

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.

15 16 17 18 19 20 21 22 23 24 QGT 6 5 4 3 2 1

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

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

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

*To my wife, Ashlie, who doesn't have much to do with computer programming
but who has everything to do with enriching the rest of my life
in more ways than I could possibly describe*

Further Praise for **Code Complete**

“An excellent guide to programming style and software construction.”

—Martin Fowler, *Refactoring*

“Steve McConnell’s *Code Complete* . . . provides a fast track to wisdom for programmers. . . . His books are fun to read, and you never forget that he is speaking from hard-won personal experience.” —Jon Bentley, *Programming Pearls*, 2d ed.

“This is simply the best book on software construction that I’ve ever read. Every developer should own a copy and read it cover to cover every year. After reading it annually for nine years, I’m still learning things from this book!”

—John Robbins, *Debugging Applications for Microsoft .NET and Microsoft Windows*

“Today’s software *must* be robust and resilient, and secure code starts with disciplined software construction. After ten years, there is still no better authority than *Code Complete*.”

—Michael Howard, Security Engineering, Microsoft Corporation; Coauthor, *Writing Secure Code*

“A comprehensive examination of the tactical issues that go into crafting a well-engineered program. McConnell’s work covers such diverse topics as architecture, coding standards, testing, integration, and the nature of software craftsmanship.”

—Grady Booch, *Object Solutions*

“The ultimate encyclopedia for the software developer is *Code Complete* by Steve McConnell. Subtitled ‘A Practical Handbook of Software Construction,’ this 850-page book is exactly that. Its stated goal is to narrow the gap between the knowledge of ‘industry gurus and professors’ (Yourdon and Pressman, for example) and common commercial practice, and ‘to help you write better programs in less time with fewer headaches.’ . . . Every developer should own a copy of McConnell’s book. Its style and content are thoroughly practical.”

—Chris Loosley, *High-Performance Client/Server*

“Steve McConnell’s seminal book *Code Complete* is one of the most accessible works discussing in detail software development methods. . . .”

—Erik Bethke, *Game Development and Production*

“A mine of useful information and advice on the broader issues in designing and producing good software.”

—John Dempster, *The Laboratory Computer: A Practical Guide for Physiologists and Neuroscientists*

“If you are serious about improving your programming skills, you should get *Code Complete* by Steve McConnell.”

—Jean J. Labrosse, *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C*

“Steve McConnell has written one of the best books on software development independent of computer environment . . . *Code Complete*.”

—Kenneth Rosen, *Unix: The Complete Reference*

“Every half an age or so, you come across a book that short-circuits the school of experience and saves you years of purgatory. . . . I cannot adequately express how good this book really is. *Code Complete* is a pretty lame title for a work of brilliance.”

—Jeff Duntemann, *PC Techniques*

“Microsoft Press has published what I consider to be the definitive book on software construction. This is a book that belongs on every software developer’s shelf.”

—Warren Keuffel, *Software Development*

“Every programmer should read this outstanding book.” —T. L. (Frank) Pappas, *Computer*

“If you aspire to be a professional programmer, this may be the wisest \$35 investment you’ll ever make. Don’t stop to read the rest of this review: just run out and buy it. McConnell’s stated purpose is to narrow the gap between the knowledge of industry gurus and common commercial practice. . . . The amazing thing is that he succeeds.”

—Richard Mateosian, *IEEE Micro*

“*Code Complete* should be required reading for anyone . . . in software development.”

—Tommy Usher, *C Users Journal*

“I’m encouraged to stick my neck out a bit further than usual and recommend, without reservation, Steve McConnell’s *Code Complete*. . . . My copy has replaced my API reference manuals as the book that’s closest to my keyboard while I work.”

—Jim Kyle, *Windows Tech Journal*

“This well-written but massive tome is arguably the best single volume ever written on the practical aspects of software implementation.”

—Tommy Usher, *Embedded Systems Programming*

“This is the best book on software engineering that I have yet read.”

—Edward Kenworth, *.EXE Magazine*

“This book deserves to become a classic, and should be compulsory reading for all developers, and those responsible for managing them.” —Peter Wright, *Program Now*

Code Complete, Second Edition

Steve McConnell

Contents at a Glance

Part I Laying the Foundation

- 1 Welcome to Software Construction3
- 2 Metaphors for a Richer Understanding of Software Development9
- 3 Measure Twice, Cut Once: Upstream Prerequisites..... 23
- 4 Key Construction Decisions 61

Part II Creating High-Quality Code

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

Part III Variables

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

Part IV Statements

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

Part V Code Improvements

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

Part VI System Considerations

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

Part VII Software Craftsmanship

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

Table of Contents

Preface	xix
Acknowledgments	xxvii
List of Checklists	xxix
List of Tables	xxxix
List of Figures	xxxiii

Part I Laying the Foundation

1	Welcome to Software Construction	3
	1.1 What Is Software Construction?	3
	1.2 Why Is Software Construction Important?	6
	1.3 How to Read This Book	8
2	Metaphors for a Richer Understanding of Software Development	9
	2.1 The Importance of Metaphors	9
	2.2 How to Use Software Metaphors	11
	2.3 Common Software Metaphors	13
3	Measure Twice, Cut Once: Upstream Prerequisites	23
	3.1 Importance of Prerequisites	24
	3.2 Determine the Kind of Software You're Working On	31
	3.3 Problem-Definition Prerequisite	36
	3.4 Requirements Prerequisite	38
	3.5 Architecture Prerequisite	43
	3.6 Amount of Time to Spend on Upstream Prerequisites	55
4	Key Construction Decisions	61
	4.1 Choice of Programming Language	61
	4.2 Programming Conventions	66
	4.3 Your Location on the Technology Wave	66
	4.4 Selection of Major Construction Practices	69

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/

Part II Creating High-Quality Code

5	Design in Construction	73
	5.1 Design Challenges	74
	5.2 Key Design Concepts	77
	5.3 Design Building Blocks: Heuristics	87
	5.4 Design Practices	110
	5.5 Comments on Popular Methodologies	118
6	Working Classes	125
	6.1 Class Foundations: Abstract Data Types (ADTs)	126
	6.2 Good Class Interfaces	133
	6.3 Design and Implementation Issues	143
	6.4 Reasons to Create a Class	152
	6.5 Language-Specific Issues	156
	6.6 Beyond Classes: Packages	156
7	High-Quality Routines	161
	7.1 Valid Reasons to Create a Routine	164
	7.2 Design at the Routine Level	168
	7.3 Good Routine Names	171
	7.4 How Long Can a Routine Be?	173
	7.5 How to Use Routine Parameters	174
	7.6 Special Considerations in the Use of Functions	181
	7.7 Macro Routines and Inline Routines	182
8	Defensive Programming	187
	8.1 Protecting Your Program from Invalid Inputs	188
	8.2 Assertions	189
	8.3 Error-Handling Techniques	194
	8.4 Exceptions	198
	8.5 Barricade Your Program to Contain the Damage Caused by Errors	203
	8.6 Debugging Aids	205
	8.7 Determining How Much Defensive Programming to Leave in Production Code	209
	8.8 Being Defensive About Defensive Programming	210

9	The Pseudocode Programming Process	215
	9.1 Summary of Steps in Building Classes and Routines	216
	9.2 Pseudocode for Pros	218
	9.3 Constructing Routines by Using the PPP	220
	9.4 Alternatives to the PPP	232

Part III Variables

10	General Issues in Using Variables.	237
	10.1 Data Literacy	238
	10.2 Making Variable Declarations Easy	239
	10.3 Guidelines for Initializing Variables	240
	10.4 Scope	244
	10.5 Persistence	251
	10.6 Binding Time	252
	10.7 Relationship Between Data Types and Control Structures	254
	10.8 Using Each Variable for Exactly One Purpose	255
11	The Power of Variable Names	259
	11.1 Considerations in Choosing Good Names	259
	11.2 Naming Specific Types of Data	264
	11.3 The Power of Naming Conventions	270
	11.4 Informal Naming Conventions	272
	11.5 Standardized Prefixes	279
	11.6 Creating Short Names That Are Readable	282
	11.7 Kinds of Names to Avoid	285
12	Fundamental Data Types	291
	12.1 Numbers in General	292
	12.2 Integers	293
	12.3 Floating-Point Numbers	295
	12.4 Characters and Strings	297
	12.5 Boolean Variables	301
	12.6 Enumerated Types	303
	12.7 Named Constants	307
	12.8 Arrays	310
	12.9 Creating Your Own Types (Type Aliasing)	311

13	Unusual Data Types	319
	13.1 Structures	319
	13.2 Pointers	323
	13.3 Global Data	335
Part IV Statements		
14	Organizing Straight-Line Code	347
	14.1 Statements That Must Be in a Specific Order	347
	14.2 Statements Whose Order Doesn't Matter	351
15	Using Conditionals	355
	15.1 <i>if</i> Statements	355
	15.2 <i>case</i> Statements	361
16	Controlling Loops	367
	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
17	Unusual Control Structures	391
	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
18	Table-Driven Methods	411
	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
19	General Control Issues	431
	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

20 The Software-Quality Landscape	463
20.1 Characteristics of Software Quality	463
20.2 Techniques for Improving Software Quality	466
20.3 Relative Effectiveness of Quality Techniques	469
20.4 When to Do Quality Assurance	473
20.5 The General Principle of Software Quality	474
21 Collaborative Construction	479
21.1 Overview of Collaborative Development Practices	480
21.2 Pair Programming	483
21.3 Formal Inspections	485
21.4 Other Kinds of Collaborative Development Practices	492
22 Developer Testing	499
22.1 Role of Developer Testing in Software Quality	500
22.2 Recommended Approach to Developer Testing	503
22.3 Bag of Testing Tricks	505
22.4 Typical Errors	517
22.5 Test-Support Tools	523
22.6 Improving Your Testing	528
22.7 Keeping Test Records	529
23 Debugging	535
23.1 Overview of Debugging Issues	535
23.2 Finding a Defect	540
23.3 Fixing a Defect	550
23.4 Psychological Considerations in Debugging	554
23.5 Debugging Tools—Obvious and Not-So-Obvious	556

