that does the same thing. First, average defect correction costs will be lower because defects will tend to be detected closer to the time they were inserted into the software. However, the defects will still be detected late in each iteration, and correcting them will require parts of the software to be redesigned, recoded, and retested—which makes the defect-correction cost higher than it needs to be.

Second, with iterative approaches costs will be absorbed piecemeal, throughout the project, rather than being clustered at the end. When all the dust settles, the total cost will be similar but it won't seem as high because the price will have been paid in small installments over the course of the project, rather than paid all at once at the end.

As Table 3-4 illustrates, a focus on prerequisites can reduce costs regardless of whether you use an iterative or a sequential approach. Iterative approaches are usually a better option for many reasons, but an iterative approach that ignores prerequisites can end up costing significantly more than a sequential project that pays close attention to prerequisites.

Table 3-4 Effect of Focusing on Prerequisites on Sequential and Iterative Projects

	Approach #3: Sequential Approach with Prerequisites		Approach #4: Iterative Approach with Prerequisites	
Project completion status	Cost of Work	Cost of Rework	Cost of Work	Cost of Rework
20%	$100,000	$20,000	$100,000	$10,000
40%	$100,000	$20,000	$100,000	$10,000
60%	$100,000	$20,000	$100,000	$10,000
80%	$100,000	$20,000	$100,000	$10,000
100%	$100,000	$20,000	$100,000	$10,000
End-of-Project Rework	$0	$0	$0	$0
TOTAL	$500,000	$100,000	$500,000	$50,000
GRAND TOTAL		$600,000		$550,000

KEY POINT

As Table 3-4 suggested, most projects are neither completely sequential nor completely iterative. It isn't practical to specify 100 percent of the requirements or design up front, but most projects find value in identifying at least the most critical requirements and architectural elements early.

Cross-Reference For details on how to adapt your development approach for programs of different sizes, see Chapter 27, "How Program Size Affects Construction."

One common rule of thumb is to plan to specify about 80 percent of the requirements up front, allocate time for additional requirements to be specified later, and then practice systematic change control to accept only the most valuable new requirements as the project progresses. Another alternative is to specify only the most important 20 percent of the requirements up front and plan to develop the rest of the software in small increments, specifying additional requirements and designs as you go. Figures 3-2 and 3-3 reflect these different approaches.

A problem definition defines what the problem is without any reference to possible solutions. It's a simple statement, maybe one or two pages, and it should sound like a problem. The statement "We can't keep up with orders for the Gigatron" sounds like a problem and is a good problem definition. The statement "We need to optimize our automated data-entry system to keep up with orders for the Gigatron" is a poor problem definition. It doesn't sound like a problem; it sounds like a solution.

As shown in Figure 3-4, problem definition comes before detailed requirements work, which is a more in-depth investigation of the problem.

Figure 3-4 The problem definition lays the foundation for the rest of the programming process.

The problem definition should be in user language, and the problem should be described from a user's point of view. It usually should not be stated in technical computer terms. The best solution might not be a computer program. Suppose you need a report that shows your annual profit. You already have computerized reports that show quarterly profits. If you're locked into the programmer mindset, you'll reason that adding an annual report to a system that already does quarterly reports should be easy. Then you'll pay a programmer to write and debug a time-consuming program that calculates annual profits. If you're not locked into the programmer mindset, you'll pay your secretary to create the annual figures by taking one minute to add up the quarterly figures on a pocket calculator.

The exception to this rule applies when the problem is with the computer: compile times are too slow or the programming tools are buggy. Then it's appropriate to state the problem in computer or programmer terms.

As Figure 3-5 suggests, without a good problem definition, you might put effort into solving the wrong problem.

Figure 3-5 Be sure you know what you're aiming at before you shoot.

KEY POINT

The penalty for failing to define the problem is that you can waste a lot of time solving the wrong problem. This is a double-barreled penalty because you also don't solve the right problem.

3.4 Requirements Prerequisite

Requirements describe in detail what a software system is supposed to do, and they are the first step toward a solution. The requirements activity is also known as "requirements development," "requirements analysis," "analysis," "requirements definition," "software requirements," "specification," "functional spec," and "spec."

Why Have Official Requirements?

