Microsoft

# Code Complete, Second Edition

*Steve McConnell*

# Contents at a Glance


**Part I** Laying the Foundation

1 Welcome to Software Construction . . . . . 3
2 Metaphors for a Richer Understanding of Software Development . . . . . 9
3 Measure Twice, Cut Once: Upstream Prerequisites. . . . . 23
4 Key Construction Decisions . . . . . 61

**Part II** Creating High-Quality Code

5 Design in Construction . . . . . 73
6 Working Classes . . . . . 125
7 High-Quality Routines. . . . . 161
8 Defensive Programming . . . . . 187
9 The Pseudocode Programming Process . . . . . 215

**Part III** Variables

10 General Issues in Using Variables. . . . . 237
11 The Power of Variable Names . . . . . 259
12 Fundamental Data Types . . . . . 291
13 Unusual Data Types . . . . . 319

**Part IV** Statements

14 Organizing Straight-Line Code. . . . . 347
15 Using Conditionals. . . . . 355
16 Controlling Loops . . . . . 367
17 Unusual Control Structures. . . . . 391
18 Table-Driven Methods. . . . . 411
19 General Control Issues. . . . . 431

Part V Code Improvements
20 The Software-Quality Landscape . . . 463
21 Collaborative Construction . . . 479
22 Developer Testing . . . 499
23 Debugging . . . 535
24 Refactoring . . . 563
25 Code-Tuning Strategies . . . 587
26 Code-Tuning Techniques . . . 609

Part VI System Considerations
27 How Program Size Affects Construction . . . 649
28 Managing Construction . . . 661
29 Integration . . . 689
30 Programming Tools . . . 709

Part VII Software Craftsmanship
31 Layout and Style . . . 729
32 Self-Documenting Code . . . 777
33 Personal Character . . . 819
34 Themes in Software Craftsmanship . . . 837
35 Where to Find More Information . . . 855

# Table of Contents

**Preface** . . . . . **xix**
**Acknowledgments** . . . . . **xxvii**
**List of Checklists** . . . . . **xxix**
**List of Tables** . . . . . **xxxi**
**List of Figures** . . . . . **xxxiii**

**Part I** **Laying the Foundation**

**1 Welcome to Software Construction . . . . . 3**
1.1 What Is Software Construction? . . . . . 3
1.2 Why Is Software Construction Important? . . . . . 6
1.3 How to Read This Book . . . . . 8

**2 Metaphors for a Richer Understanding of Software Development . . . . . 9**
2.1 The Importance of Metaphors . . . . . 9
2.2 How to Use Software Metaphors . . . . . 11
2.3 Common Software Metaphors . . . . . 13

**3 Measure Twice, Cut Once: Upstream Prerequisites . . . . . 23**
3.1 Importance of Prerequisites . . . . . 24
3.2 Determine the Kind of Software You're Working On . . . . . 31
3.3 Problem-Definition Prerequisite . . . . . 36
3.4 Requirements Prerequisite . . . . . 38
3.5 Architecture Prerequisite . . . . . 43
3.6 Amount of Time to Spend on Upstream Prerequisites . . . . . 55

**4 Key Construction Decisions . . . . . 61**
4.1 Choice of Programming Language . . . . . 61
4.2 Programming Conventions . . . . . 66
4.3 Your Location on the Technology Wave . . . . . 66
4.4 Selection of Major Construction Practices . . . . . 69

## Part II Creating High-Quality Code


**5 Design in Construction . . . . . 73**

5.1 Design Challenges . . . . . 74
5.2 Key Design Concepts . . . . . 77
5.3 Design Building Blocks: Heuristics . . . . . 87
5.4 Design Practices . . . . . 110
5.5 Comments on Popular Methodologies . . . . . 118

**6 Working Classes . . . . . 125**

6.1 Class Foundations: Abstract Data Types (ADTs) . . . . . 126
6.2 Good Class Interfaces . . . . . 133
6.3 Design and Implementation Issues . . . . . 143
6.4 Reasons to Create a Class . . . . . 152
6.5 Language-Specific Issues . . . . . 156
6.6 Beyond Classes: Packages . . . . . 156

