

Microsoft®

# CODE COMPLETE

# 2

Second Edition



A practical handbook of software construction

**Steve McConnell**

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

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

Copyright © 2004 by Steven C. McConnell

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Cataloging-in-Publication Data  
McConnell, Steve

Code Complete / Steve McConnell.--2nd ed.

p. cm.

Includes index.

ISBN 0-7356-1967-0

1. Computer Software--Development--Handbooks, manuals, etc. I. Title.

QA76.76.D47M39 2004

005.1--dc22

2004049981

Printed and bound in the United States of America.

ISBN: 978-0-7356-1967-8

Twenty-fourth Printing: February 2015

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

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

Microsoft, Microsoft Press, PowerPoint, Visual Basic, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

**Acquisitions Editors:** Linda Engelman and Robin Van Steenburgh

**Project Editor:** Devon Musgrave

**Indexer:** Bill Myers

**Principal Desktop Publisher:** Carl Diltz

Body Part No. X10-53130

[2014-03-21

# Code Complete, Second Edition

*Steve McConnell*

# Contents at a Glance

<b>Part I</b>	<b>Laying the Foundation</b>	
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
<b>Part II</b>	<b>Creating High-Quality Code</b>	
5	Design in Construction . . . . .	73
6	Working Classes . . . . .	125
7	High-Quality Routines. . . . .	161
8	Defensive Programming . . . . .	187
9	The Pseudocode Programming Process. . . . .	215
<b>Part III</b>	<b>Variables</b>	
10	General Issues in Using Variables. . . . .	237
11	The Power of Variable Names . . . . .	259
12	Fundamental Data Types . . . . .	291
13	Unusual Data Types . . . . .	319
<b>Part IV</b>	<b>Statements</b>	
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 .....	xxxix
List of Figures .....	xxxiii

## Part I Laying the Foundation

<b>1</b>	<b>Welcome to Software Construction .....</b>	<b>3</b>
	1.1 What Is Software Construction? .....	3
	1.2 Why Is Software Construction Important? .....	6
	1.3 How to Read This Book .....	8
<b>2</b>	<b>Metaphors for a Richer Understanding of Software Development .....</b>	<b>9</b>
	2.1 The Importance of Metaphors .....	9
	2.2 How to Use Software Metaphors .....	11
	2.3 Common Software Metaphors .....	13
<b>3</b>	<b>Measure Twice, Cut Once: Upstream Prerequisites .....</b>	<b>23</b>
	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
<b>4</b>	<b>Key Construction Decisions .....</b>	<b>61</b>
	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

**What do you think of this book?**  
We want to hear from you!

Microsoft is interested in hearing your feedback about this publication so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit: [www.microsoft.com/learning/booksurvey/](http://www.microsoft.com/learning/booksurvey/)

**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

<b>9</b>	<b>The Pseudocode Programming Process</b> . . . . .	<b>215</b>
	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
<b>Part III</b>	<b>Variables</b>	
<b>10</b>	<b>General Issues in Using Variables</b> . . . . .	<b>237</b>
	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
<b>11</b>	<b>The Power of Variable Names</b> . . . . .	<b>259</b>
	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
<b>12</b>	<b>Fundamental Data Types</b> . . . . .	<b>291</b>
	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



<b>13</b>	<b>Unusual Data Types</b> . . . . .	<b>319</b>
	13.1 Structures . . . . .	319
	13.2 Pointers . . . . .	323
	13.3 Global Data . . . . .	335
<b>Part IV Statements</b>		
<b>14</b>	<b>Organizing Straight-Line Code</b> . . . . .	<b>347</b>
	14.1 Statements That Must Be in a Specific Order . . . . .	347
	14.2 Statements Whose Order Doesn't Matter . . . . .	351
<b>15</b>	<b>Using Conditionals</b> . . . . .	<b>355</b>
	15.1 <i>if</i> Statements . . . . .	355
	15.2 <i>case</i> Statements . . . . .	361
<b>16</b>	<b>Controlling Loops</b> . . . . .	<b>367</b>
	16.1 Selecting the Kind of Loop . . . . .	367
	16.2 Controlling the Loop . . . . .	373
	16.3 Creating Loops Easily—From the Inside Out . . . . .	385
	16.4 Correspondence Between Loops and Arrays . . . . .	387
<b>17</b>	<b>Unusual Control Structures</b> . . . . .	<b>391</b>
	17.1 Multiple Returns from a Routine . . . . .	391
	17.2 Recursion . . . . .	393
	17.3 <i>goto</i> . . . . .	398
	17.4 Perspective on Unusual Control Structures . . . . .	408
<b>18</b>	<b>Table-Driven Methods</b> . . . . .	<b>411</b>
	18.1 General Considerations in Using Table-Driven Methods . . . . .	411
	18.2 Direct Access Tables . . . . .	413
	18.3 Indexed Access Tables . . . . .	425
	18.4 Stair-Step Access Tables . . . . .	426
	18.5 Other Examples of Table Lookups . . . . .	429
<b>19</b>	<b>General Control Issues</b> . . . . .	<b>431</b>
	19.1 Boolean Expressions . . . . .	431
	19.2 Compound Statements (Blocks) . . . . .	443

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**

<b>20</b>	<b>The Software-Quality Landscape . . . . .</b>	<b>463</b>
	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
<b>21</b>	<b>Collaborative Construction . . . . .</b>	<b>479</b>
	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
<b>22</b>	<b>Developer Testing . . . . .</b>	<b>499</b>
	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
<b>23</b>	<b>Debugging . . . . .</b>	<b>535</b>
	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

<b>24</b>	<b>Refactoring</b> . . . . .	<b>563</b>
	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
<b>25</b>	<b>Code-Tuning Strategies.</b> . . . . .	<b>587</b>
	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
<b>26</b>	<b>Code-Tuning Techniques</b> . . . . .	<b>609</b>
	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**

<b>27</b>	<b>How Program Size Affects Construction</b> . . . . .	<b>649</b>
	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

<b>28</b>	<b>Managing Construction</b> .....	<b>661</b>
	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
<b>29</b>	<b>Integration</b> .....	<b>689</b>
	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
<b>30</b>	<b>Programming Tools</b> .....	<b>709</b>
	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
<b>Part VII Software Craftsmanship</b>		
<b>31</b>	<b>Layout and Style</b> .....	<b>729</b>
	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

<b>32</b>	<b>Self-Documenting Code</b> . . . . .	<b>777</b>
	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
<b>33</b>	<b>Personal Character</b> . . . . .	<b>819</b>
	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
<b>34</b>	<b>Themes in Software Craftsmanship</b> . . . . .	<b>837</b>
	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**

**What do you think of this book?**  
 We want to hear from you!

Microsoft is interested in hearing your feedback about this publication so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit: [www.microsoft.com/learning/booksurvey/](http://www.microsoft.com/learning/booksurvey/)



# 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.

Copyrighted image

## 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](mailto:stevemcc@construx.com), or visit my website at [www.stevemccconnell.com](http://www.stevemccconnell.com).

Copyrighted image

**E-mail.**

*[mspinput@microsoft.com](mailto: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

<b>Table 3-1</b>	Average Cost of Fixing Defects Based on When They're Introduced and Detected	29
<b>Table 3-2</b>	Typical Good Practices for Three Common Kinds of Software Projects	31
<b>Table 3-3</b>	Effect of Skipping Prerequisites on Sequential and Iterative Projects	33
<b>Table 3-4</b>	Effect of Focusing on Prerequisites on Sequential and Iterative Projects	34
<b>Table 4-1</b>	Ratio of High-Level-Language Statements to Equivalent C Code	62
<b>Table 5-1</b>	Popular Design Patterns	104
<b>Table 5-2</b>	Design Formality and Level of Detail Needed	116
<b>Table 6-1</b>	Variations on Inherited Routines	145
<b>Table 8-1</b>	Popular-Language Support for Exceptions	198
<b>Table 11-1</b>	Examples of Good and Bad Variable Names	261
<b>Table 11-2</b>	Variable Names That Are Too Long, Too Short, or Just Right	262
<b>Table 11-3</b>	Sample Naming Conventions for C++ and Java	277
<b>Table 11-4</b>	Sample Naming Conventions for C	278
<b>Table 11-5</b>	Sample Naming Conventions for Visual Basic	278
<b>Table 11-6</b>	Sample of UDTs for a Word Processor	280
<b>Table 11-7</b>	Semantic Prefixes	280
<b>Table 12-1</b>	Ranges for Different Types of Integers	294
<b>Table 13-1</b>	Accessing Global Data Directly and Through Access Routines	341
<b>Table 13-2</b>	Parallel and Nonparallel Uses of Complex Data	342
<b>Table 16-1</b>	The Kinds of Loops	368
<b>Table 19-1</b>	Transformations of Logical Expressions Under DeMorgan's Theorems	436
<b>Table 19-2</b>	Techniques for Counting the Decision Points in a Routine	458
<b>Table 20-1</b>	Team Ranking on Each Objective	469
<b>Table 20-2</b>	Defect-Detection Rates	470
<b>Table 20-3</b>	Extreme Programming's Estimated Defect-Detection Rate	472
<b>Table 21-1</b>	Comparison of Collaborative Construction Techniques	495
<b>Table 23-1</b>	Examples of Psychological Distance Between Variable Names	556
<b>Table 25-1</b>	Relative Execution Time of Programming Languages	600
<b>Table 25-2</b>	Costs of Common Operations	601

<b>Table 27-1</b>	Project Size and Typical Error Density	652
<b>Table 27-2</b>	Project Size and Productivity	653
<b>Table 28-1</b>	Factors That Influence Software-Project Effort	674
<b>Table 28-2</b>	Useful Software-Development Measurements	678
<b>Table 28-3</b>	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

- Figure 29-4** In phased integration, you integrate so many components at once that it's hard to know where the error is. It might be in any of the components or in any of their connections. In incremental integration, the error is usually either in the new component or in the connection between the new component and the system. 693
- Figure 29-5** In top-down integration, you add classes at the top first, at the bottom last. 695
- Figure 29-6** As an alternative to proceeding strictly top to bottom, you can integrate from the top down in vertical slices. 696
- Figure 29-7** In bottom-up integration, you integrate classes at the bottom first, at the top last. 697
- Figure 29-8** As an alternative to proceeding purely bottom to top, you can integrate from the bottom up in sections. This blurs the line between bottom-up integration and feature-oriented integration, which is described later in this chapter. 698
- Figure 29-9** In sandwich integration, you integrate top-level and widely used bottom-level classes first and you save middle-level classes for last. 698
- Figure 29-10** In risk-oriented integration, you integrate classes that you expect to be most troublesome first; you implement easier classes later. 699
- Figure 29-11** In feature-oriented integration, you integrate classes in groups that make up identifiable features—usually, but not always, multiple classes at a time. 700
- Figure 29-12** In T-shaped integration, you build and integrate a deep slice of the system to verify architectural assumptions and then you build and integrate the breadth of the system to provide a framework for developing the remaining functionality. 701
- Figure 34-1** Programs can be divided into levels of abstraction. A good design will allow you to spend much of your time focusing on only the upper layers and ignoring the lower layers. 846

# Part I

# Laying the Foundation

Copyrighted image



## Chapter 1

# Welcome to Software Construction

cc2e.com/0178

## Contents

- 1.1 What Is Software Construction?: page 3
- 1.2 Why Is Software Construction Important?: page 6
- 1.3 How to Read This Book: page 8

## Related Topics

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

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

## 1.1 What Is Software Construction?

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

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

- Integration testing
- Integration
- System testing
- Corrective maintenance

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

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

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



Copyrighted image

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

**KEY POINT**

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

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

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



Copyrighted image

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

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

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

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

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

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

## 1.2 Why Is Software Construction Important?

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

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



Here's why:

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

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

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

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

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

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



**KEY POINT**

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

## 1.3 How to Read This Book

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

### Key Points

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

## Chapter 2

# Metaphors for a Richer Understanding of Software Development

cc2e.com/0278

### Contents

- 2.1 The Importance of Metaphors: page 9
- 2.2 How to Use Software Metaphors: page 11
- 2.3 Common Software Metaphors: page 13

### Related Topic

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

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

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

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

## 2.1 The Importance of Metaphors

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

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

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

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

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

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

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

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

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

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

—*Fernando J. Corbató*

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

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

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

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

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

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

## 2.2 How to Use Software Metaphors



**KEY POINT**

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

An algorithm is a set of well-defined instructions for carrying out a particular task. An algorithm is predictable, deterministic, and not subject to chance. An algorithm tells

you how to go from point A to point B with no detours, no side trips to points D, E, and F, and no stopping to smell the roses or have a cup of joe.

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

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

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

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

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

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

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

## 2.3 Common Software Metaphors

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

### Software Penmanship: Writing Code

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

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



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

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

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

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

Copyrighted image

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

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

## Software Farming: Growing a System

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



### KEY POINT

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

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

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



rotating in a crop of C++ instead of barley, of letting the land rest for a year to increase the supply of nitrogen in the hard disk.

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



Copyrighted image

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

## Software Oyster Farming: System Accretion

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

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

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

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

After you've formed the skeleton, little by little you lay on the muscle and skin. You change each of the dummy classes to real classes. Instead of having your program

pretend to accept input, you drop in code that accepts real input. Instead of having your program pretend to produce output, you drop in code that produces real output. You add a little bit of code at a time until you have a fully working system.