24	Refactoring	563
	24.1 Kinds of Software Evolution.	564
	24.2 Introduction to Refactoring.	565
	24.3 Specific Refactorings.	571
	24.4 Refactoring Safely.	579
	24.5 Refactoring Strategies	582
25	Code-Tuning Strategies.	587
	25.1 Performance Overview.	588
	25.2 Introduction to Code Tuning	591
	25.3 Kinds of Fat and Molasses	597
	25.4 Measurement.	603
	25.5 Iteration	605
	25.6 Summary of the Approach to Code Tuning	606
26	Code-Tuning Techniques	609
	26.1 Logic	610
	26.2 Loops.	616
	26.3 Data Transformations.	624
	26.4 Expressions.	630
	26.5 Routines	639
	26.6 Recoding in a Low-Level Language	640
	26.7 The More Things Change, the More They Stay the Same	643

Part VI System Considerations

27	How Program Size Affects Construction	649
	27.1 Communication and Size.	650
	27.2 Range of Project Sizes	651
	27.3 Effect of Project Size on Errors	651
	27.4 Effect of Project Size on Productivity.	653
	27.5 Effect of Project Size on Development Activities	654

28	Managing Construction	661
	28.1 Encouraging Good Coding	662
	28.2 Configuration Management	664
	28.3 Estimating a Construction Schedule	671
	28.4 Measurement	677
	28.5 Treating Programmers as People	680
	28.6 Managing Your Manager	686
29	Integration	689
	29.1 Importance of the Integration Approach	689
	29.2 Integration Frequency—Phased or Incremental?	691
	29.3 Incremental Integration Strategies	694
	29.4 Daily Build and Smoke Test	702
30	Programming Tools	709
	30.1 Design Tools	710
	30.2 Source-Code Tools	710
	30.3 Executable-Code Tools	716
	30.4 Tool-Oriented Environments	720
	30.5 Building Your Own Programming Tools	721
	30.6 Tool Fantasyland	722
Part VII	Software Craftsmanship	
31	Layout and Style	729
	31.1 Layout Fundamentals	730
	31.2 Layout Techniques	736
	31.3 Layout Styles	738
	31.4 Laying Out Control Structures	745
	31.5 Laying Out Individual Statements	753
	31.6 Laying Out Comments	763
	31.7 Laying Out Routines	766
	31.8 Laying Out Classes	768

32	Self-Documenting Code	777
	32.1 External Documentation	777
	32.2 Programming Style as Documentation	778
	32.3 To Comment or Not to Comment	781
	32.4 Keys to Effective Comments	785
	32.5 Commenting Techniques	792
	32.6 IEEE Standards	813
33	Personal Character	819
	33.1 Isn't Personal Character Off the Topic?	820
	33.2 Intelligence and Humility	821
	33.3 Curiosity	822
	33.4 Intellectual Honesty	826
	33.5 Communication and Cooperation	828
	33.6 Creativity and Discipline	829
	33.7 Laziness	830
	33.8 Characteristics That Don't Matter As Much As You Might Think	830
	33.9 Habits	833
34	Themes in Software Craftsmanship	837
	34.1 Conquer Complexity	837
	34.2 Pick Your Process	839
	34.3 Write Programs for People First, Computers Second	841
	34.4 Program into Your Language, Not in It	843
	34.5 Focus Your Attention with the Help of Conventions	844
	34.6 Program in Terms of the Problem Domain	845
	34.7 Watch for Falling Rocks	848
	34.8 Iterate, Repeatedly, Again and Again	850
	34.9 Thou Shalt Rend Software and Religion Asunder	851

35	Where to Find More Information	855
	35.1 Information About Software Construction	856
	35.2 Topics Beyond Construction	857
	35.3 Periodicals	859
	35.4 A Software Developer's Reading Plan	860
	35.5 Joining a Professional Organization	862
	Bibliography.	863
	Index	885

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/

Preface

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

—Fred Brooks

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

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

Who Should Read This Book?

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

Experienced Programmers

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

Technical Leads

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

Self-Taught Programmers

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

Students

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

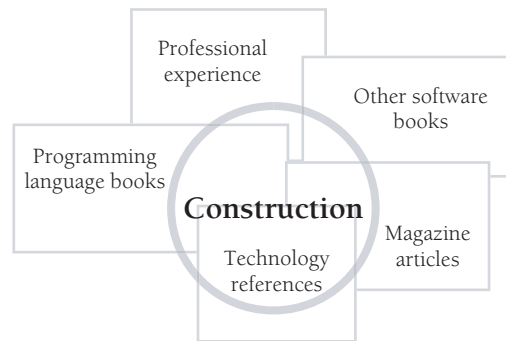
Where Else Can You Find This Information?

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

quently, programmers may have difficulty finding a good source of programming information.

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

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



Key Benefits of This Handbook

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

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

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

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

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

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

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

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

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

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

Access to other sources of information This book collects much of the available information on software construction, but it's hardly the last word. Throughout the

chapters, “Additional Resources” sections describe other books and articles you can read as you pursue the topics you find most interesting.

cc2e.com/1234

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

Why This Handbook Was Written

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

The Topic of Construction Has Been Neglected

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

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

Construction Is Important

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

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

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

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

No Comparable Book Is Available

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

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

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

conceived this book as a full discussion of code construction—from one programmer to another.

Author Note

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

*Bellevue, Washington
Memorial Day, 2004*

Microsoft Learning Technical Support

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

<http://www.microsoft.com/learning/support/>

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

<http://www.microsoft.com/learning/support/search.asp>

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

Postal Mail:

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

E-mail:

mspinput@microsoft.com

Acknowledgments

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

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

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

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

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

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

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

Checklists

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

Configuration Management 669

Integration 707

Programming Tools 724

Layout 773

Self-Documenting Code 780

Good Commenting Technique 816

Tables

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

Table 27-1	Project Size and Typical Error Density	652
Table 27-2	Project Size and Productivity	653
Table 28-1	Factors That Influence Software-Project Effort	674
Table 28-2	Useful Software-Development Measurements	678
Table 28-3	One View of How Programmers Spend Their Time	681

Figures

- Figure 1-1** Construction activities are shown inside the gray circle. Construction focuses on coding and debugging but also includes detailed design, unit testing, integration testing, and other activities. 4
- Figure 1-2** This book focuses on coding and debugging, detailed design, construction planning, unit testing, integration, integration testing, and other activities in roughly these proportions. 5
- Figure 2-1** The letter-writing metaphor suggests that the software process relies on expensive trial and error rather than careful planning and design. 14
- Figure 2-2** It's hard to extend the farming metaphor to software development appropriately. 15
- Figure 2-3** The penalty for a mistake on a simple structure is only a little time and maybe some embarrassment. 17
- Figure 2-4** More complicated structures require more careful planning. 18
- Figure 3-1** The cost to fix a defect rises dramatically as the time from when it's introduced to when it's detected increases. This remains true whether the project is highly sequential (doing 100 percent of requirements and design up front) or highly iterative (doing 5 percent of requirements and design up front). 30
- Figure 3-2** Activities will overlap to some degree on most projects, even those that are highly sequential. 35
- Figure 3-3** On other projects, activities will overlap for the duration of the project. One key to successful construction is understanding the degree to which prerequisites have been completed and adjusting your approach accordingly. 35
- Figure 3-4** The problem definition lays the foundation for the rest of the programming process. 37
- Figure 3-5** Be sure you know what you're aiming at before you shoot. 38
- Figure 3-6** Without good requirements, you can have the right general problem but miss the mark on specific aspects of the problem. 39
- Figure 3-7** Without good software architecture, you may have the right problem but the wrong solution. It may be impossible to have successful construction. 44
- Figure 5-1** The Tacoma Narrows bridge—an example of a wicked problem. 75

- Figure 5-2** The levels of design in a program. The system (1) is first organized into sub-systems (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

Chapter 5

Design in Construction

cc2e.com/0578

Contents

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

Related Topics

- Software architecture: Section 3.5
- Working classes: Chapter 6
- Characteristics of high-quality routines: Chapter 7
- Defensive programming: Chapter 8
- Refactoring: Chapter 24
- How program size affects construction: Chapter 27

Some people might argue that design isn't really a construction activity, but on small projects, many activities are thought of as construction, often including design. On some larger projects, a formal architecture might address only the system-level issues and much design work might intentionally be left for construction. On other large projects, the design might be intended to be detailed enough for coding to be fairly mechanical, but design is rarely that complete—the programmer usually designs part of the program, officially or otherwise.

Cross-Reference For details on the different levels of formality required on large and small projects, see Chapter 27, "How Program Size Affects Construction."

On small, informal projects, a lot of design is done while the programmer sits at the keyboard. "Design" might be just writing a class interface in pseudocode before writing the details. It might be drawing diagrams of a few class relationships before coding them. It might be asking another programmer which design pattern seems like a better choice. Regardless of how it's done, small projects benefit from careful design just as larger projects do, and recognizing design as an explicit activity maximizes the benefit you will receive from it.

Design is a huge topic, so only a few aspects of it are considered in this chapter. A large part of good class or routine design is determined by the system architecture, so be

sure that the architecture prerequisite discussed in Section 3.5 has been satisfied. Even more design work is done at the level of individual classes and routines, described in Chapter 6, “Working Classes,” and Chapter 7, “High-Quality Routines.”

If you’re already familiar with software design topics, you might want to just hit the highlights in the sections about design challenges in Section 5.1 and key heuristics in Section 5.3.

5.1 Design Challenges

Cross-Reference The difference between heuristic and deterministic processes is described in Chapter 2, “Metaphors for a Richer Understanding of Software Development.”

The phrase “software design” means the conception, invention, or contrivance of a scheme for turning a specification for computer software into operational software. Design is the activity that links requirements to coding and debugging. A good top-level design provides a structure that can safely contain multiple lower-level designs. Good design is useful on small projects and indispensable on large projects.

Design is also marked by numerous challenges, which are outlined in this section.

Design Is a Wicked Problem

The picture of the software designer deriving his design in a rational, error-free way from a statement of requirements is quite unrealistic. No system has ever been developed in that way, and probably none ever will. Even the small program developments shown in textbooks and papers are unreal. They have been revised and polished until the author has shown us what he wishes he had done, not what actually did happen.

—David Parnas and
Paul Clements

Horst Rittel and Melvin Webber defined a “wicked” problem as one that could be clearly defined only by solving it, or by solving part of it (1973). This paradox implies, essentially, that you have to “solve” the problem once in order to clearly define it and then solve it again to create a solution that works. This process has been motherhood and apple pie in software development for decades (Peters and Tripp 1976).

In my part of the world, a dramatic example of such a wicked problem was the design of the original Tacoma Narrows bridge. At the time the bridge was built, the main consideration in designing a bridge was that it be strong enough to support its planned load. In the case of the Tacoma Narrows bridge, wind created an unexpected, side-to-side harmonic ripple. One blustery day in 1940, the ripple grew uncontrollably until the bridge collapsed, as shown in Figure 5-1.