**7 High-Quality Routines . . . . . 161**

7.1 Valid Reasons to Create a Routine . . . . . 164
7.2 Design at the Routine Level . . . . . 168
7.3 Good Routine Names . . . . . 171
7.4 How Long Can a Routine Be? . . . . . 173
7.5 How to Use Routine Parameters . . . . . 174
7.6 Special Considerations in the Use of Functions . . . . . 181
7.7 Macro Routines and Inline Routines . . . . . 182

**8 Defensive Programming . . . . . 187**

8.1 Protecting Your Program from Invalid Inputs . . . . . 188
8.2 Assertions . . . . . 189
8.3 Error-Handling Techniques . . . . . 194
8.4 Exceptions . . . . . 198
8.5 Barricade Your Program to Contain the Damage Caused by Errors . . . . . 203
8.6 Debugging Aids . . . . . 205
8.7 Determining How Much Defensive Programming to Leave in Production Code . . . . . 209
8.8 Being Defensive About Defensive Programming . . . . . 210

**9 The Pseudocode Programming Process . . . . . 215**

9.1 Summary of Steps in Building Classes and Routines . . . . . 216
9.2 Pseudocode for Pros . . . . . 218
9.3 Constructing Routines by Using the PPP . . . . . 220
9.4 Alternatives to the PPP . . . . . 232

**Part III Variables**

**10 General Issues in Using Variables. . . . . 237**

10.1 Data Literacy . . . . . 238
10.2 Making Variable Declarations Easy . . . . . 239
10.3 Guidelines for Initializing Variables . . . . . 240
10.4 Scope . . . . . 244
10.5 Persistence . . . . . 251
10.6 Binding Time . . . . . 252
10.7 Relationship Between Data Types and Control Structures . . . . . 254
10.8 Using Each Variable for Exactly One Purpose . . . . . 255

**11 The Power of Variable Names . . . . . 259**

11.1 Considerations in Choosing Good Names . . . . . 259
11.2 Naming Specific Types of Data . . . . . 264
11.3 The Power of Naming Conventions . . . . . 270
11.4 Informal Naming Conventions . . . . . 272
11.5 Standardized Prefixes . . . . . 279
11.6 Creating Short Names That Are Readable . . . . . 282
11.7 Kinds of Names to Avoid . . . . . 285

**12 Fundamental Data Types . . . . . 291**

12.1 Numbers in General . . . . . 292
12.2 Integers . . . . . 293
12.3 Floating-Point Numbers . . . . . 295
12.4 Characters and Strings . . . . . 297
12.5 Boolean Variables . . . . . 301
12.6 Enumerated Types . . . . . 303
12.7 Named Constants . . . . . 307
12.8 Arrays . . . . . 310
12.9 Creating Your Own Types (Type Aliasing) . . . . . 311

19.3 Null Statements . . . . . 444
19.4 Taming Dangerously Deep Nesting . . . . . 445
19.5 A Programming Foundation: Structured Programming . . . . . 454
19.6 Control Structures and Complexity . . . . . 456

**Part V Code Improvements**

**20 The Software-Quality Landscape . . . . . 463**

20.1 Characteristics of Software Quality . . . . . 463
20.2 Techniques for Improving Software Quality . . . . . 466
20.3 Relative Effectiveness of Quality Techniques . . . . . 469
20.4 When to Do Quality Assurance . . . . . 473
20.5 The General Principle of Software Quality . . . . . 474

**21 Collaborative Construction . . . . . 479**

21.1 Overview of Collaborative Development Practices . . . . . 480
21.2 Pair Programming . . . . . 483
21.3 Formal Inspections . . . . . 485
21.4 Other Kinds of Collaborative Development Practices . . . . . 492

**22 Developer Testing . . . . . 499**

22.1 Role of Developer Testing in Software Quality . . . . . 500
22.2 Recommended Approach to Developer Testing . . . . . 503
22.3 Bag of Testing Tricks . . . . . 505
22.4 Typical Errors . . . . . 517
22.5 Test-Support Tools . . . . . 523
22.6 Improving Your Testing . . . . . 528
22.7 Keeping Test Records . . . . . 529