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

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

## Software Construction: Building Software



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

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

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

Copyrighted image

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

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

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

Copyrighted image

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

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

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

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

Careful planning doesn't necessarily mean exhaustive planning or over-planning. You can plan out the structural supports and decide later whether to put in hardwood floors or carpeting, what color to paint the walls, what roofing material to use, and so

on. A well-planned project improves your ability to change your mind later about details. The more experience you have with the kind of software you're building, the more details you can take for granted. You just want to be sure that you plan enough so that lack of planning doesn't create major problems later.

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

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

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

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

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

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

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

## Applying Software Techniques: The Intellectual Toolbox



KEY POINT

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

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

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

## Combining Metaphors



KEY POINT

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

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

## Additional Resources

[cc2e.com/0285](http://cc2e.com/0285)

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

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

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

## Chapter 3

# Measure Twice, Cut Once: Upstream Prerequisites

cc2e.com/0309

### Contents

- 3.1 Importance of Prerequisites: page 24
- 3.2 Determine the Kind of Software You're Working On: page 31
- 3.3 Problem-Definition Prerequisite: page 36
- 3.4 Requirements Prerequisite: page 38
- 3.5 Architecture Prerequisite: page 43
- 3.6 Amount of Time to Spend on Upstream Prerequisites: page 55

### Related Topics

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

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

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

The carpenter's saying, "Measure twice, cut once" is highly relevant to the construction part of software development, which can account for as much as 65 percent of the total project costs. The worst software projects end up doing construction two or

three times or more. Doing the most expensive part of the project twice is as bad an idea in software as it is in any other line of work.

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

## 3.1 Importance of Prerequisites

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

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

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



**KEY POINT**

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

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

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



## Do Prerequisites Apply to Modern Software Projects?

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

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



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

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

## Causes of Incomplete Preparation

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

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

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

cc2e.com/0316

Some programmers do know how to perform upstream activities, but they don't prepare because they can't resist the urge to begin coding as soon as possible. If you feed your

horse at this trough, I have two suggestions. Suggestion 1: Read the argument in the next section. It may tell you a few things you haven't thought of. Suggestion 2: Pay attention to the problems you experience. It takes only a few large programs to learn that you can avoid a lot of stress by planning ahead. Let your own experience be your guide.

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

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

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

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

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

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

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

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

## Utterly Compelling and Foolproof Argument for Doing Prerequisites Before Construction

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



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

### Appeal to Logic

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

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

### Appeal to Analogy

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



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



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

## Boss-Readiness Test

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

Which of these statements are self-fulfilling prophecies?

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

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

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

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

## 3.2 Determine the Kind of Software You're Working On

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

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

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

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

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

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

On real projects, you'll find infinite variations on the three themes presented in this table; however, the generalities in the table are illuminating. Business systems projects tend to benefit from highly iterative approaches, in which planning, requirements,

and architecture are interleaved with construction, system testing, and quality-assurance activities. Life-critical systems tend to require more sequential approaches—requirements stability is part of what's needed to ensure ultrahigh levels of reliability.

## Iterative Approaches' Effect on Prerequisites

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

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

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

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

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

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

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

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



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

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

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



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

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

Copyrighted image

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

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

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

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

Copyrighted image

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



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

## 3.4 Requirements Prerequisite

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

### Why Have Official Requirements?

An explicit set of requirements is important for several reasons.

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

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



Paying attention to requirements helps to minimize changes to a system after development begins. If you find a coding error during coding, you change a few lines of code and work goes on. If you find a requirements error during coding, you have to alter the design to meet the changed requirement. You might have to throw away part of the old design, and because it has to accommodate code that's already written, the new design will take longer than it would have in the first place. You also have to discard

code and test cases affected by the requirement change and write new code and test cases. Even code that's otherwise unaffected must be retested so that you can be sure the changes in other areas haven't introduced any new errors.



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

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

Copyrighted image



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

## The Myth of Stable Requirements

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

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

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



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

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

## Handling Requirements Changes During Construction



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

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

**Make sure everyone knows the cost of requirements changes** Clients get excited when they think of a new feature. In their excitement, their blood thins and runs to their medulla oblongata and they become giddy, forgetting all the meetings you had to discuss requirements, the signing ceremony, and the completed requirements document. The easiest way to handle such feature-intoxicated people is to say, "Gee, that

sounds like a great idea. Since it's not in the requirements document, I'll work up a revised schedule and cost estimate so that you can decide whether you want to do it now or later." The words "schedule" and "cost" are more sobering than coffee and a cold shower, and many "must haves" will quickly turn into "nice to haves."

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

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

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

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

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

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

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

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

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

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

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



**KEY POINT**

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

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

Copyrighted image

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



**HARD DATA**

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

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

## Major Classes

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

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

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

## Data Design

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

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

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

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

## Business Rules

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

## Reuse Decisions

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

## Change Strategy

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

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

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

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

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

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

## General Architectural Quality

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

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

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



# Part VII

# Software Craftsmanship

Copyrighted image

# Index

## Symbols and Numbers

- \* (pointer declaration symbol), 332, 334–335, 763
- & (pointer reference symbol), 332
- > (pointer symbol), 328
- 80/20 rule, 592

## A

- abbreviation of names, 283–285
- abstract data types. *See* ADTs
- Abstract Factory pattern, 104
- abstraction
  - access routines for, 340–342
  - ADTs for. *See* ADTs
  - air lock analogy, 136
  - checklist, 157
  - classes for, 152, 157
  - cohesion with, 138
  - complexity, for handling, 839
  - consistent level for class
    - interfaces, 135–136
  - defined, 89
  - erosion under modification
    - problem, 138
  - evaluating, 135
  - exactness goal, 136–137
  - forming consistently, 89–90
  - good example for class interfaces, 133–134
  - guidelines for creating class
    - interfaces, 135–138
  - high-level problem domain terms, 847
  - implementation structures,
    - low-level, 846
  - inconsistent, 135–136, 138
  - interfaces, goals for, 133–138
  - levels of, 845–847
  - opposites, pairs of, 137
  - OS level, 846
  - patterns for, 103
  - placing items in inheritance trees, 146
  - poor example for class interfaces, 134–135
  - problem domain terms, low-level, 846
  - programming-language level, 846
  - routines for, 164

- access routines
  - abstraction benefit, 340
  - abstraction, level of, 341–342
  - advantages of, 339–340
  - barricaded variables benefit, 339
  - centralized control from, 339
  - creating, 340
  - g\_ prefix guideline, 340
  - information hiding benefit, 340
  - lack of support for, overcoming, 340–342
  - locking, 341
  - parallelism from, 342
  - requiring, 340
- accidental problems, 77–78
- accreting a system metaphor, 15–16
- accuracy, 464
- Ada
  - description of, 63
  - parameter order, 174–175
- adaptability, 464
- Adapter pattern, 104
- addition, dangers of, 295
- ADTs (abstract data types)
  - abstraction with, 130
  - access routines, 339–342
  - benefits of, 126–129
  - changes not propagating benefit, 128
  - classes based on, 133
  - cooling system example, 129–130
  - data, meaning of, 126
  - defined, 126
  - documentation benefit, 128
  - explicit instancing, 132
  - files as, 130
  - guidelines, 130–131
  - hiding information with, 127
  - instancing, 132
  - implicit instancing, 132
  - interfaces, making more
    - informative, 128
  - low-level data types as, 130
  - media independence with, 131
  - multiple instances, handling, 131–133
  - need for, example of, 126–127
  - non-object-oriented languages
    - with, 131–133
  - objects as, 130

- operations examples, table of, 129–130
- passing of data, minimization of, 128
- performance improvements with, 128
- purpose of, 126
- real-world entities, working with, 128–129
- representation question, 130
- simple items as, 131
- verification of code benefit, 128
- agile development, 58, 658
- algebraic identities, 630
- algorithms
  - commenting, 809
  - heuristics compared to, [12](#)
  - metaphors serving as, 11–12
  - resources on, 607
  - routines, planning for, 223
- aliasing, 311–316
- analysis skills development, 823
- approaches to development
  - agile development, 58, 658
  - bottom-up approaches, 112–113, 697–698
  - Extreme Programming, 58, 471–472, 482, 708, 856
  - importance of, 839–841
  - iterative approach. *See* iteration in development
  - premature optimization problem, 840
  - quality control, 840. *See also* quality of software
  - resources for, 58–59
  - sequential approach, 35–36
  - team processes, 839–840
  - top-down approaches, 111–113, 694–696
- architecture
  - building block definition, 45
  - business rules, [46](#)
  - buying vs. building components, 51
  - changes, [44](#), [52](#)
  - checklist for, [54–55](#)
  - class design, [46](#)
  - commitment delay strategy, [52](#)
  - conceptual integrity of, [52](#)

architecture, *continued*  
 data design, [46](#)  
 defined, 43  
 error handling, 49–50  
 fault tolerance, 50  
 GUIs, 47  
 importance of, [44](#)  
 input/output, 49  
 internationalization planning, 48  
 interoperability, 48  
 key point for, 60  
 localization planning, 48  
 machine independence, 53  
 overengineering, 51  
 percent of total activity, by size of  
 project, 654–655  
 performance goals, 48  
 performance-oriented, 590  
 prerequisite nature of, [44](#)  
 program organization, 45–46  
 quality, 52–53, 55  
 resource management, 47  
 resources on developing, 57  
 reuse decisions, [52](#)  
 risky areas, identifying, 53  
 scalability, 48  
 security design, 47  
 technical feasibility, 51  
 time allowed for, 56  
 user interface design, 47  
 validation design, 50

arithmetic expressions  
 misleading precedence example,  
 733  
 magnitudes, greatly different, 295  
 multiplication, changing to  
 addition, 623–624  
 rounding errors, 297

arrays  
 C language macro for, 311  
 checklist, 317  
 containers as an alternative, 310  
 costs of operations, 602  
 cross-talk, 311  
 defined, 310  
 dimensions, minimizing,  
 625–626  
 end points, checking, 310  
 foreach loops with, 372  
 indexes of, 310–311  
 layout of references, 754  
 loops with, 387–388  
 multidimensional, 310  
 naming conventions for, 280–281

performance tuning, 593–594,  
 603–604  
 refactoring, 572  
 references, minimizing, 626–627  
 semantic prefixes for, 280–281  
 sentinel tests for loops, 621–623  
 sequential access guideline, 310

assembly language  
 description of, 63  
 listing tools, 720  
 recoding to, 640–642

assertions  
 aborting program recommended,  
 206  
 arguments for, 189  
 assumptions to check, list of, 190  
 barricades, relation to, 205  
 benefits of, 189  
 building your own mechanism  
 for, 191  
 C++ example, 191  
 dangerous use of example, 192  
 defined, 189  
 dependencies, checking for, 350  
 error handling with, 191, 193–194  
 executable code in, 191–192  
 guidelines for, 191–193  
 Java example of, 190  
 postcondition verification,  
 192–193  
 precondition verification,  
 192–193  
 removing from code, 190  
 resources for, 212  
 Visual Basic examples, 192–194

assignment statements, 249, 758  
 author role in inspections, 486  
 auto\_ptrs, 333  
 automated testing, 528–529

## B

backup plans, 669, 670  
 bad data, testing for, 514–515  
 barricades  
 assertions, relation to, 205  
 class-level, 204  
 input data conversions, 204  
 interfaces as boundaries, 203  
 operating room analogy, 204  
 purpose of, 203

base classes  
 abstract overridable routines, 145  
 abstraction aspect of, 89  
 coupling, too tight, 143

Liskov Substitution Principle,  
 144–145  
 overridable vs. non-overridable  
 routines, 145–146  
 protected data, 143  
 routines overridden to do  
 nothing, 146–147  
 single classes from, 146

Basic, 65. *See also* Visual Basic  
 basis testing, structured, 503,  
 505–509

BCD (binary coded decimal) type,  
 297

BDUF (big design up front), 119  
 beauty, 80  
 begin-end pairs, 742–743  
 bibliographies, software, 858  
 big-bang integration, 691  
 big design up front (BDUF), 119  
 binary searches, 428

binding  
 in code, 252  
 compile time, 252–253  
 heuristic design with, 107  
 just in time, 253  
 key point, 258  
 load time, 253  
 run time, 253  
 variables, timing of, 252–254

black-box testing, 500  
 blank lines for formatting, 747–748,  
 765–766

blocks  
 braces writing rule, 443  
 comments on, 795–796  
 conditionals, clarifying, 443  
 defined, 443  
 emulated pure layout style,  
 740–743  
 pure, layout style, 738–740  
 single statements, 748–749

Book Paradigm, 812–813

boolean expressions  
 0, comparisons to, 441–442  
 0s and 1s as values, 432  
 breaking into partial tests, 433  
 C languages syntax, 442–443  
 characters, comparisons to zero,  
 441  
 checklist for, 459  
 constants in comparisons,  
 442–443  
 decision tables, moving to, 435  
 DeMorgan's Theorems, applying,  
 436–437