This is a good example of a wicked problem because, until the bridge collapsed, its engineers didn’t know that aerodynamics needed to be considered to such an extent. Only by building the bridge (solving the problem) could they learn about the additional consideration in the problem that allowed them to build another bridge that still stands.



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

One of the main differences between programs you develop in school and those you develop as a professional is that the design problems solved by school programs are rarely, if ever, wicked. Programming assignments in school are devised to move you in a beeline from beginning to end. You'd probably want to tar and feather a teacher who gave you a programming assignment, then changed the assignment as soon as you finished the design, and then changed it again just as you were about to turn in the completed program. But that very process is an everyday reality in professional programming.

Design Is a Sloppy Process (Even If it Produces a Tidy Result)

The finished software design should look well organized and clean, but the process used to develop the design isn't nearly as tidy as the end result.

Further Reading For a fuller exploration of this viewpoint, see "A Rational Design Process: How and Why to Fake It" (Parnas and Clements 1986).

Design is sloppy because you take many false steps and go down many blind alleys—you make a lot of mistakes. Indeed, making mistakes is the point of design—it's cheaper to make mistakes and correct designs than it would be to make the same mistakes, recognize them after coding, and have to correct full-blown code. Design is sloppy because a good solution is often only subtly different from a poor one.

Cross-Reference For a better answer to this question, see “How Much Design is Enough?” in Section 5.4 later in this chapter.

Design is also sloppy because it’s hard to know when your design is “good enough.” How much detail is enough? How much design should be done with a formal design notation, and how much should be left to be done at the keyboard? When are you done? Since design is open-ended, the most common answer to that question is “When you’re out of time.”

Design Is About Tradeoffs and Priorities

In an ideal world, every system could run instantly, consume zero storage space, use zero network bandwidth, never contain any errors, and cost nothing to build. In the real world, a key part of the designer’s job is to weigh competing design characteristics and strike a balance among those characteristics. If a fast response rate is more important than minimizing development time, a designer will choose one design. If minimizing development time is more important, a good designer will craft a different design.

Design Involves Restrictions

The point of design is partly to create possibilities and partly to *restrict possibilities*. If people had infinite time, resources, and space to build physical structures, you would see incredible sprawling buildings with one room for each shoe and hundreds of rooms. This is how software can turn out without deliberately imposed restrictions. The constraints of limited resources for constructing buildings force simplifications of the solution that ultimately improve the solution. The goal in software design is the same.

Design Is Nondeterministic

If you send three people away to design the same program, they can easily return with three vastly different designs, each of which could be perfectly acceptable. There might be more than one way to skin a cat, but there are usually dozens of ways to design a computer program.

Design Is a Heuristic Process



Because design is nondeterministic, design techniques tend to be heuristics—“rules of thumb” or “things to try that sometimes work”—rather than repeatable processes that are guaranteed to produce predictable results. Design involves trial and error. A design tool or technique that worked well on one job or on one aspect of a job might not work as well on the next project. No tool is right for everything.

Design Is Emergent

cc2e.com/0539

A tidy way of summarizing these attributes of design is to say that design is “emergent.” Designs don’t spring fully formed directly from someone’s brain. They evolve and improve through design reviews, informal discussions, experience writing the code itself, and experience revising the code.

Further Reading Software isn't the only kind of structure that changes over time. Physical structures evolve, too—see *How Buildings Learn* (Brand 1995).

Virtually all systems undergo some degree of design changes during their initial development, and then they typically change to a greater extent as they're extended into later versions. The degree to which change is beneficial or acceptable depends on the nature of the software being built.

5.2 Key Design Concepts

Good design depends on understanding a handful of key concepts. This section discusses the role of complexity, desirable characteristics of designs, and levels of design.

Software's Primary Technical Imperative: Managing Complexity

Cross-Reference For discussion of the way complexity affects programming issues other than design, see Section 34.1, "Conquer Complexity."

To understand the importance of managing complexity, it's useful to refer to Fred Brooks's landmark paper, "No Silver Bullets: Essence and Accidents of Software Engineering" (1987).

Accidental and Essential Difficulties

Brooks argues that software development is made difficult because of two different classes of problems—the *essential* and the *accidental*. In referring to these two terms, Brooks draws on a philosophical tradition going back to Aristotle. In philosophy, the essential properties are the properties that a thing must have in order to be that thing. A car must have an engine, wheels, and doors to be a car. If it doesn't have any of those essential properties, it isn't really a car.

Accidental properties are the properties a thing just happens to have, properties that don't really bear on whether the thing is what it is. A car could have a V8, a turbocharged 4-cylinder, or some other kind of engine and be a car regardless of that detail. A car could have two doors or four; it could have skinny wheels or mag wheels. All those details are accidental properties. You could also think of accidental properties as *incidental*, *discretionary*, *optional*, and *happenstance*.

Cross-Reference Accidental difficulties are more prominent in early-wave development than in late-wave development. For details, see Section 4.3, "Your Location on the Technology Wave."

Brooks observes that the major accidental difficulties in software were addressed long ago. For example, accidental difficulties related to clumsy language syntaxes were largely eliminated in the evolution from assembly language to third-generation languages and have declined in significance incrementally since then. Accidental difficulties related to noninteractive computers were resolved when time-share operating systems replaced batch-mode systems. Integrated programming environments further eliminated inefficiencies in programming work arising from tools that worked poorly together.

Brooks argues that progress on software's remaining *essential* difficulties is bound to be slower. The reason is that, at its essence, software development consists of working out all the details of a highly intricate, interlocking set of concepts. The essential difficulties arise from the necessity of interfacing with the complex, disorderly real world; accurately and completely identifying the dependencies and exception cases; designing solutions that can't be just approximately correct but that must be exactly correct; and so on. Even if we could invent a programming language that used the same terminology as the real-world problem we're trying to solve, programming would still be difficult because of the challenge in determining precisely how the real world works. As software addresses ever-larger real-world problems, the interactions among the real-world entities become increasingly intricate, and that in turn increases the essential difficulty of the software solutions.

The root of all these essential difficulties is complexity—both accidental and essential.

Importance of Managing Complexity

There are two ways of constructing a software design: one way is to make it so simple that there are *obviously* no deficiencies, and the other is to make it so complicated that there are no *obvious* deficiencies.

—C. A. R. Hoare

When software-project surveys report causes of project failure, they rarely identify technical reasons as the primary causes of project failure. Projects fail most often because of poor requirements, poor planning, or poor management. But when projects do fail for reasons that are primarily technical, the reason is often uncontrolled complexity. The software is allowed to grow so complex that no one really knows what it does. When a project reaches the point at which no one completely understands the impact that code changes in one area will have on other areas, progress grinds to a halt.



KEY POINT

Managing complexity is the most important technical topic in software development. In my view, it's so important that Software's Primary Technical Imperative has to be *managing complexity*.

Complexity is not a new feature of software development. Computing pioneer Edsger Dijkstra pointed out that computing is the only profession in which a single mind is obliged to span the distance from a bit to a few hundred megabytes, a ratio of 1 to 10^9 , or nine orders of magnitude (Dijkstra 1989). This gigantic ratio is staggering. Dijkstra put it this way: "Compared to that number of semantic levels, the average mathematical theory is almost flat. By evoking the need for deep conceptual hierarchies, the automatic computer confronts us with a radically new intellectual challenge that has no precedent in our history." Of course software has become even more complex since 1989, and Dijkstra's ratio of 1 to 10^9 could easily be more like 1 to 10^{15} today.

One symptom that you have bogged down in complexity overload is when you find yourself doggedly applying a method that is clearly irrelevant, at least to any outside observer. It is like the mechanically inept person whose car breaks down—so he puts water in the battery and empties the ashtrays.

—P. J. Plauger

Dijkstra pointed out that no one's skull is really big enough to contain a modern computer program (Dijkstra 1972), which means that we as software developers shouldn't try to cram whole programs into our skulls at once; we should try to organize our programs in such a way that we can safely focus on one part of it at a time. The goal is to minimize the amount of a program you have to think about at any one time. You might think of this as mental juggling—the more mental balls the program requires you to keep in the air at once, the more likely you'll drop one of the balls, leading to a design or coding error.

At the software-architecture level, the complexity of a problem is reduced by dividing the system into subsystems. Humans have an easier time comprehending several simple pieces of information than one complicated piece. The goal of all software-design techniques is to break a complicated problem into simple pieces. The more independent the subsystems are, the more you make it safe to focus on one bit of complexity at a time. Carefully defined objects separate concerns so that you can focus on one thing at a time. Packages provide the same benefit at a higher level of aggregation.

Keeping routines short helps reduce your mental workload. Writing programs in terms of the problem domain, rather than in terms of low-level implementation details, and working at the highest level of abstraction reduce the load on your brain.

The bottom line is that programmers who compensate for inherent human limitations write code that's easier for themselves and others to understand and that has fewer errors.

How to Attack Complexity

Overly costly, ineffective designs arise from three sources:

- A complex solution to a simple problem
- A simple, incorrect solution to a complex problem
- An inappropriate, complex solution to a complex problem

As Dijkstra pointed out, modern software is inherently complex, and no matter how hard you try, you'll eventually bump into some level of complexity that's inherent in the real-world problem itself. This suggests a two-prong approach to managing complexity:



KEY POINT

- Minimize the amount of essential complexity that anyone's brain has to deal with at any one time.
- Keep accidental complexity from needlessly proliferating.

Once you understand that all other technical goals in software are secondary to managing complexity, many design considerations become straightforward.

Desirable Characteristics of a Design

When I am working on a problem I never think about beauty. I think only how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong.

—R. Buckminster Fuller

Cross-Reference These characteristics are related to general software-quality attributes. For details on general attributes, see Section 20.1, “Characteristics of Software Quality.”

A high-quality design has several general characteristics. If you could achieve all these goals, your design would be very good indeed. Some goals contradict other goals, but that’s the challenge of design—creating a good set of tradeoffs from competing objectives. Some characteristics of design quality are also characteristics of a good program: reliability, performance, and so on. Others are internal characteristics of the design.

Here’s a list of internal design characteristics:

Minimal complexity The primary goal of design should be to minimize complexity for all the reasons just described. Avoid making “clever” designs. Clever designs are usually hard to understand. Instead make “simple” and “easy-to-understand” designs. If your design doesn’t let you safely ignore most other parts of the program when you’re immersed in one specific part, the design isn’t doing its job.

