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 were prevalent in the nineties. Because these practices aren't often reported outside the pages of highly specialized technical journals, however, most programming organizations aren't yet using them in the nineties. 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 lifecycle, it makes powerful software development techniques understandable to self-taught programmers as well as to programmers with formal training.

Self-Taught Programmers

If you haven't had much formal training, you're in good company. About 50,000 new programmers enter the profession each year (BLS 2002), 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 most programmers don't receive a formal education in software development. Many self-taught programmers are found in the emerging group of professionals—engineers, accountants, teachers, scientists, 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

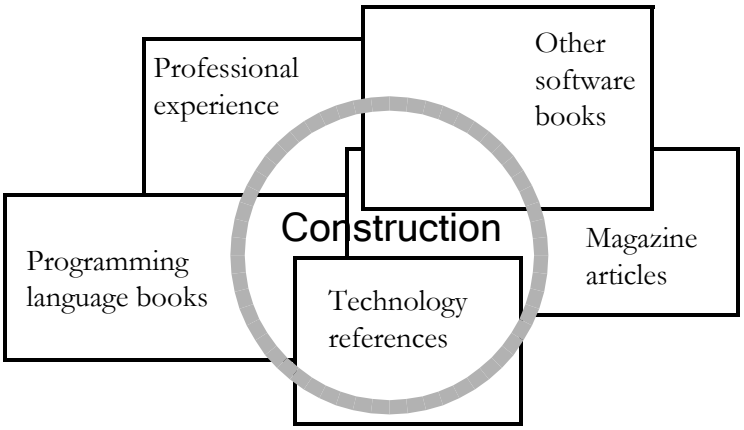
The counterpoint to the programmer with experience but little formal training is the fresh college graduate. The recent graduate is often rich in theoretical knowledge but poor in the practical know-how that goes into building production programs. The practical lore of good coding is often passed down 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 reside 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, Windows or Linux operating-system functions, or about the details of 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.



F00xx01

Figure 1

The information in this book is distilled from many sources.

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 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 dozens of software-engineering books and the hundreds of 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, 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.

In order 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.

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 history of computer programming provides more insight into the particular need for a handbook on software construction.

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.

Construction typically makes up about 80 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 this topic.)

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).

Small-scale coding errors might be less expensive to fix than errors in requirements or architecture, but 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

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 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 ago or more 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.

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.

Book Website

CC2E.COM/1234

Updated checklists, recommended reading, web links, and other content are provided on a companion website at *www.cc2e.com*. To access information related to *Code Complete, 2d Ed.*, enter *cc2e.com/* followed by the four-digit code, as shown in the left margin and throughout the book.

Author Note

If you have any comments, please feel free to contact me care of Microsoft Press, on the Internet as stevemcc@construx.com, or at my Web site at *www.stevemcconnell.com.*

Bellevue, Washington
New Year's Day, 2004

Notes about the Second Edition

When I wrote *Code Complete, First Edition*, I knew that programmers needed a comprehensive book on software construction. I thought a well-written book could sell twenty to thirty thousand copies. In my wildest fantasies (and my fantasies were pretty wild), I thought sales might approach one hundred thousand copies.

Ten years later, I find that *CC1* has sold more than a quarter million copies in English and has been translated into more than a dozen languages. The success of the book has been a pleasant surprise.

Comparing and contrasting the two editions seems like it might produce some insights into the broader world of software development, so here are some thoughts about the second edition in a Q&A format.

Why did you write a second edition? Weren't the principles in the first edition supposed to be timeless?

I've been telling people for years that the principles in the first edition were still 95 percent relevant, even though the cosmetics, such as the specific programming languages used to illustrate the points, had gotten out of date. I knew that the old-fashioned languages used in the examples made the book inaccessible to many readers.