- evaluation guidelines, 438–440
  - functions, moving to, 434–435
  - identifiers for, 431–433
  - if statements, negatives in, 435–436
  - implicit comparisons, 433
  - Java syntax, 439, 443
  - layout guidelines, 749–750
  - logical identities, 630
  - negatives in, 435–437
  - numeric, structuring, 440–441
  - parentheses for clarifying, 437–438
  - pointers, comparisons with, 441
  - positive form recommended, 435–437
  - refactoring, 572
  - short circuit evaluation, 438–440
  - simplifying, 433–435
  - variables in. *See* boolean variables
  - zero, comparisons to, 441–442
  - boolean functions
    - creating from expressions, 434–435
    - if statements, used in, 359
  - boolean tests
    - breaking into partial tests, 433
    - hiding with routines, 165
    - simplifying, 301–302
    - zero, comparisons to, 441–442
  - boolean variables
    - 0s and 1s as values, 432
    - C, creating data type, 302–303
    - checklist, 317
    - documentation with, 301
    - enumerated types as alternative, 304
    - expressions with. *See* boolean expressions
    - identifiers for, 431–433
    - naming, 268–269
    - simplifying tests with, 301–302
    - zeros and ones as values, 432
  - boss readiness test on prerequisites, 30–31
  - bottom-up approach to design, 112–113
  - bottom-up integration, 697–698
  - boundary analysis, 513–514
  - braces
    - block layout with, 740–743
    - styles compared, 734
  - break statements
    - C++ loops, 371–372
    - caution about, 381
    - guidelines, 379–380
  - labeled, 381
  - multiple in one loop, 380
  - nested-if simplification with, 446–447
  - while loops with, 379
  - bridge failure, Tacoma Narrows, 74
  - Bridge pattern, 104
  - brute-force debugging, 548–549
  - buffer overruns, 196
  - bugs. *See* debugging; defects in code; errors
  - build tools, 716–717. *See also* compilers
  - building metaphor, 16–19
  - building vs. buying components, [18](#)
  - builds, daily. *See* daily build and smoke tests
  - business rules
    - architecture prerequisites, [46](#)
    - change, identifying areas of, 98
    - good practices table for, 31–32
    - subsystem design, 85
  - buying components, [18](#), 51
- C**
- C language
    - ADTs with, 131
    - boolean expression syntax, 442–443
    - description of, 64
    - naming conventions for, 275, 278
    - pointers, 334–335
    - string data types, 299–301, 317
    - string index errors, 299–300
  - C#, 64
  - C++
    - assertion example, 191
    - boolean expression syntax, 442–443
    - debugging stubs with, 208–209
    - description of, 64
    - DoNothing() macros, 444–445
    - exceptions in, 198–199
    - inline routines, 184–185
    - interface considerations, 139–141
    - layout recommended, 745
    - macro routines, 182–184
    - naming conventions for, 275–277
    - null statements, 444–445
    - parameters, by reference vs. by value, 333
    - pointers, 325, 328–334, 763
    - preprocessors, excluding debug code, 207–208
    - resources for, 159
    - side effects, 759–761
    - source files, layout in, 773
  - caching, code tuning with, 628–629
  - Capability Maturity Model (CMM), 491
  - capturing design work, 117–118
  - Cardinal Rule of Software Evolution, 565
  - CASE (computer-aided software engineering) tools, 710
  - case statements
    - alpha ordering, 361
    - checklist, 365
    - debugging, 206
    - default clauses, 363
    - drop-throughs, 363–365
    - end of case statements, 363–365
    - endline layout, 751–752
    - error detection in, 363
    - frequency of execution ordering, 361, 612–613
    - if statements, comparing performance with, 614
    - key points, 366
    - language support for, 361
    - nested ifs, converting from, 448–449, 451
    - normal case first rule, 361
    - numeric ordering, 361
    - ordering cases, 361
    - parallel modifications to, 566
    - phony variables, 361–362
    - polymorphism preferable to, 147–148
    - redesigning, 453
    - refactoring, 566, 573
    - simple action guideline, 361
    - table-driven methods using, 421–422
  - change control. *See* configuration management
  - character arrays, 299–300. *See also* string data types
  - character data types
    - arrays vs. string pointers, 299
  - C language, 299–301
  - character sets, 298
  - checklist, 316–317
  - conversion strategies, 299
  - magic (literal) characters, 297–298
  - Unicode, 298, 299
  - character, personal
    - analysis skills, 823
    - communication skills, 828

- character, personal, *continued*
  - compiler messages, treatment of, 826–827
  - computer-science graduates, 829
  - cooperation skills, 828
  - creativity, 829, 857
  - curiosity, 822–825
  - development process awareness, 822
  - discipline, 829
  - estimations, 827–828
  - experience, 831–832
  - experimentation, 822–823
  - gonzo programming, 832
  - habits, 833–834
  - humility, 821, 826, 834
  - importance of, 819–820
  - intellectual honesty, 826–828
  - intelligence, 821
  - judgment, 848
  - key points, 835
  - laziness, 830
  - mistakes, admitting to, 826
  - persistence, 831
  - practices compensating for weakness, 821
  - problem solving, 823
  - professional development, 824–825
  - reading, 824
  - religion in programming, harmful effects of, 851–853
  - resources on, 834–835
  - status reporting, 827
  - successful projects, learning from, 823–824
- checklists
  - abstraction, 157
  - architecture, 54–55
  - arrays, 317
  - backups, 670
  - boolean expressions, 459
  - case statements, 365
  - character data types, 316–317
  - classes, 157–158, 233–234, 578–579, 774, 780
  - coding practices, 69
  - code tuning, 607–608, 642–643
  - comments, 774, 816–817
  - conditional statements, 365
  - configuration management, 669–670
  - constants, 317
  - construction practices, 69–70
  - control structures, 459, 773, 780
  - daily build and smoke tests, 707
  - data organization, 780
  - data types, 316–318
  - debugging, 559–561
  - defects, 489, 559–560
  - defensive programming, 211–212
  - design, 122–123, 781
  - documentation, 780–781, 816–817
  - encapsulation, 158
  - enumerated types, 317
  - fixing defects, 560
  - formal inspections, 489, 491–492
  - formatting, 773–774
  - goto statements, 410
  - if statements, 365
  - inheritance, 158
  - initialization, 257
  - integration, 707
  - interfaces, 579
  - layout, 773–774
  - list of, xxix–xxx
  - loops, 388–389
  - names, 288–289, 780
  - pair programming, 484
  - parameters, 185
  - performance tuning, 607–608
  - pointers, 344
  - prerequisites, 59
  - pseudocoding, 233–234
  - programming tools, 724–725
  - quality assurance, 42–43, 70, 476
  - refactoring, 570, 577–579, 584
  - requirements, 40, 42–43
  - routines, 185, 774, 780
  - speed, tuning for, 642–643
  - statements, 774
  - straight-line code, 353
  - strings, 316–317
  - structures, 343
  - table-driven methods, 429
  - testing, 503, 532
  - tools, 70
  - type creation, 318
  - variables, 257–258, 288–289, 343–344
- circular dependencies, 95
- classes
  - abstract data types. *See* ADTs
  - abstract objects, modeling, 152
  - abstraction checklist, 157
  - alternates to PPP, 232–233
  - architecture prerequisites, 46
  - assumptions about users, 141
  - base. *See* base classes
  - bidirectional associations, 577
  - calls to, refactoring, 575
  - case statements vs. inheritance, 147–148
  - centralizing control with, 153
  - changes, limiting effects of, 153
  - checklists, 157–158, 774, 780
  - coding routines from pseudocode, 225–229
  - cohesion as refactoring indicator, 566
  - complexity issues, 152–153
  - constant values returned, 574
  - constructors, 151–152
  - containment, 143–144
  - coupling considerations, 100–102, 142–143
  - data-free, 155
  - deep inheritance trees, 147
  - defined, 125
  - delegation vs. inheritance, refactoring, 576
  - descendants, refactoring indicator for, 567
  - designing, 86, 216, 220–225, 233
  - disallowing functions and operators, 150
  - documenting, 780, 810
  - encapsulation, 139–143, 158
  - extension, refactoring with, 576
  - factoring, benefit of, 154
  - files containing, 771–772
  - foreign routines, refactoring with, 576
  - formalizing contracts for interfaces, 106
  - formatting, 768–771
  - friend, encapsulation violation concern, 141
  - functions in. *See* functions; routines
  - global data, hiding, 153
  - god classes, 155
  - hacking approach to, 233
  - hiding implementation details, 153
  - implementation checklist, 158
  - indirect calls to other classes, 150
  - information hiding, 92–93
  - inheritance, 144–149, 158
  - initializing members, 243
  - integration, 691, 694, 697
  - irrelevant classes, 155
  - is a relationships, 144
  - key points for, 160, 234

- language-specific issues, 156
- layout of, 768–771
- limiting collaboration, 150
- Liskov Substitution Principle, 144–145
- member variables, naming, 273, 279
- methods of. *See* routines
- minimizing accessibility rule, 139
- mixins, 149
- modeling real-world objects, 152
- multiple per file, layout of, 769–770
- naming, 277, 278
- number of members, 143
- number of routines, 150
- object names, differentiating from, 272–273
- objects, contrasted with, 86
- overformatting, 770
- overriding routines, 145–146, 156
- packages, 155–157
- parallel modifications refactoring indicator, 566
- planning for program families, 154
- private vs. protected data, 148
- private, declaring members as, 150
- procedures in. *See* routines
- protected data, 148
- pseudocode for designing, 232–234
- public members, 139, 141, 576
- read-time convenience rule, 141
- reasons for creating, 152–156
- refactoring, 155, 574–576, 578–579, 582
- resources, 159
- reusability benefit of, 154
- review and test step, 217
- routine construction step, 217
- routines in. *See* routines
- routines, unused, 146–147, 576
- semantic violations of
  - encapsulation, 141–142
- Set() routines, unnecessary, 576
- similar sub and superclasses, 576
- single-instance, 146
- singleton property, enforcing, 151
- steps in creating, 216–217
- streamlining parameter passing, 153
- subclasses, 165, 575
- superclasses for common code, 575
- test-first development, 233
- testing with stub objects, 523
- unidirectional associations, 577
- visibility of, 93
- warning signs for, 848, 849
- class-hierarchy generators, 713
- cleanup steps, PPP, 232
- cleanroom development, 521
- CMM (Capability Maturity Model), 491
- Cobol, 64
- code coverage testing, 506
- code libraries, 222, 717
- code quality analysis tools, 713–714
- code reading method, 494
- code tuning
  - 80/20 rule, 592
  - advantages from, 591
  - algebraic identities, 630
  - appeal of, 591–592
  - arrays, 593–594, 603–604, 625–627
  - assembler, listing tools, 720
  - assembler, recoding to, 640–642
  - bottleneck identification, 594
  - caching data, 628–629
  - checklists, 607–608, 642–643
  - comparing logic structures, 614
  - competing objectives dilemma, 595
  - compiler considerations, 590, 596–597
  - converting data types, 635
  - correctness, importance of, 595–596
  - data transformations, 624–629
  - data type choices, 635
  - database indexing, 601
  - defects in code, 601
  - defined, 591
  - DES example, 605–606
  - design view, 589–590
  - disadvantages of, 591
  - disassemblers, 720
  - execution profiler tools, 720
  - expressions, 630–639
  - feature specific, 595
  - frequency, testing in order of, 612–613
  - frequently used code spots, 592
  - hardware considerations, 591
  - improvements possible, 605
  - indexing data, 627–628
  - inefficiency, sources of, 598–601
  - initializing at compile time, 632–633
  - inline routines, 639–640
  - input/output, 598–599
  - integers preferred to floating, 625
  - interpreted vs. compiled languages, 592, 600–601
  - iteration of, 608, 850
  - jamming loops, 617–618
  - key points, 608, 645
  - language specificity, 644
  - lazy evaluation, 615–616
  - lines of code, minimizing number of, 593–594
  - logic manipulation guidelines, 610–616
  - lookup tables for, 614–615, 635
  - loops, 616–624
  - low-level language, recoding to, 640–642
  - measurement to locate hot spots, 603–604, 644
  - memory vs. file operations, 598–599
  - minimizing work inside loops, 620–621
  - multiplication, changing to addition, 623–624
  - nested loop order, 623
  - old wives' tales, 593–596
  - operating system considerations, 590
  - operation speeds, presumptions about, 594
  - operations, costs of common, 601–603
  - optimizing as you go, 594–595
  - overview of, 643–644
  - paging operations, 599
  - Pareto Principle, 592
  - precomputing results, 635–638
  - program requirements view of, 589
  - refactoring, compared to, 609
  - resource goals, 590
  - resources on, 606–607, 644–645
  - right shifting, 634
  - routines, 590, 639–640
  - sentinel tests for loops, 621–623
  - short-circuit evaluation, 610
  - speed, importance of, 595–596
  - strength reduction, 623–624, 630–632

construction schedules, estimating,  
*continued*  
 overview, 671  
 planning estimation time, 671  
 reduction of scope, 676  
 reestimating, 672  
 requirements specification, 672  
 resources for, 677  
 teams, expanding, 676

constructors  
 deep vs. shallow copies, 151–152  
 exceptions with, 199  
 guidelines for, 151–152  
 initializing data members, 151  
 refactoring, 577  
 singleton property, enforcing, 151

container classes, 310

containment, 88, 143

continuation lines, 754–758

continue statements, 379, 380, 381

continuous integration, 706

control structures  
 boolean expressions in. *See* boolean expressions  
 case. *See* case statements  
 checklists, 459, 773, 780  
 commenting, 804–805, 817  
 complexity, contributions to, 456–459  
 compound statements, 443  
 conditional flow. *See* conditional statements  
 continuation lines in, 757  
 data types, relationship to, 254–255  
 documentation, 780  
 double indented begin-end pairs, 746–747  
 gotos. *See* goto statements  
 if statements. *See* if statements  
 iteration, 255, 456  
 key points, 460  
 layout styles, 745–752  
 loops. *See* loops  
 multiple returns from routines, 391–393  
 null statements, 444–445  
 recursive. *See* recursion  
 reliability correlated with complexity, 457  
 returns as. *See* return statements  
 selective data with, 254  
 sequential data with, 254  
 structured programming, 454–455

unindented begin-end pairs, 746  
 unusual, overview of, 408

conventions, coding  
 benefits of, 844–845  
 checklist, 69  
 formatting. *See* layout  
 hazards, avoiding with, 844  
 predictability benefit, 844

converting data types, 635

cooperation skills, importance of, 828

correctness, 197, 463

costs. *See also* performance tuning  
 change estimates, 666  
 collaboration benefits, 480–481  
 debugging, time consumed by, 474–475  
 defects contributing to, 519–520  
 detection of defects, 472  
 error-prone routines, 518  
 estimating, 658, 828  
 fixing of defects, 472–473, 519  
 General Principle of Software Quality, 474–475, 522  
 pair programming vs. inspections, 480–481  
 resources on, 658

counted loops. *See* for loops

coupling  
 base classes to derived classes, 143  
 classes, too tightly, 142–143  
 design considerations, 100–102  
 flexibility of, 100–101  
 goals of, 100  
 loose, 80, 100–102  
 object-parameter type, 101  
 semantic type, 102  
 simple-data-parameter type, 101  
 simple-object type, 101  
 size of, 100  
 visibility of, 100

coverage  
 monitoring tools, 526  
 structured basis testing, 505–509

CRC (Class, Responsibility, Collaboration) cards, 118

creativity, importance of, 829, 857

cross-reference tools, 713

curiosity, role in character, 822–825

Currency data types, 297

customization, building metaphor for, [18](#)