**23 Debugging . . . . . 535**

23.1 Overview of Debugging Issues . . . . . 535
23.2 Finding a Defect . . . . . 540
23.3 Fixing a Defect . . . . . 550
23.4 Psychological Considerations in Debugging . . . . . 554
23.5 Debugging Tools—Obvious and Not-So-Obvious . . . . . 556

**24 Refactoring . . . . . 563**

24.1 Kinds of Software Evolution . . . . . 564
24.2 Introduction to Refactoring . . . . . 565
24.3 Specific Refactorings . . . . . 571
24.4 Refactoring Safely . . . . . 579
24.5 Refactoring Strategies . . . . . 582

**25 Code-Tuning Strategies . . . . . 587**

25.1 Performance Overview . . . . . 588
25.2 Introduction to Code Tuning . . . . . 591
25.3 Kinds of Fat and Molasses . . . . . 597
25.4 Measurement . . . . . 603
25.5 Iteration . . . . . 605
25.6 Summary of the Approach to Code Tuning . . . . . 606

**26 Code-Tuning Techniques . . . . . 609**

26.1 Logic . . . . . 610
26.2 Loops . . . . . 616
26.3 Data Transformations . . . . . 624
26.4 Expressions . . . . . 630
26.5 Routines . . . . . 639
26.6 Recoding in a Low-Level Language . . . . . 640
26.7 The More Things Change, the More They Stay the Same . . . . . 643

**Part VI System Considerations**

**27 How Program Size Affects Construction . . . . . 649**

27.1 Communication and Size . . . . . 650
27.2 Range of Project Sizes . . . . . 651
27.3 Effect of Project Size on Errors . . . . . 651
27.4 Effect of Project Size on Productivity . . . . . 653
27.5 Effect of Project Size on Development Activities . . . . . 654

**28 Managing Construction . . . . . 661**

28.1 Encouraging Good Coding . . . . . 662
28.2 Configuration Management . . . . . 664
28.3 Estimating a Construction Schedule . . . . . 671
28.4 Measurement . . . . . 677
28.5 Treating Programmers as People . . . . . 680
28.6 Managing Your Manager . . . . . 686

**29 Integration . . . . . 689**

29.1 Importance of the Integration Approach . . . . . 689
29.2 Integration Frequency—Phased or Incremental? . . . . . 691
29.3 Incremental Integration Strategies . . . . . 694
29.4 Daily Build and Smoke Test . . . . . 702

**30 Programming Tools . . . . . 709**

30.1 Design Tools . . . . . 710
30.2 Source-Code Tools . . . . . 710
30.3 Executable-Code Tools . . . . . 716
30.4 Tool-Oriented Environments . . . . . 720
30.5 Building Your Own Programming Tools . . . . . 721
30.6 Tool Fantasyland . . . . . 722

**Part VII Software Craftsmanship**

**31 Layout and Style . . . . . 729**

31.1 Layout Fundamentals . . . . . 730
31.2 Layout Techniques . . . . . 736
31.3 Layout Styles . . . . . 738
31.4 Laying Out Control Structures . . . . . 745
31.5 Laying Out Individual Statements . . . . . 753
31.6 Laying Out Comments . . . . . 763
31.7 Laying Out Routines . . . . . 766
31.8 Laying Out Classes . . . . . 768

**32 Self-Documenting Code . . . . . 777**

32.1 External Documentation . . . . . 777
32.2 Programming Style as Documentation . . . . . 778
32.3 To Comment or Not to Comment . . . . . 781
32.4 Keys to Effective Comments . . . . . 785
32.5 Commenting Techniques . . . . . 792
32.6 IEEE Standards . . . . . 813

**33 Personal Character . . . . . 819**

33.1 Isn't Personal Character Off the Topic? . . . . . 820
33.2 Intelligence and Humility . . . . . 821
33.3 Curiosity . . . . . 822
33.4 Intellectual Honesty . . . . . 826
33.5 Communication and Cooperation . . . . . 828
33.6 Creativity and Discipline . . . . . 829
33.7 Laziness . . . . . 830
33.8 Characteristics That Don't Matter As Much As You Might Think . . . . . 830
33.9 Habits . . . . . 833