Ease of maintenance Ease of maintenance means designing for the maintenance programmer. Continually imagine the questions a maintenance programmer would ask about the code you’re writing. Think of the maintenance programmer as your audience, and then design the system to be self-explanatory.

Loose coupling Loose coupling means designing so that you hold connections among different parts of a program to a minimum. Use the principles of good abstractions in class interfaces, encapsulation, and information hiding to design classes with as few interconnections as possible. Minimal connectedness minimizes work during integration, testing, and maintenance.

Extensibility Extensibility means that you can enhance a system without causing violence to the underlying structure. You can change a piece of a system without affecting other pieces. The most likely changes cause the system the least trauma.

Reusability Reusability means designing the system so that you can reuse pieces of it in other systems.

High fan-in High fan-in refers to having a high number of classes that use a given class. High fan-in implies that a system has been designed to make good use of utility classes at the lower levels in the system.

Low-to-medium fan-out Low-to-medium fan-out means having a given class use a low-to-medium number of other classes. High fan-out (more than about seven) indicates that a class uses a large number of other classes and may therefore be overly complex. Researchers have found that the principle of low fan-out is beneficial whether you're considering the number of routines called from within a routine or the number of classes used within a class (Card and Glass 1990; Basili, Briand, and Melo 1996).

Portability Portability means designing the system so that you can easily move it to another environment.

Leanness Leanness means designing the system so that it has no extra parts (Wirth 1995, McConnell 1997). Voltaire said that a book is finished not when nothing more can be added but when nothing more can be taken away. In software, this is especially true because extra code has to be developed, reviewed, tested, and considered when the other code is modified. Future versions of the software must remain backward-compatible with the extra code. The fatal question is "It's easy, so what will we hurt by putting it in?"

Stratification Stratification means trying to keep the levels of decomposition stratified so that you can view the system at any single level and get a consistent view. Design the system so that you can view it at one level without dipping into other levels.

Cross-Reference For more on working with old systems, see Section 24.5, "Refactoring Strategies."

For example, if you're writing a modern system that has to use a lot of older, poorly designed code, write a layer of the new system that's responsible for interfacing with the old code. Design the layer so that it hides the poor quality of the old code, presenting a consistent set of services to the newer layers. Then have the rest of the system use those classes rather than the old code. The beneficial effects of stratified design in such a case are (1) it compartmentalizes the messiness of the bad code and (2) if you're ever allowed to jettison the old code or refactor it, you won't need to modify any new code except the interface layer.

Cross-Reference An especially valuable kind of standardization is the use of design patterns, which are discussed in "Look for Common Design Patterns" in Section 5.3.

Standard techniques The more a system relies on exotic pieces, the more intimidating it will be for someone trying to understand it the first time. Try to give the whole system a familiar feeling by using standardized, common approaches.

Levels of Design

Design is needed at several different levels of detail in a software system. Some design techniques apply at all levels, and some apply at only one or two. Figure 5-2 illustrates the levels.

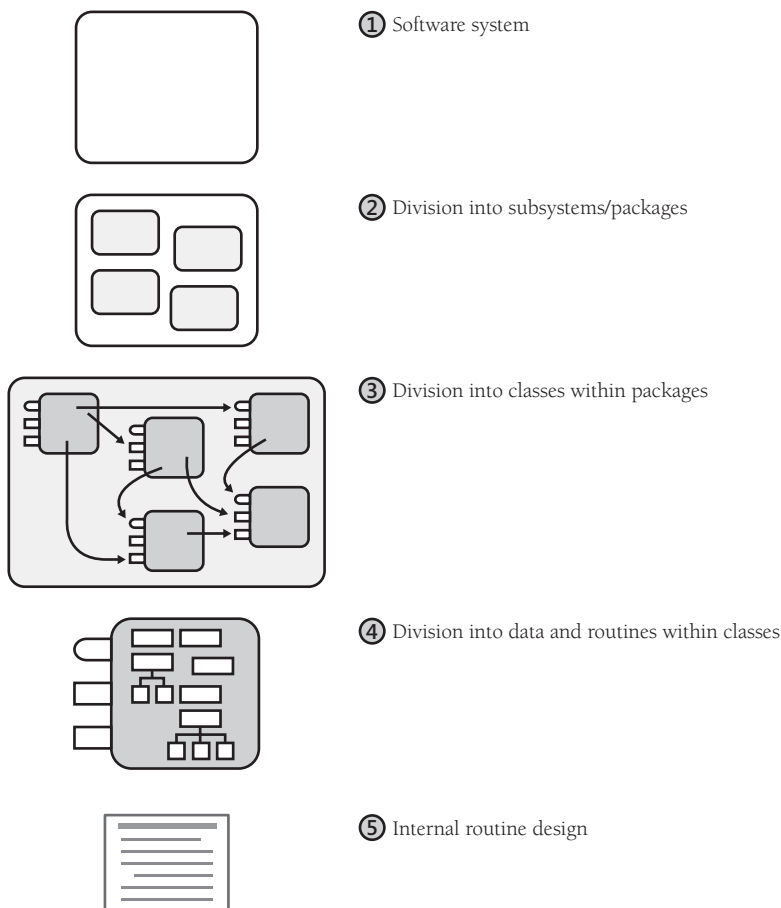


Figure 5-2 The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

Level 1: Software System

In other words—and this is the rock-solid principle on which the whole of the Corporation's Galaxywide success is founded—their fundamental design flaws are completely hidden by their superficial design flaws.
—Douglas Adams

The first level is the entire system. Some programmers jump right from the system level into designing classes, but it's usually beneficial to think through higher level combinations of classes, such as subsystems or packages.

Level 2: Division into Subsystems or Packages

The main product of design at this level is the identification of all major subsystems. The subsystems can be big: database, user interface, business rules, command interpreter,

report engine, and so on. The major design activity at this level is deciding how to partition the program into major subsystems and defining how each subsystem is allowed to use each other subsystem. Division at this level is typically needed on any project that takes longer than a few weeks. Within each subsystem, different methods of design might be used—choosing the approach that best fits each part of the system. In Figure 5-2, design at this level is marked with a 2.

Of particular importance at this level are the rules about how the various subsystems can communicate. If all subsystems can communicate with all other subsystems, you lose the benefit of separating them at all. Make each subsystem meaningful by restricting communications.

Suppose for example that you define a system with six subsystems, as shown in Figure 5-3. When there are no rules, the second law of thermodynamics will come into play and the entropy of the system will increase. One way in which entropy increases is that, without any restrictions on communications among subsystems, communication will occur in an unrestricted way, as in Figure 5-4.

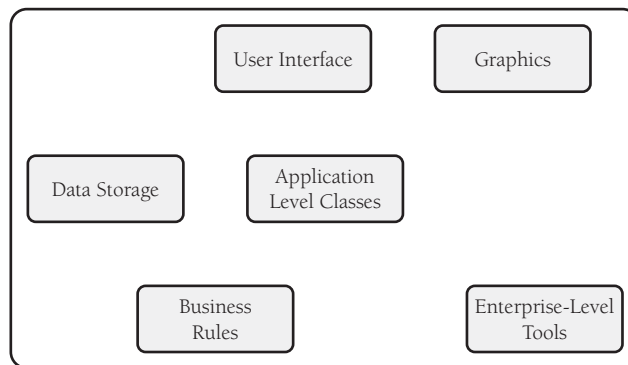


Figure 5-3 An example of a system with six subsystems.

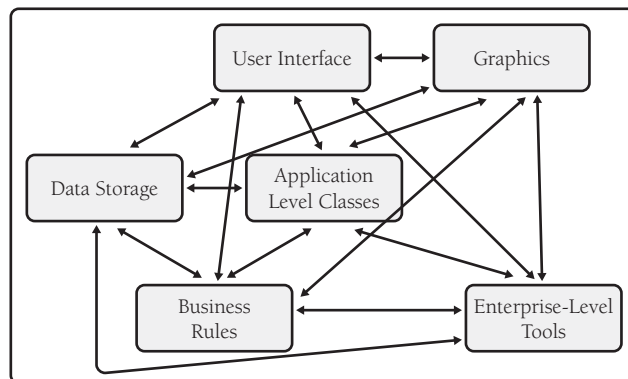


Figure 5-4 An example of what happens with no restrictions on intersubsystem communications.

As you can see, every subsystem ends up communicating directly with every other subsystem, which raises some important questions:

- How many different parts of the system does a developer need to understand at least a little bit to change something in the graphics subsystem?
- What happens when you try to use the business rules in another system?
- What happens when you want to put a new user interface on the system, perhaps a command-line UI for test purposes?
- What happens when you want to put data storage on a remote machine?

You might think of the lines between subsystems as being hoses with water running through them. If you want to reach in and pull out a subsystem, that subsystem is going to have some hoses attached to it. The more hoses you have to disconnect and reconnect, the more wet you're going to get. You want to architect your system so that if you pull out a subsystem to use elsewhere, you won't have many hoses to reconnect and those hoses will reconnect easily.

With forethought, all of these issues can be addressed with little extra work. Allow communication between subsystems only on a “need to know” basis—and it had better be a *good* reason. If in doubt, it's easier to restrict communication early and relax it later than it is to relax it early and then try to tighten it up after you've coded several hundred intersubsystem calls. Figure 5-5 shows how a few communication guidelines could change the system depicted in Figure 5-4.

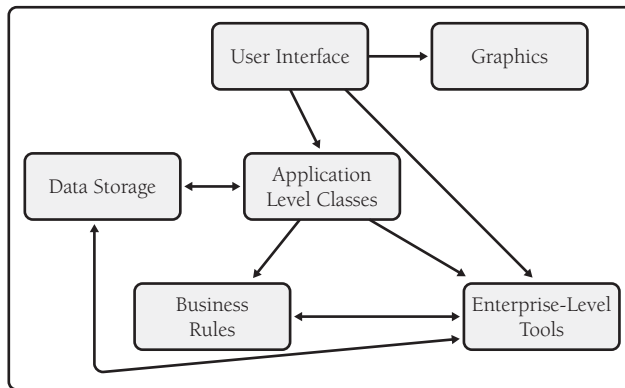


Figure 5-5 With a few communication rules, you can simplify subsystem interactions significantly.

To keep the connections easy to understand and maintain, err on the side of simple intersubsystem relations. The simplest relationship is to have one subsystem call routines in another. A more involved relationship is to have one subsystem contain classes from another. The most involved relationship is to have classes in one subsystem inherit from classes in another.