**D**

daily build and smoke tests  
 automation of, 704  
 benefits of, 702  
 broken builds, 703, 705  
 build groups, 704  
 checklist, 707  
 defined, 702  
 diagnosis benefit, 702  
 holding area for additions, 704–705  
 importance of, 706  
 morning releases, 705  
 pressure, 706  
 pretest requirement, 704  
 revisions, 704  
 smoke tests, 703  
 unsurfaced work, 702

data  
 architecture prerequisites, [46](#)  
 bad classes, testing for, 514–515  
 change, identifying areas of, 99  
 code tuning. *See* data transformations for code tuning  
 combined states, 509–510  
 defined state, 509–510  
 defined-used paths, testing, 510–512  
 design, [46](#)  
 entered state, 509  
 exited state, 509  
 good classes, testing, 515–516  
 killed state, 509–510  
 legacy, compatibility with, 516  
 nominal case errors, 515  
 test, generators for, 524–525  
 types. *See* data types  
 used state, 509–510

data dictionaries, 715

data flow testing, 509–512

data literacy test, 238–239

data recorder tools, 526

data structures. *See* structures  
 data transformations for code tuning  
 array dimension minimization, 625–626  
 array reference minimization, 626–627  
 caching data, 628–629  
 floating point to integers, 625  
 indexing data, 627–628  
 purpose of, 624

- data types
  - “a” prefix convention, 272
  - abstract data types. *See* ADTs
  - arrays. *See* arrays
  - BCD, 297
  - boolean. *See* boolean variables
  - change, identifying areas of, 99
  - characters. *See* character data types
  - checklist, 316–318
  - complex. *See* structures
  - control structures, relationship to, 254–255
  - creating. *See* type creation
  - Currency, 297
  - definitions, 278
  - enumerated types. *See* enumerated types
  - floating-point. *See* floating-point data types
  - integers. *See* integer data types
  - iterative data, 255
  - key points for, 318
  - naming, 273, 277, 278
  - numeric. *See* numeric data types
  - overloaded primitives, 567
  - pointers. *See* pointers
  - refactoring to classes, 567, 572
  - resources on, 239
  - selective data, 254
  - sequential data, 254
  - strings. *See* string data types
  - structures. *See* structures
  - t\_ prefix convention, 272
  - user-defined. *See* type creation
  - variables of, differentiating from, 272–273
- databases
  - performance issues, 601
  - SQL, 65
  - subsystem design, 85
- data-level refactoring, 571–572, 577
- days-in-month, determining, 413–414
- deallocation
  - goto statements for, 399
  - pointers, of, 326, 330, 332
- Debug.Assert statements, 191–193
- debugging
  - aids to. *See* debugging aids
  - binary searches of code, 546
  - blindness, sources of, 554–555
  - breakpoints, 558
  - breaks, taking, 548
  - brute-force, 548–549
  - changes, recent, 547
  - checklist, 559–561
  - comments, misplaced, 550
  - common defects lists, 547
  - compilers as tools for, 549, 557
  - confessional debugging, 547–548
  - costs of, 29–30, 474–475
  - debugger tools, 526–527, 545, 556–559, 719. *See also* debugging aids
  - defects as opportunities, 537–538
  - defensive. *See* debugging aids defined, 535
  - Diff tool, 556
  - execution profilers for, 557–558
  - expanding suspicious regions, 547
  - experience of programmers, effects of, 537
  - finding defects, 540, 559–560
  - fixing defects, 550–554
  - guessing, 539
  - history of, 535–536
  - hypothesis testing, 543–544, 546
  - incremental approach, 547
  - ineffective approach to, 539–540
  - key points, 562
  - line numbers from compilers, 549
  - lint tool, 557
  - listing possibilities, 546
  - locating error sources, 543–544
  - logic checking tools, 557
  - multiple compiler messages, 550
  - narrowing code searches, 546
  - obvious fixes, 539
  - performance variations, 536–537
  - project-wide compilers settings, 557
  - psychological considerations, 554–556
  - quality of software, role in, 536
  - quotation marks, misplaced, 550
  - readability improvements, 538
  - recommended approach, 541
  - reexamining defect-prone code, 547
  - resources for, 561
  - Satan’s helpers, 539–540
  - scaffolding for, 558
  - scientific method of, 540–544
  - self-knowledge from, 538
  - source-code comparators, 556
  - stabilizing errors, 542–543
  - superstitious approaches, 539–540
  - symbolic debuggers, 526–527
  - syntax checking, 549–550, 557, 560
  - system debuggers, 558
  - test case creation, 544
  - testing, compared to, 500
  - time for, setting maximums, 549
  - tools for, 526–527, 545, 556–559, 719. *See also* debugging aids
  - understanding the problems, 539
  - unit tests, 545
  - varying test cases, 545
  - warnings, treating as errors, 557
- debugging aids
  - C++ preprocessors, 207–208
  - case statements, 206
  - early introduction recommended, 206
  - offensive programming, 206
  - planning removal of, 206–209
  - pointers, checking, 208–209
  - preprocessors, 207–208
  - production constraints in
    - development versions, 205
    - purpose of, 205
  - stubs, 208–209
  - version control tools, 207
- decision tables. *See* table-driven methods
- declarations
  - commenting, 794, 802–803, 816
  - const recommended, 243
  - declare and define near first use rule, 242–243
  - define near first use rule, 242–243
  - final recommended, 243
  - formatting, 761–763
  - implicit declarations, 239–240
  - multiple on one line, 761–762
  - naming. *See* naming conventions
  - numerical data, commenting, 802
  - order of, 762
  - placement of, 762
  - pointers, 325–326, 763
  - using all declared, 257
- Decorator pattern, 104
- defects in code
  - classes prone to error, 517–518
  - classifications of, 518–520
  - clerical errors (typos), 519
  - Code Complete example, 490–491
  - construction, proportion resulting from, 520–521



- defects in code, *continued*
  - cost of detection, 472
  - cost of fixing, 472–473
  - databases of, 527
  - detection by various techniques, table of, 470
  - distribution of, 517–518
  - ease of fixing defects, 519
  - error checklists, 489
  - expected rate of, 521–522
  - finding, checklist, 559–560
  - fixing. *See* debugging; fixing defects
  - formal inspections for detecting. *See* formal inspections
  - intermittent, 542–543
  - misunderstood designs as sources for, 519
  - opportunities presented by, 537–538
  - outside of construction domain, 519
  - percentage of, measurement, 469–472
  - performance issues, 601
  - programmers at fault for, 519
  - readability improvements, 538
  - refactoring after fixing, 582
  - scope of, 519
  - self-knowledge from, 538
  - size of projects, effects on, 651–653
  - sources of, table, 518
  - stabilizing, 542–543
- defensive programming
  - assertions, 189–194
  - assumptions to check, list of, 190
  - barricades, 203–205
  - checklist, 211–212
  - debugging aids, 205–209
  - defined, 187
  - error handling for, 194–197
  - exceptions, 198–203, 211
  - friendly messages guideline, 210
  - graceful crashing guideline, 210
  - guidelines for production code, 209–210
  - hard crash errors guideline, 209
  - important errors guideline, 209
  - key points for, 213
  - logging guideline, 210
  - problems caused by, 210
  - quality improvement techniques, other, 188
  - robustness vs. correctness, 197
  - security issues, 212
  - trivial errors guideline, 209
  - validating input, 188
- defined data state, 509–510
- defining variables. *See* declarations
- Delphi, recoding to assembler, 640–642
- DeMorgan's Theorems, applying, 436–437
- dependencies, code-ordering
  - checker tools, 716
  - circular, 95
  - clarifying, 348–350
  - concept of, 347
  - documentation, 350
  - error checking, 350
  - hidden, 348
  - initialization order, 348
  - naming routines, 348–349
  - non-obvious, 348
  - organization of code, 348
  - parameters, effective, 349
- design
  - abstractions, forming consistent, 89–90
  - accidental problems, 77–78
  - BDUF, 119
  - beauty, 80
  - bottom-up approach to design, 112–113
  - business logic subsystem, 85
  - capturing work, 117–118
  - central points of control, 107
  - change, identifying areas of, 97–99
  - changes, management of, 666–667
  - characteristics of high quality, 80–81
  - checklists, 122–123, 781
  - classes, division into, 86
  - collaboration, 115
  - communications among subsystems, 83–84
  - completion of, determining, 115–117
  - complexity management, 77–80
  - construction activity, as, 73–74
  - contract, by, 233
  - coupling considerations, 100–102
  - database access subsystem, 85
  - defined, 74
  - diagrams, drawing, 107
  - discussion, summarizing, 117
- divide and conquer technique, 111
- documentation, as, 781
- documentation overkill, 117
- emergent nature of, 76
- encapsulation, 90–91
- enough, determining, 118–119
- essential problems, 77–78
- extensibility goal, 80
- formality of, determining, 115–117
- formalizing class contracts, 106
- goals checklist, 122–123
- good practices table for, 31–32
- heuristic. *See* heuristic design
- hierarchies for, 105–106
- high fan-in goal, 80
- IEEE standards, 122
- information hiding, 92–97, 120
- inheritance, 91–92
- iteration practice, 111–117
- key points, 123
- leanness goal, 81
- level of detail needed, 115–117
- levels of, 82–87
- loose coupling goal, 80
- low-to-medium fan-out goal, 81
- maintenance goals, 80
- mental limitations of humans, 79
- metrics, warning signs from, 848
- nondeterministic nature of, 76, 87
- object-oriented, resource for, 119
- objects, real world, finding, 87–89
- packages level, 82–85
- patterns, common. *See* patterns
- performance tuning
  - considerations, 589–590
- portability goal, 81
- practice heuristics. *See* heuristic design
- practices, 110–118, 122
- prioritizing during, 76
- prototyping, 114–115
- resources for, 119–121
- restrictive nature of, 76
- reusability goal, 80
- routines, of, 86–87
- sloppy process nature of, 75–76
- software system level, 82
- standard techniques goal, 81
- standards, IEEE, 122
- stratification goal, 81
- strong cohesion, 105
- subsystem level, 82–85