**34 Themes in Software Craftsmanship . . . . . 837**

34.1 Conquer Complexity . . . . . 837
34.2 Pick Your Process . . . . . 839
34.3 Write Programs for People First, Computers Second . . . . . 841
34.4 Program into Your Language, Not in It . . . . . 843
34.5 Focus Your Attention with the Help of Conventions . . . . . 844
34.6 Program in Terms of the Problem Domain . . . . . 845
34.7 Watch for Falling Rocks . . . . . 848
34.8 Iterate, Repeatedly, Again and Again . . . . . 850
34.9 Thou Shalt Rend Software and Religion Asunder . . . . . 851

**35 Where to Find More Information . . . . . 855**

35.1 Information About Software Construction . . . . . 856
35.2 Topics Beyond Construction . . . . . 857
35.3 Periodicals . . . . . 859
35.4 A Software Developer's Reading Plan . . . . . 860
35.5 Joining a Professional Organization . . . . . 862

**Bibliography . . . . . 863**

**Index . . . . . 885**

# 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.

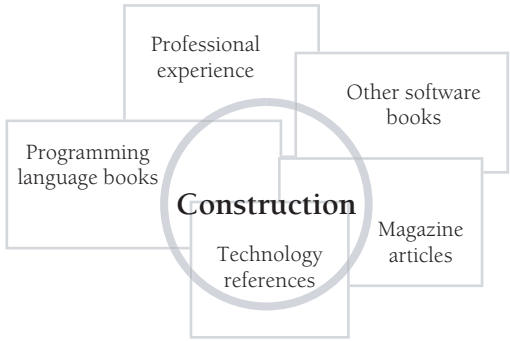
# Where Else Can You Find This Information?

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. Conse-

quently, 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*.

*Bellevue, Washington*
*Memorial Day, 2004*

### Microsoft Learning Technical Support

Every effort has been made to ensure the accuracy of this book. Microsoft Press provides corrections for books through the World Wide Web at the following address:

*http://www.microsoft.com/learning/support/*

To connect directly to the Microsoft Knowledge Base and enter a query regarding a question or issue that you may have, go to:

*http://www.microsoft.com/learning/support/search.asp*

If you have comments, questions, or ideas regarding this book, please send them to Microsoft Press using either of the following methods:

Postal Mail:

*Microsoft Press*
*Attn: Code Complete 2E Editor*
*One Microsoft Way*
*Redmond, WA 98052-6399*

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

Requirements 42
Architecture 54
Upstream Prerequisites 59
Major Construction Practices 69
Design in Construction 122
Class Quality 157
High-Quality Routines 185
Defensive Programming 211
The Pseudocode Programming Process 233
General Considerations In Using Data 257
Naming Variables 288
Fundamental Data 316
Considerations in Using Unusual Data Types 343
Organizing Straight-Line Code 353
Using Conditionals 365
Loops 388
Unusual Control Structures 410
Table-Driven Methods 429
Control-Structure Issues 459
A Quality-Assurance Plan 476
Effective Pair Programming 484
Effective Inspections 491
Test Cases 532
Debugging Reminders 559
Reasons to Refactor 570
Summary of Refactorings 577
Refactoring Safely 584
Code-Tuning Strategies 607
Code-Tuning Techniques 642

Configuration Management 669
Integration 707
Programming Tools 724
Layout 773
Self-Documenting Code 780
Good Commenting Technique 816