An explicit set of requirements is important for several reasons.

Explicit requirements help to ensure that the user rather than the programmer drives the system's functionality. If the requirements are explicit, the user can review them and agree to them. If they're not, the programmer usually ends up making requirements decisions during programming. Explicit requirements keep you from guessing what the user wants.

Explicit requirements also help to avoid arguments. You decide on the scope of the system before you begin programming. If you have a disagreement with another programmer about what the program is supposed to do, you can resolve it by looking at the written requirements.

KEY POINT

Paying attention to requirements helps to minimize changes to a system after development begins. If you find a coding error during coding, you change a few lines of code and work goes on. If you find a requirements error during coding, you have to alter the design to meet the changed requirement. You might have to throw away part of the old design, and because it has to accommodate code that's already written, the new design will take longer than it would have in the first place. You also have to discard code and test cases affected by the requirement change and write new code and test cases. Even code that's otherwise unaffected must be retested so that you can be sure the changes in other areas haven't introduced any new errors.

HARD DATA

As Table 3-1 reported, data from numerous organizations indicates that on large projects an error in requirements detected during the architecture stage is typically 3 times as expensive to correct as it would be if it were detected during the requirements stage. If detected during coding, it's 5–10 times as expensive; during system test, 10 times; and post-release, a whopping 10–100 times as expensive as it would be if it were detected during requirements development. On smaller projects with lower administrative costs, the multiplier post-release is closer to 5–10 than 100 (Boehm and Turner 2004). In either case, it isn't money you'd want to have taken out of your salary.

Specifying requirements adequately is a key to project success, perhaps even more important than effective construction techniques. (See Figure 3-6.) Many good books have been written about how to specify requirements well. Consequently, the next few sections don't tell you how to do a good job of specifying requirements, they tell you how to determine whether the requirements have been done well and how to make the best of the requirements you have.

Figure 3-6 Without good requirements, you can have the right general problem but miss the mark on specific aspects of the problem.

The Myth of Stable Requirements

Requirements are like water. They're easier to build on when they're frozen.
—*Anonoymous*

Stable requirements are the holy grail of software development. With stable requirements, a project can proceed from architecture to design to coding to testing in a way that's orderly, predictable, and calm. This is software heaven! You have predictable expenses, and you never have to worry about a feature costing 100 times as much to implement as it would otherwise because your user didn't think of it until you were finished debugging.

It's fine to hope that once your customer has accepted a requirements document, no changes will be needed. On a typical project, however, the customer can't reliably describe what is needed before the code is written. The problem isn't that the customers are a lower life form. Just as the more you work with the project, the better you understand it, the more they work with it, the better they understand it. The development process helps customers better understand their own needs, and this is a major source of requirements changes (Curtis, Krasner, and Iscoe 1988; Jones 1998; Wiegers 2003). A plan to follow the requirements rigidly is actually a plan not to respond to your customer.

HARD DATA

How much change is typical? Studies at IBM and other companies have found that the average project experiences about a 25 percent change in requirements during development (Boehm 1981, Jones 1994, Jones 2000), which accounts for 70 to 85 percent of the rework on a typical project (Leffingwell 1997, Wiegers 2003).

Maybe you think the Pontiac Aztek was the greatest car ever made, belong to the Flat Earth Society, and make a pilgrimage to the alien landing site at Roswell, New Mexico, every four years. If you do, go ahead and believe that requirements won't change on your projects. If, on the other hand, you've stopped believing in Santa Claus and the Tooth Fairy, or at least have stopped admitting it, you can take several steps to minimize the impact of requirements changes.

Handling Requirements Changes During Construction

Here are several things you can do to make the best of changing requirements during construction:

KEY POINT

Use the requirements checklist at the end of the section to assess the quality of your requirements If your requirements aren't good enough, stop work, back up, and make them right before you proceed. Sure, it feels like you're getting behind if you stop coding at this stage. But if you're driving from Chicago to Los Angeles, is it a waste of time to stop and look at a road map when you see signs for New York? No. If you're not heading in the right direction, stop and check your course.