A good general rule is that a system-level diagram like Figure 5-5 should be an acyclic graph. In other words, a program shouldn't contain any circular relationships in which Class A uses Class B, Class B uses Class C, and Class C uses Class A.

On large programs and families of programs, design at the subsystem level makes a difference. If you believe that your program is small enough to skip subsystem-level design, at least make the decision to skip that level of design a conscious one.

Common Subsystems Some kinds of subsystems appear again and again in different systems. Here are some of the usual suspects.

Cross-Reference For more on simplifying business logic by expressing it in tables, see Chapter 18, "Table-Driven Methods."

Business rules Business rules are the laws, regulations, policies, and procedures that you encode into a computer system. If you're writing a payroll system, you might encode rules from the IRS about the number of allowable withholdings and the estimated tax rate. Additional rules for a payroll system might come from a union contract specifying overtime rates, vacation and holiday pay, and so on. If you're writing a program to quote automobile insurance rates, rules might come from government regulations on required liability coverages, actuarial rate tables, or underwriting restrictions

User interface Create a subsystem to isolate user-interface components so that the user interface can evolve without damaging the rest of the program. In most cases, a user-interface subsystem uses several subordinate subsystems or classes for the GUI interface, command line interface, menu operations, window management, help system, and so forth.

Database access You can hide the implementation details of accessing a database so that most of the program doesn't need to worry about the messy details of manipulating low-level structures and can deal with the data in terms of how it's used at the business-problem level. Subsystems that hide implementation details provide a valuable level of abstraction that reduces a program's complexity. They centralize database operations in one place and reduce the chance of errors in working with the data. They make it easy to change the database design structure without changing most of the program.

System dependencies Package operating-system dependencies into a subsystem for the same reason you package hardware dependencies. If you're developing a program for Microsoft Windows, for example, why limit yourself to the Windows environment? Isolate the Windows calls in a Windows-interface subsystem. If you later want to move your program to Mac OS or Linux, all you'll have to change is the interface subsystem. An interface subsystem can be too extensive for you to implement on your own, but such subsystems are readily available in any of several commercial code libraries.

Level 3: Division into Classes

Further Reading For a good discussion of database design, see *Agile Database Techniques* (Ambler 2003).

Design at this level includes identifying all classes in the system. For example, a database-interface subsystem might be further partitioned into data access classes and persistence framework classes as well as database metadata. Figure 5-2, Level 3, shows how one of Level 2's subsystems might be divided into classes, and it implies that the other three subsystems shown at Level 2 are also decomposed into classes.

Details of the ways in which each class interacts with the rest of the system are also specified as the classes are specified. In particular, the class's interface is defined. Overall, the major design activity at this level is making sure that all the subsystems have been decomposed to a level of detail fine enough that you can implement their parts as individual classes.

Cross-Reference For details on characteristics of high-quality classes, see Chapter 6, "Working Classes."

The division of subsystems into classes is typically needed on any project that takes longer than a few days. If the project is large, the division is clearly distinct from the program partitioning of Level 2. If the project is very small, you might move directly from the whole-system view of Level 1 to the classes view of Level 3.

Classes vs. Objects A key concept in object-oriented design is the differentiation between objects and classes. An object is any specific entity that exists in your program at run time. A class is the static thing you look at in the program listing. An object is the dynamic thing with specific values and attributes you see when you run the program. For example, you could declare a class *Person* that had attributes of name, age, gender, and so on. At run time you would have the objects *nancy*, *hank*, *diane*, *tony*, and so on—that is, specific instances of the class. If you're familiar with database terms, it's the same as the distinction between "schema" and "instance." You could think of the class as the cookie cutter and the object as the cookie. This book uses the terms informally and generally refers to classes and objects more or less interchangeably.

Level 4: Division into Routines

Design at this level includes dividing each class into routines. The class interface defined at Level 3 will define some of the routines. Design at Level 4 will detail the class's private routines. When you examine the details of the routines inside a class, you can see that many routines are simple boxes but a few are composed of hierarchically organized routines, which require still more design.

The act of fully defining the class's routines often results in a better understanding of the class's interface, and that causes corresponding changes to the interface—that is, changes back at Level 3.

This level of decomposition and design is often left up to the individual programmer, and it's needed on any project that takes more than a few hours. It doesn't need to be done formally, but it at least needs to be done mentally.

Level 5: Internal Routine Design

Cross-Reference For details on creating high-quality routines, see Chapter 7, “High-Quality Routines,” and Chapter 8, “Defensive Programming.”

Design at the routine level consists of laying out the detailed functionality of the individual routines. Internal routine design is typically left to the individual programmer working on an individual routine. The design consists of activities such as writing pseudocode, looking up algorithms in reference books, deciding how to organize the paragraphs of code in a routine, and writing programming-language code. This level of design is always done, though sometimes it’s done unconsciously and poorly rather than consciously and well. In Figure 5-2, design at this level is marked with a 5.

5.3 Design Building Blocks: Heuristics

Software developers tend to like our answers cut and dried: “Do A, B, and C, and X, Y, Z will follow every time.” We take pride in learning arcane sets of steps that produce desired effects, and we become annoyed when instructions don’t work as advertised. This desire for deterministic behavior is highly appropriate to detailed computer programming, where that kind of strict attention to detail makes or breaks a program. But software design is a much different story.

Because design is nondeterministic, skillful application of an effective set of heuristics is the core activity in good software design. The following subsections describe a number of heuristics—ways to think about a design that sometime produce good design insights. You might think of heuristics as the guides for the trials in “trial and error.” You undoubtedly have run across some of these before. Consequently, the following subsections describe each of the heuristics in terms of Software’s Primary Technical Imperative: managing complexity.

Find Real-World Objects

Ask not first what the system does; ask WHAT it does it to!
—Bertrand Meyer

The first and most popular approach to identifying design alternatives is the “by the book” object-oriented approach, which focuses on identifying real-world and synthetic objects.

The steps in designing with objects are

Cross-Reference For more details on designing using classes, see Chapter 6, “Working Classes.”

- Identify the objects and their attributes (methods and data).
- Determine what can be done to each object.
- Determine what each object is allowed to do to other objects.
- Determine the parts of each object that will be visible to other objects—which parts will be public and which will be private.
- Define each object’s public interface.

These steps aren't necessarily performed in order, and they're often repeated. Iteration is important. Each of these steps is summarized below.

Identify the objects and their attributes Computer programs are usually based on real-world entities. For example, you could base a time-billing system on real-world employees, clients, timecards, and bills. Figure 5-6 shows an object-oriented view of such a billing system.

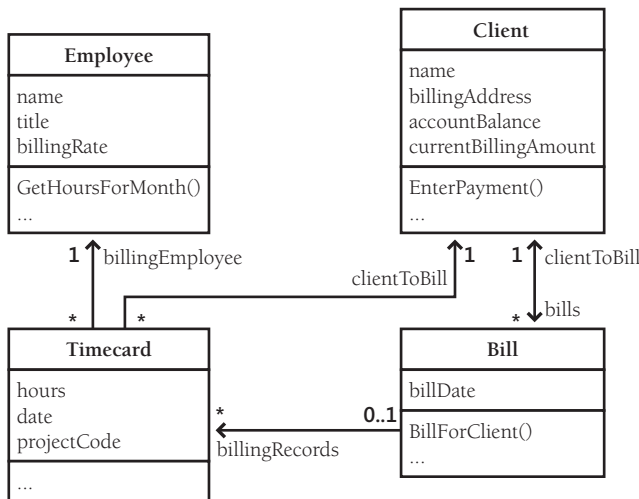


Figure 5-6 This billing system is composed of four major objects. The objects have been simplified for this example.

Identifying the objects' attributes is no more complicated than identifying the objects themselves. Each object has characteristics that are relevant to the computer program. For example, in the time-billing system, an employee object has a name, a title, and a billing rate. A client object has a name, a billing address, and an account balance. A bill object has a billing amount, a client name, a billing date, and so on.

Objects in a graphical user interface system would include windows, dialog boxes, buttons, fonts, and drawing tools. Further examination of the problem domain might produce better choices for software objects than a one-to-one mapping to real-world objects, but the real-world objects are a good place to start.

Determine what can be done to each object A variety of operations can be performed on each object. In the billing system shown in Figure 5-6, an employee object could have a change in title or billing rate, a client object could have its name or billing address changed, and so on.

Determine what each object is allowed to do to other objects This step is just what it sounds like. The two generic things objects can do to each other are containment and inheritance. Which objects can *contain* which other objects? Which objects can *inherit*

from which other objects? In Figure 5-6, a timecard object can contain an employee object and a client object, and a bill can contain one or more timecards. In addition, a bill can indicate that a client has been billed, and a client can enter payments against a bill. A more complicated system would include additional interactions.

Cross-Reference For details on classes and information hiding, see “Hide Secrets (Information Hiding)” in Section 5.3.

Determine the parts of each object that will be visible to other objects One of the key design decisions is identifying the parts of an object that should be made public and those that should be kept private. This decision has to be made for both data and methods.

Define each object’s interfaces Define the formal, syntactic, programming-language-level interfaces to each object. The data and methods the object exposes to every other object is called the object’s “public interface.” The parts of the object that it exposes to derived objects via inheritance is called the object’s “protected interface.” Think about both kinds of interfaces.

When you finish going through the steps to achieve a top-level object-oriented system organization, you’ll iterate in two ways. You’ll iterate on the top-level system organization to get a better organization of classes. You’ll also iterate on each of the classes you’ve defined, driving the design of each class to a more detailed level.

Form Consistent Abstractions

Abstraction is the ability to engage with a concept while safely ignoring some of its details—handling different details at different levels. Any time you work with an aggregate, you’re working with an abstraction. If you refer to an object as a “house” rather than a combination of glass, wood, and nails, you’re making an abstraction. If you refer to a collection of houses as a “town,” you’re making another abstraction.

Base classes are abstractions that allow you to focus on common attributes of a set of derived classes and ignore the details of the specific classes while you’re working on the base class. A good class interface is an abstraction that allows you to focus on the interface without needing to worry about the internal workings of the class. The interface to a well-designed routine provides the same benefit at a lower level of detail, and the interface to a well-designed package or subsystem provides that benefit at a higher level of detail.

From a complexity point of view, the principal benefit of abstraction is that it allows you to ignore irrelevant details. Most real-world objects are already abstractions of some kind. As just mentioned, a house is an abstraction of windows, doors, siding, wiring, plumbing, insulation, and a particular way of organizing them. A door is in turn an abstraction of a particular arrangement of a rectangular piece of material with hinges and a doorknob. And the doorknob is an abstraction of a particular formation of brass, nickel, iron, or steel.