- system dependencies subsystem, 85
  - testing for implementation, 503
  - tools for, 710
  - top-down approach, 111–113
  - tradeoffs, 76
  - UML diagrams, 118
  - user interface subsystem, 85
  - visual documentation of, 118
  - wicked problem nature of, 74–75
  - Wikis, capturing on, 117
  - destructors, exceptions with, 199
  - detailed-design documents, 778
  - developer testing. *See* testing
  - development processes. *See* approaches to development
  - development standards, IEEE, 813
  - diagrams
    - heuristic design use of, 107
    - UML, 118
  - Diff tools, 556, 712
  - direct access tables
    - advantages of, 420
    - arrays for, 414
    - case statement approach, 421–422
    - days-in-month example, 413–414
    - defined, 413
    - design method for, 420
    - flexible-message-format example, 416–423
    - fudging keys for, 423–424
    - insurance rates example, 415–416
    - keys for, 423–424
    - object approach, 422–423
    - transforming keys, 424
  - disassemblers, 720
  - discipline, importance of, 829
  - discourse rules, 733
  - disposing of objects, 206
  - divide and conquer technique, 111
  - division, 292–293
  - Do loops, 369–370. *See also* loops
  - documentation
    - abbreviation of names, 284–285
    - ADTs for, 128
    - bad code, of, 568
    - Book Paradigm for, 812–813
    - capturing work, 117–118
    - checklists, 780–781, 816–817
    - classes, 780
    - comments. *See* comments
    - control structures, 780
    - CRC cards for, 118
    - dependencies, clarifying, 350
    - design as, 117, 781
    - detailed-design documents, 778
    - external, 777–778
    - Javadoc, 807, 815
    - key points, 817
    - names as, 284–285, 778–779, 780
    - organization of data, 780
    - parameter assumptions, 178
    - pseudocode, deriving from, 220
    - resources on, 815
    - routine parameter assumptions, 178
    - routines, 780
    - SDFs, 778
    - self-documenting code, 778–781
    - size of projects, effects of, 657
    - source code as, 7
    - standards, IEEE, 813–814
    - style differences, managing, 683
    - UDFs, 778
    - visual, of designs, 118
    - why vs. how, 797–798
  - dog-and-pony shows, 495
  - dog tag fields, 326–327
  - DoNothing() macros, 444–445
  - DRY (Don't Repeat Yourself) principle, 565
  - duplication
    - avoiding with routines, 164–165
    - code as refactoring indicator, 565
- E**
- early-wave environments, 67
  - ease of maintenance design goal, 80
  - eclecticism, 851–852
  - editing tools
    - beautifiers, 712
    - class-hierarchy generators, 713
    - cross-reference tools, 713
    - Diff tools, 712
    - grep, 711
    - IDEs, 710–711
    - interface documentation, 713
    - merge tools, 712
    - multiple-file string searches, 711–712
    - templates, 713
  - efficiency, 464
  - eighty/twenty (80/20) rule, 592
  - else clauses
    - boolean function calls with, 359
    - case statements instead of, 360
    - chains, in, 358–360
    - common cases first guideline, 359–360
    - correctness testing, 358
    - default for covering all cases, 360
    - gotos with, 406–407
    - null, 358
  - embedded life-critical systems, 31–32
  - emergent nature of design process, 76
  - emulated pure blocks layout style, 740–743
  - encapsulation
    - assumptions about users, 141
    - checklist, 158
    - classes, role for, 139–143
    - coupling classes too tightly, 142–143
    - downcast objects, 574
    - friend class concern, 141
    - heuristic design with, 90–91
    - minimizing accessibility, 139
    - private details in class interfaces, 139–141
    - public data members, 567
    - public members of classes, 139
    - public routines in interfaces concern, 141
    - semantic violations of, 141–142
    - weak, 567
  - endless loops, 367, 374
  - endline comments, 793–795
  - endline layout, 743–745, 751–752, 767
  - enumerated types
    - benefits of, 303
    - booleans, alternative to, 304
    - C++, 303–304, 306
    - changes benefit, 304
    - checklist, 317
    - comments substituting for, 802–803
    - creating for Java, 307
    - defined, 303
    - emulation by global variables, 338
    - explicit value pitfalls, 306
    - first entry invalid trick, 305–306
    - iterating through, 305
    - Java, creating for, 307
    - languages available in, 303
    - loop limits with, 305
    - naming, 269, 274, 277–279
    - parameters using, 303
    - readability from, 303
    - reliability benefit, 304

- enumerated types, *continued*
    - standard for, 306
    - validation with, 304–305
    - Visual Basic, 303–306
  - equality, floating-point, 295–296
  - equivalence partitioning, 512
  - error codes, 195
  - error detection, doing early, 29–30
  - error guessing, 513
  - error handling. *See also* exceptions
    - architecture prerequisites, 49–50
    - assertions, compared to, 191
    - barricades, 203–205
    - buffer overruns compromising, 196
    - closest legal value, 195
    - defensive programming,
      - techniques for, 194–197
    - error codes, returning, 195
    - error-processing routines, calling, 196
    - high-level design implication, 197
    - local handling, 196
    - logging warning messages, 195
    - messages, 49, 195–196, 210
    - next valid data, returning, 195
    - previous answers, reusing, 195
    - propagation design, 49
    - refactoring, 577
    - returning neutral values, 194
    - robustness, 51, 197
    - routines, designing along with, 222
    - shutting down, 196
    - validation design, 50
  - error messages
    - codes, returning, 195
    - design, 49
    - displaying, 196
    - friendly messages guideline, 210
  - errors. *See also* defects in code; exceptions
    - classifications of, 518–520
    - coding. *See* defects in code
    - dog tag fields, 326–327
    - exceptions. *See* exceptions handling. *See also* error handling
    - goto statements for processing, 401–402
    - sources of, table, 518
  - essential problems, 77–78
  - estimating schedules
    - approaches to, list of, 671
    - change costs, 666
    - control, compared to, 675
  - factors influencing, 674–675
  - level of detail for, 672
  - inaccuracy, character-based, 827–828
  - multiple techniques with
    - comparisons, 672
  - objectives, establishing, 671
  - optimism, 675
  - overview, 671
  - planning for estimation time, 671
  - redoing periodically, 672
  - reduction of scope, 676
  - requirements specification, 672
  - resources for, 677
  - teams, expanding, 676
- event handlers, 170
  - evolution. *See* software evolution
  - Evolutionary Delivery. *See* incremental development metaphor
  - exceptions. *See also* error handling
    - abstraction issues, 199–200
    - alternatives to, 203
    - base classes for, project specific, 203
    - C++, 198–199
    - centralized reporters, 201–202
    - constructors with, 199
    - defensive programming checklist, 211
    - destructors with, 199
    - empty catch blocks rule, 201
    - encapsulation, breaking, 200
    - full information rule, 200
    - Java, 198–201
    - languages, table comparing, 198–199
    - level of abstraction rule, 199–200
    - library code generation of, 201
    - local handling rule, 199
    - non-exceptional conditions, 199
    - purpose of, 198, 199
    - readability of code using, 199
    - refactoring, 577
    - resources for, 212–213
    - standardizing use of, 202–203
    - Visual Basic, 198–199, 202
  - execution profilers, 557–558, 720
  - executable-code tools
    - build tools, 716–717
    - code libraries, 717
    - code-generation wizards, 718
    - compilers. *See* compilers
    - installation tools, 718
    - linkers, 716
    - preprocessors, 718–719
    - setup tools, 718
  - Exit Function, 391. *See also* return statements
  - Exit statements. *See* break statements
  - Exit Sub, 392–393. *See also* return statements
  - exiting loops, 369–372, 377–381
  - experience, personal, 831–832
  - experimental prototyping, 114–115
  - experimentation as learning, 822–823, 852–853
  - exponential expressions, 631–632
  - expressions
    - boolean. *See* boolean expressions
    - constants, data types for, 635
    - initializing at compile time, 632–633
    - layout guidelines, 749–750
    - precomputing results, 635–638
    - right shifting, 634
    - strength reduction, 630–632
    - subexpression elimination, 638–639
    - system calls, performance of, 633–634
  - extensibility design goal, 80
  - external audits, 467
  - external documentation, 777–778
  - Extreme Programming
    - collaboration component of, 482
    - defect detection, 471–472
    - defined, 58
    - resources on, 708, 856
- ## F
- Facade pattern, 104
  - factorials, 397–398
  - factoring, 154. *See also* refactoring
  - factory methods
    - Factory Method pattern, 103–104
    - nested ifs refactoring example, 452–453
    - refactoring to, 577
  - fan-in, 80
  - fan-out, 81
  - farming metaphor, 14–15
  - fault tolerance, 50
  - feature-oriented integration, 700–701
  - Fibonacci numbers, 397–398
  - figures, list of, xxxiii

- plain if-then statements, 355–357
- refactoring, 573
- simplification, 445–447
- single-statement layout, 748–749
- tables, replacing with, 413–414
- types of, 355
- implicit declarations, 239–240
- implicit instancing, 132
- in keyword, creating, 175–176
- incomplete preparation, causes of, 25–27
- incremental development metaphor, 15–16
- incremental integration
  - benefits of, 693–694
  - bottom-up strategy, 697–698
  - classes, 694, 697
  - customer relations benefit, 694
  - defined, 692
  - disadvantages of top-down strategy, 695–696
  - errors, locating, 693
  - feature-oriented integration, 700–701
  - interface specification, 695, 697
  - progress monitoring benefit, 693
  - resources on, 708
  - results, early, 693
  - risk-oriented integration, 699
  - sandwich strategy, 698–699
  - scheduling benefits, 694
  - slices approach, 698
  - steps in, 692
  - strategies for, overview, 694
  - stubs, 694, 696
  - summary of approaches, 702
  - test drivers, 697
  - top-down strategy for, 694–696
  - T-shaped integration, 701
  - vertical-slice approach, 696
- indentation, 737, 764–768
- indexed access tables, 425–426, 428–429
- indexes, supplementing data types with, 627–628
- indexes, loop
  - alterations, 377
  - checklist, 389
  - enumerated types for, 305
  - final values, 377–378
  - scope of, 383–384
  - variable names, 265
- infinite loops, 367, 374
- informal reviews, 467, 492–493
- information hiding
  - access routines for, 340
  - ADTs for, 127
  - barriers to, 95–96
  - categories of secrets, 94
  - circular dependencies problem, 95
  - class data mistaken for global data, 95–96
  - class design considerations, 93
  - class implementation details, 153
  - example, 93–94
  - excessive distribution problem, 95
  - importance of, 92
  - interfaces, class, 93
  - performance issues, 96
  - privacy rights of classes, 92–93
  - resources for, 120
  - secrets concept, 92
  - type creation for, 313–314
- inheritance
  - access privileges from, 148
  - case statements, 147–148
  - checklist, 158
  - containment compared to, 143
  - decisions involved in, 144
  - deep trees, 147
  - defined, 144
  - design rule for, 144
  - functions, private, overriding, 146
  - guidelines, list of, 149
  - heuristic design with, 91–92
  - identifying as a design step, 88
  - is a relationships, 144
  - key points for, 160
  - Liskov Substitution Principle, 144–145
  - main goal of, 136
  - mixins, 149
  - multiple, 148–149
  - overridable vs. non-overridable routines, 145–146
  - parallel modifications refactoring indicator, 566
  - placement of common items in tree, 146
  - private vs. protected data, 148
  - private, avoiding, 143
  - recommended bias against, 149
  - routines overridden to do nothing, 146–147
  - single-instance classes, 146
  - similar sub and super classes, 576
- initializing variables
  - accumulators, 243
  - at declaration guideline, 241
  - C++ example, 241
  - checklist for, 257
  - class members, 243
  - compiler settings, 243
  - consequences of failing to, 240
  - const recommended, 243
  - constants, 243
  - counters, 243
  - declare and define near first use rule, 242–243
  - final recommended, 243
  - first use guideline, 241–242
  - fixing defects, 553
  - global variables, 337
  - importance of, 240–241
  - Java example, 242–243
  - key point, 258
  - loops, variables used in, 249
  - parameter validity, 244
  - pointer problems, 241, 244, 325–326
  - Principle of Proximity, 242
  - reinitialization, 243
  - strings, 300
  - system perturbers, testing with, 527
  - Visual Basic examples, 241–242
- initializing working memory, 244
- inline routines, 184–185
- input parameters, 274
- input/output. *See* I/O
- inspections. *See* formal inspections
- installation tools, 718
- instancing objects
  - ADTs, 132
  - factory method, 103–104
  - singleton, 104, 151
- integer data types
  - checklist, 316
  - costs of operations, 602
  - division considerations, 293
  - overflows, 293–295
  - ranges of, 294
- Integrated Development Environments (IDEs), 710–711
- integration
  - benefits of, 690–691, 693–694
  - big-bang, 691
  - bottom-up strategy, 697–698
  - broken builds, 703
  - checklist, 707