Make sure everyone knows the cost of requirements changes Clients get excited when they think of a new feature. In their excitement, their blood thins and runs to their medulla oblongata and they become giddy, forgetting all the meetings you had to discuss requirements, the signing ceremony, and the completed requirements document. The easiest way to handle such feature-intoxicated people is to say, "Gee, that sounds like a great idea. Since it's not in the requirements document, I'll work up a revised schedule and cost estimate so that you can decide whether you want to do it now or later." The words "schedule" and "cost" are more sobering than coffee and a cold shower, and many "must haves" will quickly turn into "nice to haves."

If your organization isn't sensitive to the importance of doing requirements first, point out that changes at requirements time are much cheaper than changes later. Use this chapter's "Utterly Compelling and Foolproof Argument for Doing Prerequisites Before Construction."

Cross-Reference For details on handling changes to design and code, see Section 28.2, "Configuration Management."

Set up a change-control procedure If your client's excitement persists, consider establishing a formal change-control board to review such proposed changes. It's all right for customers to change their minds and to realize that they need more capabilities. The problem is their suggesting changes so frequently that you can't keep up. Having a built-in procedure for controlling changes makes everyone happy. You're happy because you know that you'll have to work with changes only at specific times. Your customers are happy because they know that you have a plan for handling their input.

Cross-Reference For details on iterative development approaches, see "Iterate" in Section 5.4 and Section 29.3, "Incremental Integration Strategies."

Use development approaches that accommodate changes Some development approaches maximize your ability to respond to changing requirements. An evolutionary prototyping approach helps you explore a system's requirements before you send your forces in to build it. Evolutionary delivery is an approach that delivers the system in stages. You can build a little, get a little feedback from your users, adjust your design a little, make a few changes, and build a little more. The key is using short development cycles so that you can respond to your users quickly.

Further Reading For details on development approaches that support flexible requirements, see *Rapid Development* (McConnell 1996).

Dump the project If the requirements are especially bad or volatile and none of the suggestions above are workable, cancel the project. Even if you can't really cancel the project, think about what it would be like to cancel it. Think about how much worse it would have to get before you would cancel it. If there's a case in which you would dump it, at least ask yourself how much difference there is between your case and that case.

Cross-Reference For details on the differences between formal and informal projects (often caused by differences in project size), see Chapter 27, "How Program Size Affects Construction."

Keep your eye on the business case for the project Many requirements issues disappear before your eyes when you refer back to the business reason for doing the project. Requirements that seemed like good ideas when considered as "features" can seem like terrible ideas when you evaluate the "incremental business value." Programmers who remember to consider the business impact of their decisions are worth their weight in gold—although I'll be happy to receive my commission for this advice in cash.

design—architecture refers to design constraints that apply systemwide, whereas high-level design refers to design constraints that apply at the subsystem or multiple-class level, but not necessarily systemwide.

Because this book is about construction, this section doesn't tell you how to develop a software architecture; it focuses on how to determine the quality of an existing architecture. Because architecture is one step closer to construction than requirements, however, the discussion of architecture is more detailed than the discussion of requirements.

KEY POINT

Why have architecture as a prerequisite? Because the quality of the architecture determines the conceptual integrity of the system. That in turn determines the ultimate quality of the system. A well-thought-out architecture provides the structure needed to maintain a system's conceptual integrity from the top levels down to the bottom. It provides guidance to programmers—at a level of detail appropriate to the skills of the programmers and to the job at hand. It partitions the work so that multiple developers or multiple development teams can work independently.

Good architecture makes construction easy. Bad architecture makes construction almost impossible. Figure 3-7 illustrates another problem with bad architecture.

Figure 3-7 Without good software architecture, you may have the right problem but the wrong solution. It may be impossible to have successful construction.

HARD DATA

Architectural changes are expensive to make during construction or later. The time needed to fix an error in a software architecture is on the same order as that needed to fix a requirements error—that is, more than that needed to fix a coding error (Basili and Perricone 1984, Willis 1998). Architecture changes are like requirements changes in that seemingly small changes can be far-reaching. Whether the architectural changes arise from the need to fix errors or the need to make improvements, the earlier you can identify the changes, the better.

The communication rules for each building block should be well defined. The architecture should describe which other building blocks the building block can use directly, which it can use indirectly, and which it shouldn't use at all.