People use abstraction continuously. If you had to deal with individual wood fibers, varnish molecules, and steel molecules every time you used your front door, you'd hardly make it in or out of your house each day. As Figure 5-7 suggests, abstraction is a big part of how we deal with complexity in the real world.



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

Cross-Reference For more details on abstraction in class design, see “Good Abstraction” in Section 6.2.

Software developers sometimes build systems at the wood-fiber, varnish-molecule, and steel-molecule level. This makes the systems overly complex and intellectually hard to manage. When programmers fail to provide larger programming abstractions, the system itself sometimes fails to make it through the front door.

Good programmers create abstractions at the routine-interface level, class-interface level, and package-interface level—in other words, the doorknob level, door level, and house level—and that supports faster and safer programming.

Encapsulate Implementation Details

Encapsulation picks up where abstraction leaves off. Abstraction says, “You’re allowed to look at an object at a high level of detail.” Encapsulation says, “Furthermore, you aren’t allowed to look at an object at any other level of detail.”

Continuing with the housing-materials analogy: encapsulation is a way of saying that you can look at the outside of the house but you can’t get close enough to make out the door’s details. You are allowed to know that there’s a door, and you’re allowed to know whether the door is open or closed, but you’re not allowed to know whether the door is made of wood, fiberglass, steel, or some other material, and you’re certainly not allowed to look at each individual wood fiber.

As Figure 5-8 suggests, encapsulation helps to manage complexity by forbidding you to look at the complexity. The section titled “Good Encapsulation” in Section 6.2 provides more background on encapsulation as it applies to class design.

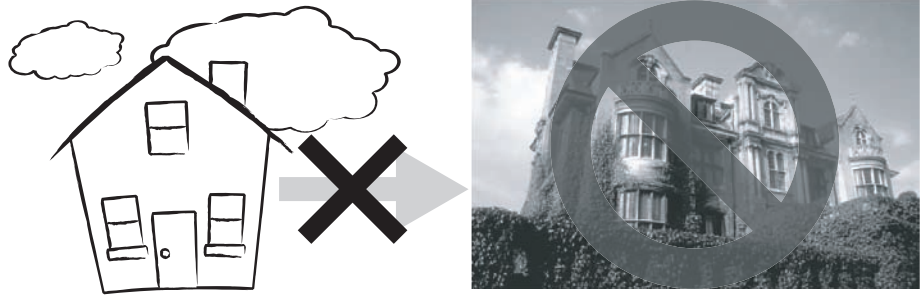


Figure 5-8 Encapsulation says that, not only are you allowed to take a simpler view of a complex concept, you are *not* allowed to look at any of the details of the complex concept. What you see is what you get—it's all you get!

Inherit—When Inheritance Simplifies the Design

In designing a software system, you'll often find objects that are much like other objects, except for a few differences. In an accounting system, for instance, you might have both full-time and part-time employees. Most of the data associated with both kinds of employees is the same, but some is different. In object-oriented programming, you can define a general type of employee and then define full-time employees as general employees, except for a few differences, and part-time employees also as general employees, except for a few differences. When an operation on an employee doesn't depend on the type of employee, the operation is handled as if the employee were just a general employee. When the operation depends on whether the employee is full-time or part-time, the operation is handled differently.

Defining similarities and differences among such objects is called “inheritance” because the specific part-time and full-time employees inherit characteristics from the general-employee type.

The benefit of inheritance is that it works synergistically with the notion of abstraction. Abstraction deals with objects at different levels of detail. Recall the door that was a collection of certain kinds of molecules at one level, a collection of wood fibers at the next, and something that keeps burglars out of your house at the next level. Wood has certain properties—for example, you can cut it with a saw or glue it with wood glue—and two-by-fours or cedar shingles have the general properties of wood as well as some specific properties of their own.

Inheritance simplifies programming because you write a general routine to handle anything that depends on a door's general properties and then write specific routines to handle specific operations on specific kinds of doors. Some operations, such as

`Open()` or `Close()`, might apply regardless of whether the door is a solid door, interior door, exterior door, screen door, French door, or sliding glass door. The ability of a language to support operations like `Open()` or `Close()` without knowing until run time what kind of door you're dealing with is called "polymorphism." Object-oriented languages such as C++, Java, and later versions of Microsoft Visual Basic support inheritance and polymorphism.

Inheritance is one of object-oriented programming's most powerful tools. It can provide great benefits when used well, and it can do great damage when used naively. For details, see "Inheritance ("is a" Relationships)" in Section 6.3.

Hide Secrets (Information Hiding)

Information hiding is part of the foundation of both structured design and object-oriented design. In structured design, the notion of "black boxes" comes from information hiding. In object-oriented design, it gives rise to the concepts of encapsulation and modularity and it is associated with the concept of abstraction. Information hiding is one of the seminal ideas in software development, and so this subsection explores it in depth.

Information hiding first came to public attention in a paper published by David Parnas in 1972 called "On the Criteria to Be Used in Decomposing Systems Into Modules." Information hiding is characterized by the idea of "secrets," design and implementation decisions that a software developer hides in one place from the rest of a program.

In the 20th Anniversary edition of *The Mythical Man Month*, Fred Brooks concluded that his criticism of information hiding was one of the few ways in which the first edition of his book was wrong. "Parnas was right, and I was wrong about information hiding," he proclaimed (Brooks 1995). Barry Boehm reported that information hiding was a powerful technique for eliminating rework, and he pointed out that it was particularly effective in incremental, high-change environments (Boehm 1987).

Information hiding is a particularly powerful heuristic for Software's Primary Technical Imperative because, beginning with its name and throughout its details, it emphasizes *hiding complexity*.

Secrets and the Right to Privacy

In information hiding, each class (or package or routine) is characterized by the design or construction decisions that it hides from all other classes. The secret might be an area that's likely to change, the format of a file, the way a data type is implemented, or an area that needs to be walled off from the rest of the program so that errors in that area cause as little damage as possible. The class's job is to keep this information hidden and to protect its own right to privacy. Minor changes to a system

might affect several routines within a class, but they should not ripple beyond the class interface.

Strive for class interfaces that are complete and minimal.

—Scott Meyers

One key task in designing a class is deciding which features should be known outside the class and which should remain secret. A class might use 25 routines and expose only 5 of them, using the other 20 internally. A class might use several data types and expose no information about them. This aspect of class design is also known as “visibility” since it has to do with which features of the class are “visible” or “exposed” outside the class.

The interface to a class should reveal as little as possible about its inner workings. As shown in Figure 5-9, a class is a lot like an iceberg: seven-eighths is under water, and you can see only the one-eighth that’s above the surface.

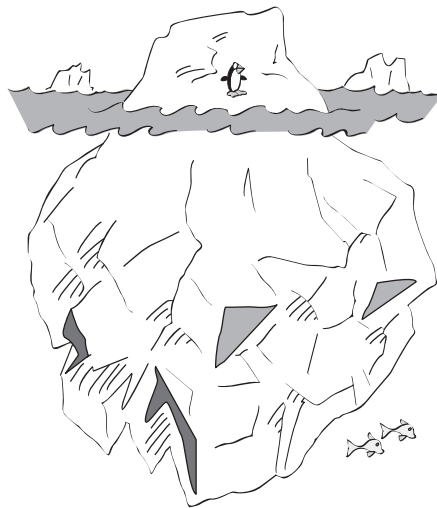


Figure 5-9 A good class interface is like the tip of an iceberg, leaving most of the class unexposed.

Designing the class interface is an iterative process just like any other aspect of design. If you don’t get the interface right the first time, try a few more times until it stabilizes. If it doesn’t stabilize, you need to try a different approach.

An Example of Information Hiding

Suppose you have a program in which each object is supposed to have a unique ID stored in a member variable called *id*. One design approach would be to use integers for the IDs and to store the highest ID assigned so far in a global variable called *g_maxId*. As each new object is allocated, perhaps in each object’s constructor, you could simply use the *id* = ++*g_maxId* statement, which would guarantee a unique *id*, and it would add the absolute minimum of code in each place an object is created. What could go wrong with that?

A lot of things could go wrong. What if you want to reserve ranges of IDs for special purposes? What if you want to use nonsequential IDs to improve security? What if you want to be able to reuse the IDs of objects that have been destroyed? What if you want to add an assertion that fires when you allocate more IDs than the maximum number you've anticipated? If you allocated IDs by spreading `id = ++g_maxId` statements throughout your program, you would have to change code associated with every one of those statements. And, if your program is multithreaded, this approach won't be thread-safe.

The way that new IDs are created is a design decision that you should hide. If you use the phrase `++g_maxId` throughout your program, you expose the way a new ID is created, which is simply by incrementing `g_maxId`. If instead you put the `id = NewId()` statement throughout your program, you hide the information about how new IDs are created. Inside the `NewId()` routine you might still have just one line of code, `return (++g_maxId)` or its equivalent, but if you later decide to reserve certain ranges of IDs for special purposes or to reuse old IDs, you could make those changes within the `NewId()` routine itself—without touching dozens or hundreds of `id = NewId()` statements. No matter how complicated the revisions inside `NewId()` might become, they wouldn't affect any other part of the program.

Now suppose you discover you need to change the type of the ID from an integer to a string. If you've spread variable declarations like `int id` throughout your program, your use of the `NewId()` routine won't help. You'll still have to go through your program and make dozens or hundreds of changes.

An additional secret to hide is the ID's type. By exposing the fact that IDs are integers, you encourage programmers to perform integer operations like `>`, `<`, `=` on them. In C++, you could use a simple *typedef* to declare your IDs to be of *IdType*—a user-defined type that resolves to *int*—rather than directly declaring them to be of type *int*. Alternatively, in C++ and other languages you could create a simple *IdType* class. Once again, hiding a design decision makes a huge difference in the amount of code affected by a change.



Information hiding is useful at all levels of design, from the use of named constants instead of literals, to creation of data types, to class design, routine design, and subsystem design.

Two Categories of Secrets

Secrets in information hiding fall into two general camps:

- Hiding complexity so that your brain doesn't have to deal with it unless you're specifically concerned with it
- Hiding sources of change so that when change occurs, the effects are localized

Sources of complexity include complicated data types, file structures, boolean tests, involved algorithms, and so on. A comprehensive list of sources of change is described later in this chapter.

Barriers to Information Hiding

Further Reading Parts of this section are adapted from “Designing Software for Ease of Extension and Contraction” (Parnas 1979).