- integration, *continued*
    - classes, 691, 694, 697
    - continuous, 706
    - customer relations, 694
    - daily build and smoke test, 702–706
    - defined, 689
    - disadvantages of top-down strategy, 695–696
    - errors, locating, 693
    - feature-oriented strategy, 700–701
    - importance of approach methods, 689–691
    - incremental. *See* incremental integration
    - interface specification, 695, 697
    - key points, 708
    - monitoring, 693
    - phased, 691–692
    - resources on, 707–708
    - risk-oriented strategy, 699
    - sandwich strategy, 698–699
    - scheduling, 694
    - slices approach, 698
    - smoke tests, 703
    - strategies for, overview, 694
    - stubs, 694, 696
    - summary of approaches, 702
    - testing, 499, 697
    - top-down strategy for, 694–696
    - T-shaped integration, 701
    - unsurfaced work, 702
    - vertical-slice approach, 696
  - integrity, 464
  - intellectual honesty, 826–828
  - intellectual toolbox approach, [20](#)
  - intelligence, role in character, 821
  - interfaces, class
    - abstraction aspect of, 89, 133–138, 566
    - calls to classes, refactoring, 575
    - cohesion, 138
    - consistent level of abstraction, 135–136
    - delegation vs. inheritance, refactoring, 576
    - documenting, 713, 810
    - erosion under modification problem, 138
    - evaluating abstraction of, 135
    - extension classes, refactoring with, 576
    - formalizing as contracts, 106
    - good abstraction example, 133–134
    - guidelines for creating, 135–138
    - foreign routines, refactoring with, 576
    - inconsistency with members problem, 138
    - inconsistent abstraction, example of, 135–136
    - information hiding role, 93
    - integration, specification during, 695, 697
    - key points for, 160
    - layout of, 768
    - mixins, 149
    - objects, designing for, 89
    - opposites, pairs of, 137
    - poor abstraction example, 134–135
    - private details in, 139–141
    - programmatically preferred to semantic, 137
    - public routines in interfaces concern, 141
    - read-time convenience rule, 141
    - refactoring, 575–576, 579
    - routines, moving to refactor, 575
    - routines, unused, 576
    - semantic violations of
      - encapsulation, 141–142
      - unrelated information, handling, 137
  - interfaces, graphic. *See* GUIs
  - interfaces, routine. *See also* parameters of routines
    - commenting, 808
    - foreign routines, refactoring with, 576
    - pseudocode for, 226
    - public member variables, 576
    - routines, hiding, 576
    - routines, moving to refactor, 575
  - internationalization, 48
  - interoperability, 48
  - interpreted languages, performance of, 600–601
  - invalid input. *See* validation
  - iteration, code. *See also* loops
    - foreach loops, 367, 372
    - iterative data, 255
    - iterator loops, defined, 367
    - Iterator pattern, 104
    - structured programming concept of, 456
  - iteration in development
    - choosing, reasons for, 35–36
    - code tuning, 850
    - design practice, 111–117
    - Extreme Programming, 58
    - importance of, 850–851
    - prerequisites, 28, 33–34
    - sequential approach compared, 33–34
    - pseudocode component of, 219
- J**
- jamming loops, 617–618
  - Java
    - assertion example in, 190
    - boolean expression syntax, 443
    - description of, 65
    - exceptions, 198–201
    - layout recommended, 745
    - live time examples, 247–248
    - naming conventions for, 276, 277
    - parameters example, 176–177
    - persistence of variables, 251
    - resources for, 159
  - Javadoc, 807, 815
  - JavaScript, 65
  - JUnit, 531
  - just in time binding, 253
- K**
- key construction decisions. *See* construction decisions
  - killed data state, 509–510
  - kinds of software projects, 31–33
- L**
- languages, programming. *See* programming language choice
  - Law of Demeter, 150
  - layout
    - array references, 754
    - assignment statement
      - continuations, 758
    - begin-end pairs, 742–743
    - blank lines, 737, 747–748
    - block style, 738–743
    - brace styles, 734, 740–743
    - C++ side effects, 759–761
    - checklist, 773–774
    - classes, 768–771
    - closely related statement elements, 755–756
    - comments, 763–766
    - complicated expressions, 749–750
    - consistency requirement, 735

- continuing statements, 754–758
- control statement continuations, 757
- control structure styles, 745–752
- declarations, 761–763
- discourse rules, 733
- documentation in code, 763–766
- double indented begin-end pairs, 746–747
- emulating pure blocks, 740–743
- endline layout, 743–745, 751–752
- ends of continuations, 756–757
- files, within, 771–773
- Fundamental Theorem of Formatting, 732
- gotos, 750–751
- incomplete statements, 754–755
- indentation, 737
- interfaces, 768
- key points, 775
- language-specific guidelines, 745
- logical expressions, 753
- logical structure, reflecting, 732, 735
- mediocre example, 731–732
- misleading indentation example, 732–733
- misleading precedence, 733
- modifications guideline, 736
- multiple statements per line, 758–761
- negative examples, 730–731
- objectives of, 735–736
- parentheses for, 738
- pointers, C++, 763
- pure blocks style, 738–740
- readability goal, 735
- religious aspects of, 735
- resources on, 774–775
- routine arguments, 754
- routine call continuations, 756
- routine guidelines, 766–768
- self-documenting code, 778–781
- single-statement blocks, 748–749
- statement continuation, 754–758
- statement length, 753
- structures, importance of, 733–734
- styles overview, 738
- unindented begin-end pairs, 746
- violations of, commenting, 801
- Visual Basic blocking style, 738
- white space, 732, 736–737, 753–754
- laziness, 830
- lazy evaluation, 615–616
- leanness design goal, 81
- legal notices, 811
- length of variable names, optimum, 262
- levels of design
  - business logic subsystem, 85
  - classes, divisions into, 86
  - database access subsystem, 85
  - overview of, 82
  - packages, 82–85
  - routines, 86–87
  - software system, 82
  - subsystems, 82–85
  - system dependencies subsystem, 85
  - user interface subsystem, 85
- libraries, code
  - purpose of, 717
  - using functionality from, 222
- libraries, book. *See* software-development libraries
- life-cycle models
  - good practices table for, 31–32
  - development standard, 813
- linked lists
  - deleting pointers, 330
  - node insertion, 327–329
  - pointers, isolating operations of, 325
- linkers, 716
- lint tool, 557
- Liskov Substitution Principle (LSP), 144–145
- lists
  - of checklists, xxix–xxx
  - of figures, xxxiii
  - of tables, xxxi–xxxii
- literal data, 297–298, 308–309
- literate programs, 13
- live time of variables, 246–248, 459
- load time, binding during, 253
- localization
  - architecture prerequisites, 48
  - string data types, 298
- locking global data, 341
- logarithms, 632–634
- logging
  - defensive programming guideline, 210
  - tools for testing, 526
- logic coverage testing, 506
- logical cohesion, 170
- logical expressions. *See also* boolean expressions
  - code tuning, 610–616
  - comparing performance of, 614
- eliminating testing redundancy, 610–611
- frequency, testing in order of, 612–613
- identities, 630
- layout of, 753
- lazy evaluation, 615–616
- lookup tables, substituting, 614–615
- short-circuit evaluation, 610
- loops
  - abnormal, 371
  - arrays with, 387–388
  - bodies of, processing, 375–376, 388
  - brackets recommended, 375
  - break statements, 371–372, 379–380, 381
  - checklist, 388–389
  - code tuning, 616–624
  - commenting, 804–805
  - completion tests, location of, 368
  - compound, simplifying, 621–623
  - continuously evaluated loops, 367. *See also* while loops
  - continuation lines in, 757
  - continue statements, 379, 380, 381
  - counted loops, 367. *See also* for loops
  - cross talk, 383
  - defined, 367
  - designing, process for, 385–387
  - do loops, 369–370
  - empty, avoiding, 375–376
  - endless loops, 367, 374
  - endpoint considerations, 381–382
  - entering, guidelines for, 373–375, 388
  - enumerated types for, 305
  - exit guidelines, 369–372, 377–381, 389
  - for loops, 372, 374–378, 732–733, 746–747
  - foreach loops, 367, 372
  - fusion of, 617–618
  - goto with, 371
  - housekeeping statements, 376
  - index alterations, 377
  - index checklist, 389
  - index final values, 377–378
  - index variable names, 265
  - index scope, 383–384
  - infinite loops, 367, 374

loops, *continued*

- initialization code for, 373, 374
- iterative data structures with, 255
- iterator loops, 367, 456
- jamming, 617–618
- key points, 389
- kinds of, generalized, 367–368
- labeled break statements, 381
- language-specific, table of, 368
- length of, 385
- minimizing work inside, 620–621
- multiple break statements, 380
- naming variables, 382–383
- nested, 382–383, 385, 623
- null statements, rewriting, 445
- off-by-one errors, 381–382
- one-function guideline, 376
- order of nesting, 623
- performance considerations, 599
- pointers inside, 620
- problems with, overview of, 373
- pseudocode method, 385–387
- refactoring, 565, 573
- repeat until clauses, 377
- routines in, 385
- safety counters with, 378–379
- scope of indexes, 383–384
- sentinel tests for, 621–623
- size as refactoring indicator, 565
- strength reduction, 623–624
- switching, 616
- termination, making obvious, 377
- testing redundancy, eliminating, 610–611
- unrolling, 618–620
- unswitching, 616–617
- variable guidelines, 382–384
- variable initializations, 249
- variables checklist, 389
- verifying termination, 377
- while loops, 368–369

## loose coupling

- design goal, as, 80
- strategies for, 100–102

low-to-medium fan-out design goal, 81

LSP (Liskov Substitution Principle), 144–145

**M**

Macintosh naming conventions, 275

macro routines. *See also* routines

- alternatives for, 184
- limitations on, 184
- multiple statements in, 183

- naming, 183, 277–278
- parentheses with, 182–183

magazines on programming, 859–860

magic variables, avoiding, 292, 297–298, 308–309

maintenance

- comments requiring, 788–791
- design goal for, 80
- error-prone routines, prioritizing for, 518
- fixing defects, problems from, 553
- maintainability defined, 464
- readability benefit for, 842
- structures for reducing, 323

major construction practices

- checklist, 69–70

managing construction

- approaches. *See* approaches to development
- change control. *See* configuration management
- code ownership attitudes, 663
- complexity, 77–79
- configuration management. *See* configuration management
- good coding, encouraging, 662–664
- inspections, management role in, 486–487
- key points, 688
- managers, 686
- measurements, 677–680
- programmers, treatment of, 680–686
- readability standard, 664
- resources on, 687
- reviewing all code, 663
- rewarding good practices, 664
- schedules, estimating, 671–677
- signing off on code, 663
- size of projects, effects of. *See* size of projects
- standards, authority to set, 662
- standards, IEEE, 687, 814
- two-person teams, 662

markers, defects from, 787

matrices. *See* arrays

mature technology environments, 67

maximum normal configurations, 515

maze recursion example, 394–396

McCabe's complexity metric, 457, 458

measure twice, cut once, [23](#)

## measurement

- advantages of, 677
- arguing against, 678
- goals for, 679
- outlier identification, 679
- resources for, 679–680
- side effects of, 678
- table of useful types of, 678–679

memory

- allocation, error detection for, 206
- corruption by pointers, 325
- fillers, 244
- initializing working, 244
- paging operation performance impact, 599
- pointers, corruption by, 325
- tools for, 527

mentoring, 482

merge tools, 712

metaphors, software

- accreting a system, 15–16
- algorithmic use of, [11](#), [12](#)
- building metaphor, 16–19
- building vs. buying components, [18](#)
- combining, [20](#)
- computer-centric vs. data-centric views, [11](#)
- customization, [18](#)
- discoveries based on, 9–10
- earth centric vs. sun centric views, 10–11
- examples of, 13–20
- farming, 14–15
- growing a system, 14–15
- heuristic use of, [12](#)
- importance of, 9–11
- incremental development, 15–16
- key points for, 21
- modeling use for, [9](#)
- overextension of, [10](#)
- oyster farming, 15–16
- pendulum example, [10](#)
- power of, [10](#)
- readability, [13](#)
- relative merits of, [10](#), [11](#)
- simple vs. complex structures, 16–17
- size of projects, [19](#)
- throwing one away, 13–14
- toolbox approach, [20](#)
- using, 11–12
- writing code example, 13–14

methodologies, 657–659. *See also* approaches to development

methods. *See* routines

- metrics reporters, 714
- minimum normal configurations, 515
- mission-critical systems, 31–32
- mixed-language environments, 276
- mixins, 149
- mock objects, 523
- modeling, metaphors as. *See* metaphors, software
- moderator role in inspections, 486
- modularity
  - design goal of, 107
  - global variables, damage from, 337–338
- modules, coupling considerations, 100–102
- multiple inheritance, 148–149
- multiple returns from routines, 391–393
- multiple-file string search capability, 711–712

## N

- named constants. *See* constants
- naming conventions
  - “a” prefix convention, 272
  - abbreviating names, 282–285
  - abbreviation guidelines, 282
  - arrays, 280–281
  - benefits of, 270–271
  - C language, 275, 278
  - C++, 275–277
  - capitalization, 274, 286
  - case-insensitive languages, 273
  - characters, hard to read, 287
  - checklist, 288–289, 780
  - class member variables, 273
  - class vs. object names, 272–273
  - common operations, for, 172–173
  - constants, 273–274
  - cross-project benefits, 270
  - descriptiveness guideline, 171
  - documentation, 284–285, 778–780
  - enumerated types, 269, 274, 277–279
  - formality, degrees of, 271
  - files, 811
  - function return values, 172
  - global variables, 273, 342
  - homonyms, 286
  - Hungarian, 279
  - informal, 272–279
  - input parameters, 274
  - Java, 276, 277

- key points, 289
- kinds of information in names, 277
- language-independence
  - guidelines, 272–274
- length, not limiting, 171
- Macintosh, 275
- meanings in names, too similar, 285
- misleading names, 285
- misspelled words, 286
- mixed-language considerations, 276
- multiple natural languages, 287
- numbers, differentiating solely by, 171
- numerals, 286
- opposites, use of, 172
- parameters, 178
- phonic abbreviations, 283
- prefix standardization, 279–281
- procedure descriptions, 172
- proliferation reduction benefit, 270
- pronunciation guideline, 283
- purpose of, 270–271
- readability, 274
- relationships, emphasis of, 271
- reserved names, 287
- routines, 171–173, 222
- semantic prefixes, 280–281
- short names, 282–285, 288–289
- similarity of names, too much, 285
- spacing characters, 274
- `t_` prefix convention, 272
- thesaurus, using, 283
- types vs. variables names, 272–273
- UDT abbreviations, 279–280
- variables, for. *See* variable names
- Visual Basic, 278–279
- when to use, 271
- nested if statements
  - case statements, converting to, 448–449, 451
  - converting to if-then-else statements, 447–448
  - factoring to routines, 449–451
  - factory method approach, converting to, 452–453
  - functional decomposition of, 450–451
  - object-oriented approach, converting to, 452–453