# Tables


**Table 3-1** Average Cost of Fixing Defects Based on When They're Introduced and Detected 29

**Table 3-2** Typical Good Practices for Three Common Kinds of Software Projects 31

**Table 3-3** Effect of Skipping Prerequisites on Sequential and Iterative Projects 33

**Table 3-4** Effect of Focusing on Prerequisites on Sequential and Iterative Projects 34

**Table 4-1** Ratio of High-Level-Language Statements to Equivalent C Code 62

**Table 5-1** Popular Design Patterns 104

**Table 5-2** Design Formality and Level of Detail Needed 116

**Table 6-1** Variations on Inherited Routines 145

**Table 8-1** Popular-Language Support for Exceptions 198

**Table 11-1** Examples of Good and Bad Variable Names 261

**Table 11-2** Variable Names That Are Too Long, Too Short, or Just Right 262

**Table 11-3** Sample Naming Conventions for C++ and Java 277

**Table 11-4** Sample Naming Conventions for C 278

**Table 11-5** Sample Naming Conventions for Visual Basic 278

**Table 11-6** Sample of UDTs for a Word Processor 280

**Table 11-7** Semantic Prefixes 280

**Table 12-1** Ranges for Different Types of Integers 294

**Table 13-1** Accessing Global Data Directly and Through Access Routines 341

**Table 13-2** Parallel and Nonparallel Uses of Complex Data 342

**Table 16-1** The Kinds of Loops 368

**Table 19-1** Transformations of Logical Expressions Under DeMorgan's Theorems 436

**Table 19-2** Techniques for Counting the Decision Points in a Routine 458

**Table 20-1** Team Ranking on Each Objective 469

**Table 20-2** Defect-Detection Rates 470

**Table 20-3** Extreme Programming's Estimated Defect-Detection Rate 472

**Table 21-1** Comparison of Collaborative Construction Techniques 495

**Table 23-1** Examples of Psychological Distance Between Variable Names 556

**Table 25-1** Relative Execution Time of Programming Languages 600

**Table 25-2** Costs of Common Operations 601

**Table 27-1** Project Size and Typical Error Density 652

**Table 27-2** Project Size and Productivity 653

**Table 28-1** Factors That Influence Software-Project Effort 674

**Table 28-2** Useful Software-Development Measurements 678

**Table 28-3** One View of How Programmers Spend Their Time 681

# Figures


**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. 4

**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. 5

**Figure 2-1** The letter-writing metaphor suggests that the software process relies on expensive trial and error rather than careful planning and design. 14

**Figure 2-2** It's hard to extend the farming metaphor to software development appropriately. 15

**Figure 2-3** The penalty for a mistake on a simple structure is only a little time and maybe some embarrassment. 17

**Figure 2-4** More complicated structures require more careful planning. 18

**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). 30

**Figure 3-2** Activities will overlap to some degree on most projects, even those that are highly sequential. 35

**Figure 3-3** On other projects, activities will overlap for the duration of the project. One key to successful construction is understanding the degree to which prerequisites have been completed and adjusting your approach accordingly. 35

**Figure 3-4** The problem definition lays the foundation for the rest of the programming process. 37

**Figure 3-5** Be sure you know what you're aiming at before you shoot. 38

**Figure 3-6** Without good requirements, you can have the right general problem but miss the mark on specific aspects of the problem. 39

**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. 44

**Figure 5-1** The Tacoma Narrows bridge—an example of a wicked problem. 75

**Figure 5-2** The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5). 82

**Figure 5-3** An example of a system with six subsystems. 83

**Figure 5-4** An example of what happens with no restrictions on intersubsystem communications. 83

**Figure 5-5** With a few communication rules, you can simplify subsystem interactions significantly. 84

**Figure 5-6** This billing system is composed of four major objects. The objects have been simplified for this example. 88

**Figure 5-7** Abstraction allows you to take a simpler view of a complex concept. 90

**Figure 5-8** Encapsulation says that, not only are you allowed to take a simpler view of a complex concept, you are not allowed to look at any of the details of the complex concept. What you see is what you get—it's all you get! 91

**Figure 5-9** A good class interface is like the tip of an iceberg, leaving most of the class unexposed. 93

**Figure 5-10** G. Polya developed an approach to problem solving in mathematics that's also useful in solving problems in software design (Polya 1957). 109

**Figure 8-1** Part of the Interstate-90 floating bridge in Seattle sank during a storm because the flotation tanks were left uncovered, they filled with water, and the bridge became too heavy to float. During construction, protecting yourself against the small stuff matters more than you might think. 189

**Figure 8-2** Defining some parts of the software that work with dirty data and some that work with clean data can be an effective way to relieve the majority of the code of the responsibility for checking for bad data. 204

**Figure 9-1** Details of class construction vary, but the activities generally occur in the order shown here. 216

**Figure 9-2** These are the major activities that go into constructing a routine. They're usually performed in the order shown. 217