In a few instances, information hiding is truly impossible, but most of the barriers to information hiding are mental blocks built up from the habitual use of other techniques.

Excessive distribution of information One common barrier to information hiding is an excessive distribution of information throughout a system. You might have hard-coded the literal *100* throughout a system. Using *100* as a literal decentralizes references to it. It’s better to hide the information in one place, in a constant *MAX_EMPLOYEES* perhaps, whose value is changed in only one place.

Another example of excessive information distribution is interleaving interaction with human users throughout a system. If the mode of interaction changes—say, from a GUI interface to a command line interface—virtually all the code will have to be modified. It’s better to concentrate user interaction in a single class, package, or subsystem you can change without affecting the whole system.

Cross-Reference For more on accessing global data through class interfaces, see “Using Access Routines Instead of Global Data” in Section 13.3.

Yet another example would be a global data element—perhaps an array of employee data with 1000 elements maximum that’s accessed throughout a program. If the program uses the global data directly, information about the data item’s implementation—such as the fact that it’s an array and has a maximum of 1000 elements—will be spread throughout the program. If the program uses the data only through access routines, only the access routines will know the implementation details.

Circular dependencies A more subtle barrier to information hiding is circular dependencies, as when a routine in class *A* calls a routine in class *B*, and a routine in class *B* calls a routine in class *A*.

Avoid such dependency loops. They make it hard to test a system because you can’t test either class *A* or class *B* until at least part of the other is ready.

Class data mistaken for global data If you’re a conscientious programmer, one of the barriers to effective information hiding might be thinking of class data as global data and avoiding it because you want to avoid the problems associated with global data. While the road to programming hell is paved with global variables, class data presents far fewer risks.

Global data is generally subject to two problems: routines operate on global data without knowing that other routines are operating on it, and routines are aware that other routines are operating on the global data but they don’t know exactly what they’re doing to it. Class data isn’t subject to either of these problems. Direct access to the data is restricted to a few routines organized into a single class. The routines are aware that other routines operate on the data, and they know exactly which other routines they are.

Of course, this whole discussion assumes that your system makes use of well-designed, small classes. If your program is designed to use huge classes that contain dozens of routines each, the distinction between class data and global data will begin to blur and class data will be subject to many of the same problems as global data.

Cross-Reference Code-level performance optimizations are discussed in Chapter 25, “Code-Tuning Strategies” and Chapter 26, “Code-Tuning Techniques.”

Perceived performance penalties A final barrier to information hiding can be an attempt to avoid performance penalties at both the architectural and the coding levels. You don’t need to worry at either level. At the architectural level, the worry is unnecessary because architecting a system for information hiding doesn’t conflict with architecting it for performance. If you keep both information hiding and performance in mind, you can achieve both objectives.

The more common worry is at the coding level. The concern is that accessing data items indirectly incurs run-time performance penalties for additional levels of object instantiations, routine calls, and so on. This concern is premature. Until you can measure the system’s performance and pinpoint the bottlenecks, the best way to prepare for code-level performance work is to create a highly modular design. When you detect hot spots later, you can optimize individual classes and routines without affecting the rest of the system.

Value of Information Hiding



Information hiding is one of the few theoretical techniques that has indisputably proven its value in practice, which has been true for a long time (Boehm 1987a). Large programs that use information hiding were found years ago to be easier to modify—by a factor of 4—than programs that don’t (Korson and Vaishnavi 1986). Moreover, information hiding is part of the foundation of both structured design and object-oriented design.

Information hiding has unique heuristic power, a unique ability to inspire effective design solutions. Traditional object-oriented design provides the heuristic power of modeling the world in objects, but object thinking wouldn’t help you avoid declaring the ID as an *int* instead of an *IdType*. The object-oriented designer would ask, “Should an ID be treated as an object?” Depending on the project’s coding standards, a “Yes” answer might mean that the programmer has to write a constructor, destructor, copy operator, and assignment operator; comment it all; and place it under configuration control. Most programmers would decide, “No, it isn’t worth creating a whole class just for an ID. I’ll just use *ints*.”

Note what just happened. A useful design alternative, that of simply hiding the ID’s data type, was not even considered. If, instead, the designer had asked, “What about the ID should be hidden?” he might well have decided to hide its type behind a simple type declaration that substitutes *IdType* for *int*. The difference between object-oriented design and information hiding in this example is more subtle than a clash of explicit rules and regulations. Object-oriented design would approve of this design decision as much as information hiding would. Rather, the difference is one of heuristics—

thinking about information hiding inspires and promotes design decisions that thinking about objects does not.

Information hiding can also be useful in designing a class's public interface. The gap between theory and practice in class design is wide, and among many class designers the decision about what to put into a class's public interface amounts to deciding what interface would be the most convenient to use, which usually results in exposing as much of the class as possible. From what I've seen, some programmers would rather expose all of a class's private data than write 10 extra lines of code to keep the class's secrets intact.

Asking "What does this class need to hide?" cuts to the heart of the interface-design issue. If you can put a function or data into the class's public interface without compromising its secrets, do. Otherwise, don't.

Asking about what needs to be hidden supports good design decisions at all levels. It promotes the use of named constants instead of literals at the construction level. It helps in creating good routine and parameter names inside classes. It guides decisions about class and subsystem decompositions and interconnections at the system level.



KEY POINT

Get into the habit of asking "What should I hide?" You'll be surprised at how many difficult design issues dissolve before your eyes.

Identify Areas Likely to Change

Further Reading The approach described in this section is adapted from "Designing Software for Ease of Extension and Contraction" (Parnas 1979).

A study of great designers found that one attribute they had in common was their ability to anticipate change (Glass 1995). Accommodating changes is one of the most challenging aspects of good program design. The goal is to isolate unstable areas so that the effect of a change will be limited to one routine, class, or package. Here are the steps you should follow in preparing for such perturbations.

1. **Identify items that seem likely to change.** If the requirements have been done well, they include a list of potential changes and the likelihood of each change. In such a case, identifying the likely changes is easy. If the requirements don't cover potential changes, see the discussion that follows of areas that are likely to change on any project.
2. **Separate items that are likely to change.** Compartmentalize each volatile component identified in step 1 into its own class or into a class with other volatile components that are likely to change at the same time.
3. **Isolate items that seem likely to change.** Design the interclass interfaces to be insensitive to the potential changes. Design the interfaces so that changes are limited to the inside of the class and the outside remains unaffected. Any other class using the changed class should be unaware that the change has occurred. The class's interface should protect its secrets.

Here are a few areas that are likely to change:

Cross-Reference One of the most powerful techniques for anticipating change is to use table-driven methods. For details, see Chapter 18, “Table-Driven Methods.”

Business rules Business rules tend to be the source of frequent software changes. Congress changes the tax structure, a union renegotiates its contract, or an insurance company changes its rate tables. If you follow the principle of information hiding, logic based on these rules won’t be strewn throughout your program. The logic will stay hidden in a single dark corner of the system until it needs to be changed.

Hardware dependencies Examples of hardware dependencies include interfaces to screens, printers, keyboards, mice, disk drives, sound facilities, and communications devices. Isolate hardware dependencies in their own subsystem or class. Isolating such dependencies helps when you move the program to a new hardware environment. It also helps initially when you’re developing a program for volatile hardware. You can write software that simulates interaction with specific hardware, have the hardware-interface subsystem use the simulator as long as the hardware is unstable or unavailable, and then unplug the hardware-interface subsystem from the simulator and plug the subsystem into the hardware when it’s ready to use.

Input and output At a slightly higher level of design than raw hardware interfaces, input/output is a volatile area. If your application creates its own data files, the file format will probably change as your application becomes more sophisticated. User-level input and output formats will also change—the positioning of fields on the page, the number of fields on each page, the sequence of fields, and so on. In general, it’s a good idea to examine all external interfaces for possible changes.

Nonstandard language features Most language implementations contain handy, nonstandard extensions. Using the extensions is a double-edged sword because they might not be available in a different environment, whether the different environment is different hardware, a different vendor’s implementation of the language, or a new version of the language from the same vendor.

If you use nonstandard extensions to your programming language, hide those extensions in a class of their own so that you can replace them with your own code when you move to a different environment. Likewise, if you use library routines that aren’t available in all environments, hide the actual library routines behind an interface that works just as well in another environment.

Difficult design and construction areas It’s a good idea to hide difficult design and construction areas because they might be done poorly and you might need to do them again. Compartmentalize them and minimize the impact their bad design or construction might have on the rest of the system.

Status variables Status variables indicate the state of a program and tend to be changed more frequently than most other data. In a typical scenario, you might originally define an error-status variable as a boolean variable and decide later that it

would be better implemented as an enumerated type with the values *ErrorType_None*, *ErrorType_Warning*, and *ErrorType_Fatal*.

You can add at least two levels of flexibility and readability to your use of status variables:

- Don't use a boolean variable as a status variable. Use an enumerated type instead. It's common to add a new state to a status variable, and adding a new type to an enumerated type requires a mere recompilation rather than a major revision of every line of code that checks the variable.
- Use access routines rather than checking the variable directly. By checking the access routine rather than the variable, you allow for the possibility of more sophisticated state detection. For example, if you wanted to check combinations of an error-state variable and a current-function-state variable, it would be easy to do if the test were hidden in a routine and hard to do if it were a complicated test hard-coded throughout the program.

Data-size constraints When you declare an array of size *100*, you're exposing information to the world that the world doesn't need to see. Defend your right to privacy! Information hiding isn't always as complicated as a whole class. Sometimes it's as simple as using a named constant such as *MAX_EMPLOYEES* to hide a *100*.

Anticipating Different Degrees of Change

Cross-Reference This section's approach to anticipating change does not involve designing ahead or coding ahead. For a discussion of those practices, see "A program contains code that seems like it might be needed someday" in Section 24.2.

Further Reading This discussion draws on the approach described in "On the design and development of program families" (Parnas 1976).

When thinking about potential changes to a system, design the system so that the effect or scope of the change is proportional to the chance that the change will occur. If a change is likely, make sure that the system can accommodate it easily. Only extremely unlikely changes should be allowed to have drastic consequences for more than one class in a system. Good designers also factor in the cost of anticipating change. If a change is not terribly likely but easy to plan for, you should think harder about anticipating it than if it isn't very likely and is difficult to plan for.