- redesigning, 453
- simplification by retesting conditions, 445–446
- simplification with break blocks, 446–447
- summary of techniques for reducing, 453–454
- too many levels of, 445–454
- nested loops
  - designing, 382–383, 385
  - ordering for performance, 623
- nondeterministic nature of design process, 76, 87
- nonstandard language features, 98
- null objects, refactoring, 573
- null statements, 444–445
- numbers, literal, 292
- numeric data types
  - BCD, 297
  - checklist, 316
  - compiler warnings, 293
  - comparisons, 440–442
  - conversions, showing, 293
  - costs of operations, 602
  - declarations, commenting, 802
  - floating-point types, 295–297, 316, 602
  - hard coded 0s and 1s, 292
  - integers, 293–295
  - literal numbers, avoiding, 292
  - magic numbers, avoiding, 292
  - magnitudes, greatly different, operations with, 295
  - mixed-type comparisons, 293
  - overflows, 293–295
  - ranges of integers, 294
  - zero, dividing by, 292

## O

- objectives, software quality, 466, 468–469
- object-oriented programming
  - hiding information. *See* information hiding
  - inheritance. *See* inheritance
  - objects. *See* classes; objects
  - polymorphism. *See* polymorphism
  - resources for, 119, 159
- object-parameter coupling, 101
- objects
  - ADTs as, 130
  - attribute identification, 88



- prerequisites, upstream, *continued*
  - choosing between iterative and sequential approaches, 35–36
  - coding too early mistake, [25](#)
  - compelling argument for, 27–31
  - data arguing for, 28–30
  - error detection, doing early, 29–30
  - goal of, [25](#)
  - good practices table for, 31–32
  - importance of, [24](#)
  - incomplete preparation, causes of, 25–27
  - iterative and sequential mixes, 34–35
  - iterative methods with, 28, 33–34
  - key points for, 59–60
  - kinds of projects, 31–33
  - logical argument for, [27](#)
  - manager ignorance problem, [26](#)
  - problem definition, 36–38
  - requirements development. *See* requirements
  - risk reduction goal, [25](#)
  - skills required for success, [25](#)
  - time allowed for, 55–56
  - WIMP syndrome, [26](#)
  - WISCA syndrome, [26](#)
- Principle of Proximity, 242, 351
- private data, 148
- problem-definition prerequisites, 36–38
- problem domain, programming at, 845–847
- problem-solving skills development, 823
- procedural cohesion, 170
- procedures. *See also* routines
  - naming guidelines for, 172
  - when to use, 181–182
- processes, development. *See* approaches to development
- productivity
  - effects of good construction practice, [7](#)
  - industry average, 474
  - size of projects, effects on, 653
- professional development, 824–825
- professional organizations, [862](#)
- program flow
  - control of. *See* control structures
  - sequential. *See* straight-line code
- program organization prerequisite, 45–46
- program size. *See* size of projects
- programmers, character of. *See* character, personal
- programmers, treatment of. *See also* teams
  - overview, 680
  - physical environment, 684–685
  - privacy of offices, 684
  - religious issues, 683–684
  - resources on, 685–686
  - style issues, 683–684
  - time allocations, 681
  - variations in performance, 681–683
- programming conventions
  - choosing, 66
  - coding practices checklist, 69
  - formatting rules. *See* layout
- programming into languages, 68–69, 843
- programming language choice
  - Ada, 63
  - assembly language, 63
  - Basic, 65
  - C, 64
  - C#, 64
  - C++, 64
  - Cobol, 64
  - expressiveness of concepts, 63
  - familiar vs. unfamiliar languages, 62
  - Fortran, 64
  - higher- vs. lower-level language productivity, 62
  - importance of, 61–63
  - Java, 65
  - JavaScript, 65
  - Perl, 65
  - PHP, 65
  - productivity from, 62
  - programming into languages, 68–69, 843
  - Python, 65
  - ratio of statements compared to C code, table of, 62
  - SQL, 65
  - thinking, effects on, 63
  - Visual Basic, 65
- programming tools
  - assembler listing tools, 720
  - beautifiers, 712
  - build tools, 716–717
  - building your own, 721–722
  - CASE tools, 710
  - checklist, 724–725
  - class-hierarchy generators, 713
  - code libraries, 717
  - code tuning, 720
  - code-generation wizards, 718
  - compilers, 716
  - cross-reference tools, 713
  - data dictionaries, 715
  - debugging tools, 526–527, 545, 558–559, 719
  - dependency checkers, 716
  - design tools, 710
  - Diff tools, 712
  - disassemblers, 720
  - editing tools, 710–713
  - executable-code tools, 716–720
  - execution profiler tools, 720
  - fantasyland, 722–723
  - graphical design tools, 710
  - grep, 711
  - IDEs, 710–711
  - interface documentation, 713
  - key points, 725
  - linkers, 716
  - merge tools, 712
  - metrics reporters, 714
  - multiple-file string searches, 711–712
  - preprocessors, 718–719
  - project-specific tools, 721–722
  - purpose of, 709
  - quality analysis, 713–714
  - refactoring tools, 714–715
  - resources on, 724
  - restructuring tools, 715
  - scripts, 722
  - semantics checkers, 713–714
  - source-code tools, 710–715
  - syntax checkers, 713–714
  - templates, 713
  - testing tools, 719
  - tool-oriented environments, 720–721
  - translators, 715
  - version control tools, 715
- project types, prerequisites corresponding to, 31–33
- protected data, 148
- prototyping, 114–115, 468
- Proximity, Principle of, 242, 351
- pseudocode
  - algorithms, researching, 223
  - bad, example of, 218–219
  - benefits from, 219–220
  - changing, efficiency of, 220
  - checking for errors, 230–231
  - checklist for PPP, 233–234

classes, steps in creating, 216–217  
 coding below comments,  
   227–229  
 coding from, 225–229  
 comments from, 220, 791  
 data structure for routines, 224  
 declarations from, 226  
 defined, 218  
 designing routines, 220–225  
 error handling considerations,  
   222  
 example for routines, 224  
 functionality from libraries, 222  
 good, example of, 219  
 guidelines for effective use, 218  
 header comments for routines,  
   223  
 high-level comments from,  
   226–227  
 iterative refinement, 219, 225  
 key points for creating, 234  
 loop design, 385–387  
 naming routines, 222  
 performance considerations,  
   222–223  
 PPP. *See* PPP  
 prerequisites, 221  
 problem definition, 221  
 refactoring, 229  
 reviewing, 224–225  
 routines, steps in creating, 217,  
   223–224  
 testing, planning for, 222  
 Pseudocode Programming Process.  
   *See* PPP  
 psychological distance, 556  
 psychological set, 554–555  
 psychological factors. *See* character,  
   personal  
 public data members, 567  
 pure blocks layout style, 738–740  
 Python  
   description of, 65  
   performance issues, 600

## Q

quality assurance. *See also* quality of  
 software  
 checklist, 70  
 good practices table for, 31–32  
 prerequisites role in, [24](#)  
 requirements checklist, 42–43  
 quality gates, 467

quality of software  
 accuracy, 464  
 adaptability, 464  
 change-control procedures, 468  
 checklist for, 476  
 collaborative construction. *See*  
   collaboration  
 correctness, 463  
 costs of finding defects, 472  
 costs of fixing defects, 472–473  
 debugging, role of, 474–475, 536  
 detection of defects by various  
   techniques, table of, 470  
 development process assurance  
   activities, 467–468  
 efficiency, 464  
 engineering guidelines, 467  
 explicit activity for, 466  
 external audits, 467  
 external characteristics of,  
   463–464  
 Extreme Programming, 471–472  
 flexibility, 464  
 gates, 467  
 General Principle of Software  
   Quality, 474–475  
 integrity, 464  
 internal characteristics, 464–465  
 key points, 477  
 maintainability, 464  
 measurement of results, 468  
 multiple defect detection  
   techniques recommended,  
   470–471  
 objectives, setting, 466, 468–469  
 optimization conflicts, 465–466  
 percentage of defects  
   measurement, 469–472  
 portability, 464  
 programmer performance,  
   objectives based, 468–469  
 prototyping, 468  
 readability, 464  
 recommended combination for,  
   473  
 relationships of characteristics,  
   465–466  
 reliability, 464  
 resources for, 476  
 reusability, 464  
 reviews, 467  
 robustness, 464  
 standards, IEEE, 477, 814  
 testing, 465, 467, 500–502

understandability, 465  
 usability, 463  
 when to do assurance of, 473

## R

random-data generators, 525  
 readability  
   as management standard, 664  
   defects exposing lack of, 538  
   defined, 464  
   formatting for. *See* layout  
   importance of, [13](#), 841–843  
   maintenance benefit from, 842  
   naming variables for. *See* naming  
   conventions; variable names  
   positive effects from, 841  
   private vs. public programs, 842  
   professional development,  
   importance to, 825  
   structures, importance of,  
   733–734  
   warning sign, as a, 849  
 reading as a skill, 824  
 reading plan for software  
   developers, 860–862  
 records, refactoring, 572  
 recursion  
   alternatives to, 398  
   checklist, 410  
   defined, 393  
   factorials using, 397–398  
   Fibonacci numbers using,  
   397–398  
   guidelines for, 394  
   key points, 410  
   maze example, 394–396  
   safety counters for, 396  
   single routine guideline, 396  
   sorting example, 393–394  
   stack space concerns, 397  
   terminating, 396  
 refactoring  
   80/20 rule, 582  
   adding routines, 582  
   algorithms, 573  
   arrays, 572  
   backing up old code, 579  
   bidirectional class associations,  
   577  
   boolean expressions, 572  
   case statements, 573  
   checklists for, 570, 577–579  
   checkpoints for, 580

- refactoring, *continued*
  - class cohesion indicator, 566
  - class interfaces, 575–576
  - classes, 566–567, 574–576, 578–579, 582
  - code tuning, compared to, 609
  - collections, 572
  - comments on bad code, 568
  - complex modules, 583
  - conditional expressions, 573
  - constant values varying among subclass, 574
  - constructors to factory methods, 577
  - data from uncontrolled sources, 576
  - data sets, related, as indicator, 566
  - data types to classes, 572
  - data-level, 571–572, 577
  - defects, fixes of, 582
  - defined, 565
  - designing code for future needs, 569–570
  - Don't Repeat Yourself principle, 565
  - duplicate code indicator, 565
  - error-prone modules, 582
  - expressions, 571
  - global variables, 568
  - GUI data, 576
  - if statements, 573
  - interfaces, 566, 575–576, 579
  - key points, 585
  - listing planned steps, 580
  - literal constants, 571
  - loops, 565, 573
  - maintenance triggering, 583
  - middleman classes, 567
  - misuse of, 582
  - null objects, 573
  - objects, 574–576
  - one-at-a-time rule, 580
  - overloaded primitive data types, 567
  - parallel modifications required indicator, 566
  - parameters, 566, 571, 573
  - PPP coding step, 229
  - public data members, 567
  - queries, 574
  - reasons not to, 571
  - records, 572
  - redesigning instead of, 582
  - reference objects, 574
  - resources on, 585
  - reviews of, 580–581
  - risk levels of, 581
  - routines, 565–567, 573–574, 578, 582
  - safety guidelines, 579–581, 584
  - setup code, 568–569
  - size guideline, 580
  - statement-level, 572–573, 577–578
  - strategies for, 582–584
  - subclasses, 567, 575
  - superclasses, 575
  - system-level, 576–577, 579
  - takedown code, 568–569
  - testing, 580
  - to do lists for, 580
  - tools for, 714–715
  - tramp data, 567
  - ugly code, interfaces to, 583–584
  - unidirectional class associations, 577
  - unit tests for, 580
  - variables, 571
  - warnings, compiler, 580
- references (&), C++, 332
- regression testing
  - diff tools for, 524
  - defined, 500
  - purpose of, 528
- reliability
  - cohesive routines, 168
  - defined, 464
- religious attitude toward programming
  - eclecticism, 851–852
  - experimentation compared to, 852–853
  - harmful effects of, 851–853
  - layout styles becoming, 735
  - managing people, 683–684
  - software oracles, 851
- reports. *See* formal inspections
- requirements
  - benefits of, 38–39
  - business cases for, [41](#)
  - change-control procedures, 40–41
  - checklists for, [40](#), 42–43
  - coding without, [26](#)
  - communicating changes in, 40–41
  - completeness, checklist, 43
  - configuration management of, 664, 666–667
  - defined, [38](#)
  - development approaches with, [41](#)
  - development process effects on, [40](#)
  - dumping projects, [41](#)
  - errors in, effects of, 38–39
  - functional, checklist, 42
  - good practices table for, 31–32
  - importance of, 38–39
  - key point for, 60
  - nonfunctional, checklist, 42
  - performance tuning, 589
  - quality, checklist, 42–43
  - rate of change, typical, 563
  - resources on developing, 56–57
  - stability of, 39–40, 840
  - testing for, 503
  - time allowed for, 55–56
- resource management
  - architecture for, 47
  - cleanup example, 401–402
  - restrictive nature of design, 76
  - restructuring tools, 715
  - retesting. *See* regression testing
- return statements
  - checklist, 410
  - guard clauses, 392–393
  - key points, 410
  - multiple, from one routine, 391–393
  - readability, 391–392
  - resources for, 408
- reusability
  - defined, 464
  - architecture prerequisites, [52](#)
- reviewer role in inspections, 486
- reviews
  - code reading, 494
  - dog-and-pony shows, 495
  - educational aspect of, 482
  - every line of code rule, 663
  - formal inspections, compared to, 485
  - formal, quality from, 467
  - informal, defined, 467
  - iteration process, place in, 850
  - refactoring conducting after, 580–581
  - walk-throughs, 492–493
- right shifting, 634
- risk-oriented integration, 699
- robustness
  - architecture prerequisites, 51
  - assertions with error handling, 193–194
  - correctness, balanced against, 197
  - defined, 197, 464