Of course my understanding of software construction had improved and evolved significantly since I published the first edition manuscript in early 1993. After I published *CC1* in 1993, I didn't read it again until early 2003. During that 10 year period, subconsciously I had been thinking that *CC1* was evolving as my thinking was evolving, but of course it wasn't. As I got into detailed work on the second edition, I found that the "cosmetic" problems ran deeper than I had thought. *CC1* was essentially a time capsule of programming practices circa 1993. Industry terminology had evolved, programming languages had evolved, my thinking had evolved, but for some reason the words on the page had not.

After working through the second edition, I still think the principles in the first edition were about 95 percent on target. But the book also needed to address new content above and beyond the 95 percent, so the cosmetic work turned out to be more like reconstructive surgery than a simple makeover.

Does the second edition discuss object-oriented programming?

Object-oriented programming was really just creeping into production coding practice when I was writing *CC1* in 1989-1993. Since then, OO has been absorbed into mainstream programming practice to such an extent that talking about "OO" these days really amounts just to talking about programming. That change is reflected throughout *CC2*. The languages used in *CC2* are all OO (C++, Java, and Visual Basic). One of the major ways that programming has changed since the early 1990s is that a programmer's basic thought unit is now the classes, whereas 10 years ago the basic thought unit was individual routines. That change has rippled throughout the book as well.

What about extreme programming and agile development? Do you talk about those approaches?

It's easiest to answer that question by first saying a bit more about OO. In the early 1990s, OO represented a truly new way of looking at software. As such, I think some time was needed to see how that new approach was going to pan out.

Extreme programming and agile development are unlike OO in that they don't introduce new practices as much as they shift the emphasis that traditional software engineering used to place on some specific practices. They emphasize practices like frequent releases, refactoring, test-first development, and frequent replanning, and de-emphasize other practices like up-front planning, up-front design, and paper documentation.

CC1 addressed many topics that would be called "agile" today. For example, here's what I said about planning in the first edition:

> *"The purpose of planning is to make sure that nobody starves or freezes during the trip; it isn't to map out each step in advance. The plan is to embrace the unexpected and capitalize on unforeseen opportunities. It's a good approach to a market characterized by rapidly changing tools, personnel, and standards of excellence."*

Much of the agile movement originates from where *CC1* left off. For example, here's what I said about agile approaches in 1993:

> *"Evolution during development is an issue that hasn't received much attention in its own right. With the rise of code-centered approaches such as prototyping and evolutionary delivery, it's likely to receive an increasing amount of attention."*

> *"The word "incremental" has never achieved the designer status of "structured" or "object-oriented," so no one has ever written a book on "incremental software engineering." That's too bad because the collection of techniques in such a book would be exceptionally potent."*

Of course evolutionary and incremental development approaches have become the backbone of agile development.

What size project will benefit from Code Complete, Second Edition?
Both large and small projects will benefit from *Code Complete*, as will business-systems projects, safety-critical projects, games, scientific and engineering applications—but these different kinds of projects will emphasize different practices. The idea that different practices apply to different kinds of software is one of the least understood ideas in software development. Indeed, it appears not to be understood by many of the people writing software development books. Fortunately, good construction practices have more in common across types of software than do good requirements, architecture, testing, and quality assurance practices. So *Code Complete* can be more applicable to multiple project types than books on other software development topics could be.

Have there been any improvements in programming in the past 10 years?
Programming tools have advanced by leaps and bounds. The tool that I described as a panacea in 1993 is commonplace today.

Computing power has advanced extraordinarily. In the performance tuning chapters, *CC2*'s disk access times are comparable to *CC1*'s in-memory access times, which is a staggering improvement. As computers become more powerful, it makes sense to have the computer do more of the construction work.

CC1's discussion of non-waterfall lifecycle models was mostly theoretical—the best organizations were using them, but most were using either code and fix or the waterfall model. Now incremental, evolutionary development approaches are in the mainstream. I still see most organizations using code and fix, but at least the organizations that aren't using code and fix are using something better than the waterfall model.