**Figure 9-3** You'll perform all of these steps as you design a routine but not necessarily in any particular order. 225

**Figure 10-1** "Long live time" means that a variable is live over the course of many statements. "Short live time" means it's live for only a few statements. "Span" refers to how close together the references to a variable are. 246

**Figure 10-2** Sequential data is data that's handled in a defined order. 254

**Figure 10-3** Selective data allows you to use one piece or the other, but not both. 255

**Figure 10-4** Iterative data is repeated. 255

**Figure 13-1** The amount of memory used by each data type is shown by double lines. 324

**Figure 13-2** An example of a picture that helps us think through the steps involved in relinking pointers. 329

**Figure 14-1** If the code is well organized into groups, boxes drawn around related sections don't overlap. They might be nested. 352

**Figure 14-2** If the code is organized poorly, boxes drawn around related sections overlap. 353

**Figure 17-1** Recursion can be a valuable tool in the battle against complexity—when used to attack suitable problems. 394

**Figure 18-1** As the name suggests, a direct-access table allows you to access the table element you're interested in directly. 413

**Figure 18-2** Messages are stored in no particular order, and each one is identified with a message ID. 417

**Figure 18-3** Aside from the Message ID, each kind of message has its own format. 418

**Figure 18-4** Rather than being accessed directly, an indexed access table is accessed via an intermediate index. 425

**Figure 18-5** The stair-step approach categorizes each entry by determining the level at which it hits a "staircase." The "step" it hits determines its category. 426

**Figure 19-1** Examples of using number-line ordering for boolean tests. 440

**Figure 20-1** Focusing on one external characteristic of software quality can affect other characteristics positively, adversely, or not at all. 466

**Figure 20-2** Neither the fastest nor the slowest development approach produces the software with the most defects. 475

**Figure 22-1** As the size of the project increases, developer testing consumes a smaller percentage of the total development time. The effects of program size are described in more detail in Chapter 27, "How Program Size Affects Construction." 502

**Figure 22-2** As the size of the project increases, the proportion of errors committed during construction decreases. Nevertheless, construction errors account for 45–75% of all errors on even the largest projects. 521

**Figure 23-1** Try to reproduce an error several different ways to determine its exact cause. 545

**Figure 24-1** Small changes tend to be more error-prone than larger changes (Weinberg 1983). 581

**Figure 24-2** Your code doesn't have to be messy just because the real world is messy. Conceive your system as a combination of ideal code, interfaces from the ideal code to the messy real world, and the messy real world. 583

**Figure 24-3** One strategy for improving production code is to refactor poorly written legacy code as you touch it, so as to move it to the other side of the "interface to the messy real world." 584

**Figure 27-1** The number of communication paths increases proportionate to the square of the number of people on the team. 650

**Figure 27-2** As project size increases, errors usually come more from requirements and design. Sometimes they still come primarily from construction (Boehm 1981, Grady 1987, Jones 1998). 652

**Figure 27-3** Construction activities dominate small projects. Larger projects require more architecture, integration work, and system testing to succeed. Requirements work is not shown on this diagram because requirements effort is not as directly a function of program size as other activities are (Albrecht 1979; Glass 1982; Boehm, Gray, and Seewaldt 1984; Boddie 1987; Card 1987; McGarry, Waligora, and McDermott 1989; Brooks 1995; Jones 1998; Jones 2000; Boehm et al. 2000). 654

**Figure 27-4** The amount of software construction work is a near-linear function of project size. Other kinds of work increase nonlinearly as project size increases. 655

**Figure 28-1** This chapter covers the software-management topics related to construction. 661

**Figure 28-2** Estimates created early in a project are inherently inaccurate. As the project progresses, estimates can become more accurate. Reestimate periodically throughout a project, and use what you learn during each activity to improve your estimate for the next activity. 673

**Figure 29-1** The football stadium add-on at the University of Washington collapsed because it wasn't strong enough to support itself during construction. It likely would have been strong enough when completed, but it was constructed in the wrong order—an integration error. 690

**Figure 29-2** Phased integration is also called "big bang" integration for a good reason! 691

**Figure 29-3** Incremental integration helps a project build momentum, like a snowball going down a hill. 692