Major Classes

Cross-Reference For details on class design, see Chapter 6, "Working Classes."

The architecture should specify the major classes to be used. It should identify the responsibilities of each major class and how the class will interact with other classes. It should include descriptions of the class hierarchies, of state transitions, and of object persistence. If the system is large enough, it should describe how classes are organized into subsystems.

The architecture should describe other class designs that were considered and give reasons for preferring the organization that was chosen. The architecture doesn't need to specify every class in the system. Aim for the 80/20 rule: specify the 20 percent of the classes that make up 80 percent of the system's behavior (Jacobsen, Booch, and Rumbaugh 1999; Kruchten 2000).

Data Design

Cross-Reference For details on working with variables, see Chapters 10 through 13.

The architecture should describe the major files and table designs to be used. It should describe alternatives that were considered and justify the choices that were made. If the application maintains a list of customer IDs and the architects have chosen to represent the list of IDs using a sequential-access list, the document should explain why a sequential-access list is better than a random-access list, stack, or hash table. During construction, such information gives you insight into the minds of the architects. During maintenance, the same insight is an invaluable aid. Without it, you're watching a foreign movie with no subtitles.

Data should normally be accessed directly by only one subsystem or class, except through access classes or routines that allow access to the data in controlled and abstract ways. This is explained in more detail in "Hide Secrets (Information Hiding)" in Section 5.3.

The architecture should specify the high-level organization and contents of any databases used. The architecture should explain why a single database is preferable to multiple databases (or vice versa), explain why a database is preferable to flat files, identify possible interactions with other programs that access the same data, explain what views have been created on the data, and so on.

Business Rules

If the architecture depends on specific business rules, it should identify them and describe the impact the rules have on the system's design. For example, suppose the system is required to follow a business rule that customer information should be no

Reuse Decisions

If the plan calls for using preexisting software, test cases, data formats, or other materials, the architecture should explain how the reused software will be made to conform to the other architectural goals—if it will be made to conform.

Change Strategy

Cross-Reference For details on handling changes systematically, see Section 28.2, "Configuration Management."

Because building a software product is a learning process for both the programmers and the users, the product is likely to change throughout its development. Changes arise from volatile data types and file formats, changed functionality, new features, and so on. The changes can be new capabilities likely to result from planned enhancements, or they can be capabilities that didn't make it into the first version of the system. Consequently, one of the major challenges facing a software architect is making the architecture flexible enough to accommodate likely changes.

Design bugs are often subtle and occur by evolution with early assumptions being forgotten as new features or uses are added to a system.
—*Fernando J. Corbató*

The architecture should clearly describe a strategy for handling changes. The architecture should show that possible enhancements have been considered and that the enhancements most likely are also the easiest to implement. If changes are likely in input or output formats, style of user interaction, or processing requirements, the architecture should show that the changes have all been anticipated and that the effects of any single change will be limited to a small number of classes. The architecture's plan for changes can be as simple as one to put version numbers in data files, reserve fields for future use, or design files so that you can add new tables. If a code generator is being used, the architecture should show that the anticipated changes are within the capabilities of the code generator.

Cross-Reference For a full explanation of delaying commitment, see "Choose Binding Time Consciously" in Section 5.3.

The architecture should indicate the strategies that are used to delay commitment. For example, the architecture might specify that a table-driven technique be used rather than hard-coded *if* tests. It might specify that data for the table is to be kept in an external file rather than coded inside the program, thus allowing changes in the program without recompiling.

General Architectural Quality

Cross-Reference For more information about how quality attributes interact, see Section 20.1, "Characteristics of Software Quality."

A good architecture specification is characterized by discussions of the classes in the system, of the information that's hidden in each class, and of the rationales for including and excluding all possible design alternatives.

The architecture should be a polished conceptual whole with few ad hoc additions. The central thesis of the most popular software-engineering book ever, *The Mythical Man-Month*, is that the essential problem with large systems is maintaining their conceptual integrity (Brooks 1995). A good architecture should fit the problem. When you look at the architecture, you should be pleased by how natural and easy the solution seems. It shouldn't look as if the problem and the architecture have been forced together with duct tape.

Part VII
Software Craftsmanship

Index

B

D

E

F

J

K

L

Q

R

S

W

Z