There has also been an amazing explosion of good software development books. When I wrote the first edition in 1989-1993, I think it was still possible for a motivated software developer to read every significant book in the field. Today I think it would be a challenge even to read every good book on one significant topic like design, requirements, or management. There still aren't a lot of other good books on construction, though.

Has anything moved backwards?

There are still far more people who talk about good practices than who actually use good practices. I see far too many people using current buzzwords as a cloak for sloppy practices. When the first edition was published, people were claiming, "I don't have to do requirements or design because I'm using object-oriented programming." That was just an excuse. Most of those people weren't really doing object-oriented programming—they were hacking, and the results were predictable, and poor. Right now, people are saying "I don't have to do requirements or design because I'm doing agile development." Again, the results are easy to predict, and poor.

Testing guru Boris Beizer said that his clients ask him, "How can I revolutionize and transform my software development without changing anything except the names and putting some slogans up on the walls?" (Johnson 1994b). Good programmers invest the effort to learn how to use current practices. Not-so-good programmers just learn the buzzwords, and that's been a software industry constant for a half century.

Which of the first edition's ideas are you most protective of?

I'm protective of the construction metaphor and the toolbox metaphor. Some writers have criticized the construction metaphor as not being well-suited to software, but most of those writers seem to have simplistic understandings of construction (You can see how I've responded to those criticisms in Chapter 2.)

The toolbox metaphor is becoming more critical as software continues to weave itself into every fiber of our lives. Understanding that different tools will work best for different kinds of jobs is critical to not using an axe to cut a stick of butter and not using a butter knife to chop down a tree. It's silly to hear people criticize software axes for being too bureaucratic when they should have chosen butter knives instead. Axes are good, and so are butter knives, but you need to know what each is used for. In software, we still see people using practices that are good practices in the right context but that are not well suited for every single task.

Will there be a third edition 10 years from now?

I'm tired of answering questions. Let's get on with the book!

1

Welcome to Software Construction

CC2E.COM/0178

Contents

Related Topics

Who should read the 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

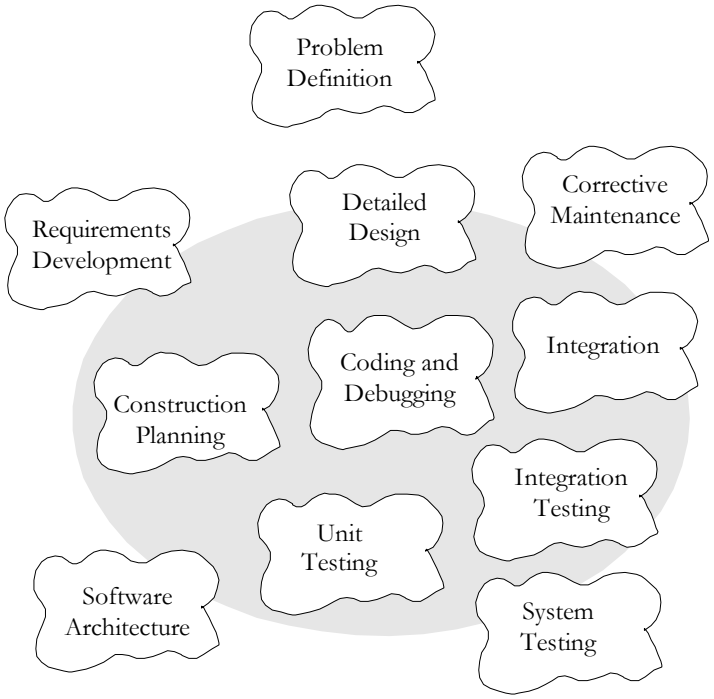
- Problem definition
- Requirements development
- Construction planning
- Software architecture, or high-level design

- 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 in a later chapter.

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.