- rounding errors, 297
- routines
  - abstract overridable, 145
  - abstraction benefit, 164
  - abstraction with object parameters, 179, 574
  - access. *See* access routines
  - algorithm selection for, 223, 573
  - alternates to PPP, 232–233
  - black-box testing of, 502
  - blank lines in, 766
  - boolean test benefit, 165
  - calculation to function example, 166–167
  - calls, costs of, 601
  - checking for errors, 230–231
  - checklists, 185, 774, 780
  - classes, converting to, criteria for, 573
  - cleanup steps, 232
  - code tuning, 639–640
  - coding from pseudocode, 225–229
  - cohesion, 168–171
  - coincidental cohesion, 170
  - commenting, 805–809, 817
  - communicational cohesion, 169
  - compiling for errors, 230–231
  - complexity metric, 458
  - complexity reduction benefit, 164
  - construction step for classes, 217
  - continuations in call lines, 756
  - coupling considerations, 100–102
  - data states, 509
  - data structures for, 224
  - declarations, 226
  - defined, 161
  - descriptiveness guideline for naming, 171
  - design by contract, 233
  - designing, 86, 220–225
  - documentation, 178, 780
  - downcast objects, 574
  - duplication benefit, 164–165
  - endline layout, 767
  - error handling considerations, 222
  - errors in, relation to length of, 173
  - event handlers, 170
  - fields of objects, passing to, 574
  - files, layout in, 772
  - functional cohesion, 168–169
  - functionality from libraries, 222
  - functions, special considerations
    - for, 181–182
  - hacking approach to, 233
  - header comments for, 223
  - high quality, counterexample, 161–163
  - high-level comments from pseudocode, 226–227
  - importance of, 163
  - in keyword creation, 175–176
  - indentation of, 766–768
  - internal design, 87
  - inline, 184–185
  - input-modify-output parameter order, 174–175
  - interface statements, 226
  - iterating pseudocode, 225
  - key points for, 186, 234
  - layout of, 754, 766–768
  - length of, guideline for, 173–174
  - limitations, documenting, 808
  - logical cohesion, 170
  - low-quality example, 161–163
  - macro. *See* macro routines
  - mentally checking for errors, 230
  - multiple returns from, 391–393
  - named parameters in, 180
  - naming, 171–173, 222, 277–278, 567
  - nested deeply, 164
  - objects, passing to, 179, 574
  - out keyword creation, 175–176
  - overridable vs. non-overridable routines, 145–146
  - overridden to do nothing, 146–147
  - overriding, 156
  - parameters. *See* parameters of routines
  - performance considerations, 165, 222–223
  - pointer hiding benefit, 165
  - portability benefit, 165
  - postconditions, 221
  - PPP checklist for, 233–234
  - preconditions, 221
  - prerequisites, 221
  - problem definition, 221
  - procedural cohesion, 170
  - procedure naming guideline, 172
  - pseudocode writing step, 223–224
  - public, using in interfaces concern, 141
  - queries, refactoring, 574
  - reasons for creating, list of, 167
  - refactoring, 229, 573–575, 578, 582
  - reliability from cohesiveness, 168
  - removing errors, 231
  - repeating steps, 232
  - returns from, multiple, 391–393
  - reviewing pseudocode, 224–225
  - sequence hiding benefit, 165
  - sequential cohesion, 168
  - setup code for, refactoring, 568–569
  - similar parameters, order for, 176
  - similar, refactoring, 574
  - simple, usefulness of, 166–167
  - size as refactoring indicator, 565–566
  - small vs. large, 166, 173–174
  - specification example, 221
  - stepping through code, 231
  - strength, 168
  - subclassing benefit, 165
  - temporal cohesion, 169
  - test-first development, 233
  - testing, 222, 231, 523
  - tramp data in, 567
  - unused, refactoring, 576
  - valid reasons for creating, 164–167
  - variable names, differentiating from, 272
  - wrong class, indicator for, 566
  - run time, binding during, 253
- S**
  - safety counters in loops, 378–379
  - sandwich integration, 698–699
  - scaffolding
    - debugging with, 558
    - testing, 523–524, 531
  - scalability, 48. *See also* size of projects
  - scientific method, classic steps in, 540
  - SCM (software configuration management), 665. *See also* configuration management
  - schedules, estimating. *See* estimating schedules
  - scope of variables
    - convenience argument, 250
    - defined, 244
    - global scope, problems with, 251

scope of variables, *continued*

- grouping related statements, 249–250
  - key point, 258
  - language differences, 244
  - live time, minimizing, 246–248
  - localizing references to variables, 245
  - loop initializations, 249
  - manageability argument, 251
  - minimizing, guidelines for, 249–251
  - restrict and expand tactic, 250
  - span of variables, 245
  - value assignments, 249
  - variable names, effects on, 262–263
- scribe role in inspections, 486
- scripts
- programming tools, as, 722
  - slowness of, 600–601
- SDFs (software development folders), 778
- security, 47
- selections, code, 455
- selective data, 254
- self-documenting code, 778–781, 796–797
- semantic coupling, 102
- semantic prefixes, 280–281
- semantics checkers, 713–714
- sentinel tests for loops, 621–623
- sequences, code. *See also* blocks
- hiding with routines, 165
  - order of. *See* dependencies, code-ordering
  - structured programming concept of, 454
- sequential approach, 33–36
- sequential cohesion, 168
- Set() routines, 576
- setup code, refactoring, 568–569
- setup tools, 718
- short-circuit evaluation, 438–440, 610
- side effects, C++, 759–761
- signing off on code, 663
- simple-data-parameter coupling, 101
- simple-object coupling, 101
- single points of control, 308
- single-statement blocks, 748–749
- singleton property, enforcing, 104, 151

size of projects

- activities, list of fastest growing, 655
  - activity types, effects on, 654–655
  - building metaphor for, [19](#)
  - communications between people, 650
  - complexity, effect of, 656–657
  - defects created, effects on, 651–653
  - documentation requirements, 657
  - estimation errors, 656–657
  - formality requirements, 657
  - key points, 659
  - methodology considerations, 657–658
  - overview, 649
  - productivity, effects on, 653
  - ranges in, 651
  - resources on, 658–659
  - single product, multiple users, 656
  - single program, single user, 656
  - system products, 656
  - systems, 656
- sizeof(), 335
- sloppy processes, 75–76
- smart pointers, 334
- smoke tests, 703
- software accretion metaphor, 15–16
- software construction overview
- activities excluded from, [6](#)
  - activities in, list of, [3](#)
  - centralness to development process, [7](#)
  - defined, 3–6
  - documentation by source code, [7](#)
  - guaranteed done nature of, [7](#)
  - importance of, 6–7
  - key points for, [8](#)
  - main activities of, [4](#)
  - percent of total development process, [7](#)
  - productivity, importance in, [7](#)
  - programming as, [5](#)
  - programming vs., [4](#)
  - source code as documentation, [7](#)
  - tasks in, list of, [5](#)
- software design. *See* design
- software development folders (SDFs), 778
- software engineering overview of resources, 858

software evolution

- background for, 563–564
  - Cardinal Rule of, 565
  - construction vs. maintenance, 564
  - improving vs. degrading direction of, 564
  - philosophy of, 564–565
- software metaphors. *See* metaphors, software
- software oracles, 851
- software quality. *See* quality of software
- Software's Primary Technical Imperative, 92
- software-development libraries
- bibliographies, 858
  - construction, 856
  - magazines, 859–860
  - overview, 855, 857–858
  - reading plan, 860–862
  - software engineering overviews, 858
- software-engineering guidelines, 467
- sorting, recursive algorithm for, 393–394
- source code
- documentation aspect of, [7](#)
  - resource for, 815
- source-code tools
- analyzing quality, 713–714
  - beautifiers, 712
  - class-hierarchy generators, 713
  - comparators, 556
  - cross-reference tools, 713
  - data dictionaries, 715
  - Diff tools, 712
  - editing tools, 710–713
  - grep, 711
  - IDEs, 710–711
  - interface documentation, 713
  - merge tools, 712
  - metrics reporters, 714
  - multiple-file string searches, 711–712
  - refactoring tools, 714–715
  - restructuring tools, 715
  - semantics checkers, 713–714
  - syntax checkers, 713–714
  - templates, 713
  - translators, 715
  - version control tools, 715
- span, 245, 459

- languages with, evaluation of, 314–315
  - modification benefit, 314
  - naming conventions, 315
  - Pascal example, 312–313
  - portability benefit, 315–316
  - predefined types, avoiding, 315
  - purpose of, 311–312
  - reasons for, 314
  - redefining predefined, 315
  - reliability benefit, 314
  - validation benefit, 314
  - type definitions, 278
- U**
- UDFs (unit development folders), 778
  - UDT (user-defined type)
    - abbreviations, 279–280
  - UML diagrams, 118, 120
  - understandability, 465. *See also* readability
  - Unicode, 288–299
  - unit development folders (UDFs), 778
  - unit testing, 499
  - UNIX programming environment, 720
  - unrolling loops, 618–620
  - unswitching loops, 616–617
  - upstream prerequisites. *See* prerequisites, upstream
  - usability, 463
  - used data state, 509–510
  - user-defined type (UDT)
    - abbreviations, 279–280
  - user interfaces
    - architecture prerequisites, 47
    - refactoring data from, 576
    - subsystem design, 85
- V**
- validation
    - assumptions to check, list of, 190
    - data types, suspicious, 188
    - enumerated types for, 304–305
    - external data sources rule, 188
    - input parameters rule, 188
  - variable names
    - abbreviation guidelines, 282
  - accurate description rule, 260–261
  - bad names, examples of, 259–260, 261
  - boolean variables, 268–269
  - C language, 275, 278
  - C++, 263, 275–277
  - capitalization, 286
  - characters, hard to read, 287
  - checklist, 288–289
  - class member variables, 273
  - computed-value qualifiers, 263–264
  - constants, 270
  - enumerated types, 269
  - full description rule, 260–261
  - global, qualifiers for, 263
  - good names, examples of, 260, 261
  - homonyms, 286
  - Java conventions, 277
  - key points, 289
  - kinds of information in, 277
  - length, optimum, 262
  - loop indexes, 265
  - misspelled words, 286
  - multiple natural languages, 287
  - namespaces, 263
  - numerals in, 286
  - opposite pairs for, 264
  - phonic abbreviations, 283
  - problem orientation rule, 261
  - psychological distance, 556
  - purpose of, 240
  - reserved names, 287
  - routine names, differentiating from, 272
  - scope, effects of, 262–263
  - similarity of names, too much, 285
  - specificity rule, 261
  - status variables, 266–267
  - temporary variables, 267–268
  - type names, differentiating from, 272–273
  - Visual Basic, 279
  - variables
    - binding time for, 252–254
    - change, identifying areas of, 98–99
    - checklist for using, 257–258
    - comments for, 803
    - counters, 243
    - data literacy test, 238–239
    - data type relationship to control structures, 254–255
    - declaring. *See* declarations
    - global. *See* global variables
    - hidden meanings, avoiding, 256–257
    - hybrid coupling, 256–257
    - implicit declarations, 239–240
    - initializing, 240–244, 257
    - iterative data, 255
    - key points, 258
    - live time, 246–248, 459
    - localizing references to, 245
    - looping, 382–384
    - naming. *See* variable names
    - persistence of, 251–252
    - Principle of Proximity, 242
    - public class members, 576
    - refactoring, 571, 576
    - reusing, 255–257
    - scope of. *See* scope of variables
    - selective data, 254
    - sequential data, 254
    - span of, 245
    - types of. *See* data types
    - using all declared, 257
  - version control
    - commenting, 811
    - debugging aid removal, 207
    - tools for, 668, 715
  - visibility. *See also* scope of variables
    - coupling criteria for, 100
    - classes, of, 93
  - vision statement prerequisites. *See* problem definition prerequisites
- Visual Basic
- assertion examples, 192–194
  - blocking style, 738
  - case-insensitivity, 273
  - description of, 65
  - enumerated types, 303–306
  - exceptions in, 198–199, 202
  - implicit declarations, turning off, 240
  - layout recommended, 745
  - naming conventions for, 278–279
  - parameters example, 180
  - resources for, 159
  - structures, 320–322

## W

- walk-throughs, 492–493, 495–496
- warning signs, 848–850
- while loops
  - advantages of, 374–375
  - break statements, 379
  - do-while loops, 369
  - exits in, 369–372
  - infinite loops, 374
  - misconception of evaluation, 554
  - null statements with, 444
- white space
  - purpose of, 368
  - tests, position of, 369
- white space
  - blank lines, 737, 747–748
  - defined, 732
  - grouping with, 737
  - importance of, 736
  - indentation, 737
  - individual statements with, 753–754
- white-box testing, 500, 502
- wicked problems, 74–75
- Wikis, 117
- WIMP syndrome, [26](#)
- WISCA syndrome, [26](#)
- workarounds, documenting, 800
- writing metaphor for coding, 13–14

## Z

- zero, dividing by, 292