F01xx01

Figure 1-1

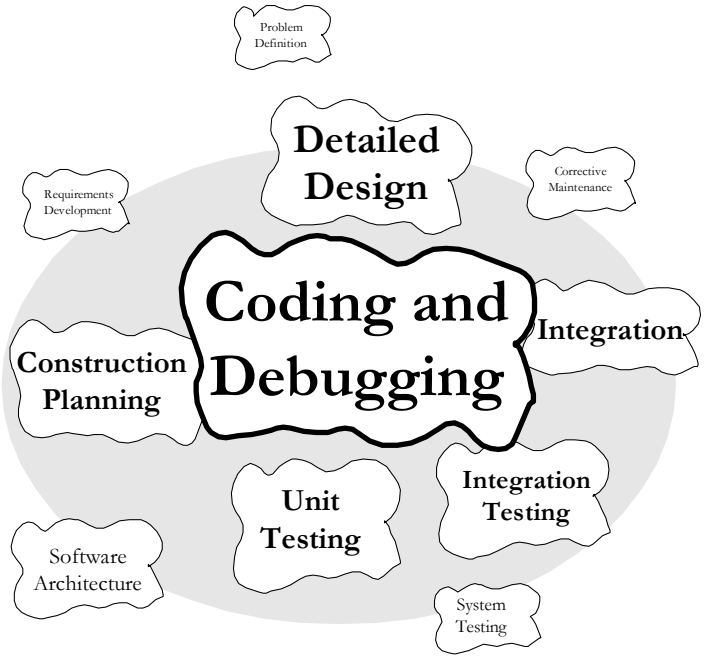
Construction activities are shown inside the gray circle. Construction focuses on coding and debugging but also includes some detailed design, unit testing, integration testing and other activities.

KEY POINT

As the figure indicates, construction is mostly coding and debugging but also involves elements of detailed design, 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 l-1’s flat-earth view of software development, Figure 1-2 shows the round-earth perspective of this book.



F01xx02

Figure 1-2

This book focuses on detailed design, coding, debugging, and unit testing 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 smaller and faster

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* parts 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 the "Where to Find More Information" chapter 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, the space program, 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:

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.

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 the central activity in software development
Requirements and architecture are done before construction so that you can do construction effectively. System 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 books 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, then 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 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, and developer testing.
- Other common terms for construction are "coding and debugging" 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.

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 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's thinking that the sun moves around the earth. Similarly, it's difficult to imagine anyone's 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 roadmap. 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 is 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. Fred Brooks says that writing software is like farming, hunting werewolves, or drowning with dinosaurs in a tar pit (1995). David Gries says it's 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 (Plauger 1993, Beck 2000). Alistair Cockburn says it's a game (2001). Eric Raymond says it's like a bazaar (2000). Paul Heckel says it's like filming Snow White and the Seven Dwarfs (1994). 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."

KEY POINT

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.

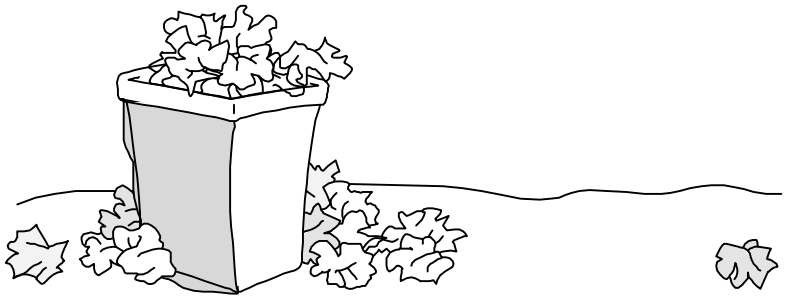
Plan to throw one away; you will, anyhow.

— 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. Planning to throw one away might be practical when you're writing a polite how-do-you-do to your aunt, and it might have been state-of-the-art software engineering practice in 1975, when Brooks wrote his book.



F02xx01

Figure 2-1

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