A good technique for identifying areas likely to change is first to identify the minimal subset of the program that might be of use to the user. The subset makes up the core of the system and is unlikely to change. Next, define minimal increments to the system. They can be so small that they seem trivial. As you consider functional changes, be sure also to consider qualitative changes: making the program thread-safe, making it localizable, and so on. These areas of potential improvement constitute potential changes to the system; design these areas using the principles of information hiding. By identifying the core first, you can see which components are really add-ons and then extrapolate and hide improvements from there.

Keep Coupling Loose

Coupling describes how tightly a class or routine is related to other classes or routines. The goal is to create classes and routines with small, direct, visible, and flexible relations to other classes and routines, which is known as “loose coupling.” The concept of coupling applies equally to classes and routines, so for the rest of this discussion I’ll use the word “module” to refer to both classes and routines.

Good coupling between modules is loose enough that one module can easily be used by other modules. Model railroad cars are coupled by opposing hooks that latch when pushed together. Connecting two cars is easy—you just push the cars together. Imagine how much more difficult it would be if you had to screw things together, or connect a set of wires, or if you could connect only certain kinds of cars to certain other kinds of cars. The coupling of model railroad cars works because it’s as simple as possible. In software, make the connections among modules as simple as possible.

Try to create modules that depend little on other modules. Make them detached, as business associates are, rather than attached, as Siamese twins are. A routine like *sin()* is loosely coupled because everything it needs to know is passed in to it with one value representing an angle in degrees. A routine such as *InitVars(var 1, var2, var3, ..., varN)* is more tightly coupled because, with all the variables it must pass, the calling module practically knows what is happening inside *InitVars()*. Two classes that depend on each other’s use of the same global data are even more tightly coupled.

Coupling Criteria

Here are several criteria to use in evaluating coupling between modules:

Size Size refers to the number of connections between modules. With coupling, small is beautiful because it’s less work to connect other modules to a module that has a smaller interface. A routine that takes one parameter is more loosely coupled to modules that call it than a routine that takes six parameters. A class with four well-defined public methods is more loosely coupled to modules that use it than a class that exposes 37 public methods.

Visibility Visibility refers to the prominence of the connection between two modules. Programming is not like being in the CIA; you don’t get credit for being sneaky. It’s more like advertising; you get lots of credit for making your connections as blatant as possible. Passing data in a parameter list is making an obvious connection and is therefore good. Modifying global data so that another module can use that data is a sneaky connection and is therefore bad. Documenting the global-data connection makes it more obvious and is slightly better.

Flexibility Flexibility refers to how easily you can change the connections between modules. Ideally, you want something more like the USB connector on your computer than like bare wire and a soldering gun. Flexibility is partly a product of the other

coupling characteristics, but it's a little different too. Suppose you have a routine that looks up the amount of vacation an employee receives each year, given a hiring date and a job classification. Name the routine *LookupVacationBenefit()*. Suppose in another module you have an *employee* object that contains the hiring date and the job classification, among other things, and that module passes the object to *LookupVacationBenefit()*.

From the point of view of the other criteria, the two modules would look loosely coupled. The *employee* connection between the two modules is visible, and there's only one connection. Now suppose that you need to use the *LookupVacationBenefit()* module from a third module that doesn't have an *employee* object but that does have a hiring date and a job classification. Suddenly *LookupVacationBenefit()* looks less friendly, unwilling to associate with the new module.

For the third module to use *LookupVacationBenefit()*, it has to know about the *Employee* class. It could dummy up an *employee* object with only two fields, but that would require internal knowledge of *LookupVacationBenefit()*, namely that those are the only fields it uses. Such a solution would be a kludge, and an ugly one. The second option would be to modify *LookupVacationBenefit()* so that it would take hiring date and job classification instead of *employee*. In either case, the original module turns out to be a lot less flexible than it seemed to be at first.

The happy ending to the story is that an unfriendly module can make friends if it's willing to be flexible—in this case, by changing to take hiring date and job classification specifically instead of *employee*.

In short, the more easily other modules can call a module, the more loosely coupled it is, and that's good because it's more flexible and maintainable. In creating a system structure, break up the program along the lines of minimal interconnectedness. If a program were a piece of wood, you would try to split it with the grain.

Kinds of Coupling

Here are the most common kinds of coupling you'll encounter.

Simple-data-parameter coupling Two modules are simple-data-parameter coupled if all the data passed between them are of primitive data types and all the data is passed through parameter lists. This kind of coupling is normal and acceptable.

Simple-object coupling A module is simple-object coupled to an object if it instantiates that object. This kind of coupling is fine.

Object-parameter coupling Two modules are object-parameter coupled to each other if *Object1* requires *Object2* to pass it an *Object3*. This kind of coupling is tighter than *Object1* requiring *Object2* to pass it only primitive data types because it requires *Object2* to know about *Object3*.

Semantic coupling The most insidious kind of coupling occurs when one module makes use not of some syntactic element of another module but of some semantic knowledge of another module's inner workings. Here are some examples:

- *Module1* passes a control flag to *Module2* that tells *Module2* what to do. This approach requires *Module1* to make assumptions about the internal workings of *Module2*, namely what *Module2* is going to do with the control flag. If *Module2* defines a specific data type for the control flag (enumerated type or object), this usage is probably OK.
- *Module2* uses global data after the global data has been modified by *Module1*. This approach requires *Module2* to assume that *Module1* has modified the data in the ways *Module2* needs it to be modified, and that *Module1* has been called at the right time.
- *Module1*'s interface states that its *Module1.Initialize()* routine should be called before its *Module1.Routine()* is called. *Module2* knows that *Module1.Routine()* calls *Module1.Initialize()* anyway, so it just instantiates *Module1* and calls *Module1.Routine()* without calling *Module1.Initialize()* first.
- *Module1* passes *Object* to *Module2*. Because *Module1* knows that *Module2* uses only three of *Object*'s seven methods, it initializes *Object* only partially—with the specific data those three methods need.
- *Module1* passes *BaseObject* to *Module2*. Because *Module2* knows that *Module1* is really passing it *DerivedObject*, it casts *BaseObject* to *DerivedObject* and calls methods that are specific to *DerivedObject*.

Semantic coupling is dangerous because changing code in the used module can break code in the using module in ways that are completely undetectable by the compiler. When code like this breaks, it breaks in subtle ways that seem unrelated to the change made in the used module, which turns debugging into a Sisyphean task.

The point of loose coupling is that an effective module provides an additional level of abstraction—once you write it, you can take it for granted. It reduces overall program complexity and allows you to focus on one thing at a time. If using a module requires you to focus on more than one thing at once—knowledge of its internal workings, modification to global data, uncertain functionality—the abstractive power is lost and the module's ability to help manage complexity is reduced or eliminated.

Classes and routines are first and foremost intellectual tools for reducing complexity. If they're not making your job simpler, they're not doing their jobs.



Look for Common Design Patterns

cc2e.com/0585

Design patterns provide the cores of ready-made solutions that can be used to solve many of software's most common problems. Some software problems require solutions that are derived from first principles. But most problems are similar to past problems, and those can be solved using similar solutions, or patterns. Common patterns include Adapter, Bridge, Decorator, Facade, Factory Method, Observer, Singleton, Strategy, and Template Method. The book *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1995) is the definitive description of design patterns.

Patterns provide several benefits that fully custom design doesn't:

Patterns reduce complexity by providing ready-made abstractions If you say, "This code uses a Factory Method to create instances of derived classes," other programmers on your project will understand that your code involves a fairly rich set of inter-relationships and programming protocols, all of which are invoked when you refer to the design pattern of Factory Method.

The Factory Method is a pattern that allows you to instantiate any class derived from a specific base class without needing to keep track of the individual derived classes anywhere but the Factory Method. For a good discussion of the Factory Method pattern, see "Replace Constructor with Factory Method" in *Refactoring* (Fowler 1999).

You don't have to spell out every line of code for other programmers to understand the design approach found in your code.

Patterns reduce errors by institutionalizing details of common solutions Software design problems contain nuances that emerge fully only after the problem has been solved once or twice (or three times, or four times, or...). Because patterns represent standardized ways of solving common problems, they embody the wisdom accumulated from years of attempting to solve those problems, and they also embody the corrections to the false attempts that people have made in solving those problems.

Using a design pattern is thus conceptually similar to using library code instead of writing your own. Sure, everybody has written a custom Quicksort a few times, but what are the odds that your custom version will be fully correct on the first try? Similarly, numerous design problems are similar enough to past problems that you're better off using a prebuilt design solution than creating a novel solution.

Patterns provide heuristic value by suggesting design alternatives A designer who's familiar with common patterns can easily run through a list of patterns and ask "Which of these patterns fits my design problem?" Cycling through a set of familiar alternatives is immeasurably easier than creating a custom design solution out of whole cloth. And the code arising from a familiar pattern will also be easier for readers of the code to understand than fully custom code would be.

Patterns streamline communication by moving the design dialog to a higher level In addition to their complexity-management benefit, design patterns can accelerate design discussions by allowing designers to think and discuss at a larger level of granularity. If you say “I can’t decide whether I should use a Creator or a Factory Method in this situation,” you’ve communicated a great deal with just a few words—as long as you and your listener are both familiar with those patterns. Imagine how much longer it would take you to dive into the details of the code for a Creator pattern and the code for a Factory Method pattern and then compare and contrast the two approaches.

If you’re not already familiar with design patterns, Table 5-1 summarizes some of the most common patterns to stimulate your interest.

Table 5-1 Popular Design Patterns

Pattern	Description
Abstract Factory	Supports creation of sets of related objects by specifying the kind of set but not the kinds of each specific object.
Adapter	Converts the interface of a class to a different interface.
Bridge	Builds an interface and an implementation in such a way that either can vary without the other varying.
Composite	Consists of an object that contains additional objects of its own type so that client code can interact with the top-level object and not concern itself with all the detailed objects.
Decorator	Attaches responsibilities to an object dynamically, without creating specific subclasses for each possible configuration of responsibilities.
Facade	Provides a consistent interface to code that wouldn’t otherwise offer a consistent interface.
Factory Method	Instantiates classes derived from a specific base class without needing to keep track of the individual derived classes anywhere but the Factory Method.
Iterator	A server object that provides access to each element in a set sequentially.
Observer	Keeps multiple objects in synch with one another by making an object responsible for notifying the set of related objects about changes to any member of the set.
Singleton	Provides global access to a class that has one and only one instance.
Strategy	Defines a set of algorithms or behaviors that are dynamically interchangeable with each other.
Template Method	Defines the structure of an algorithm but leaves some of the detailed implementation to subclasses.

If you haven’t seen design patterns before, your reaction to the descriptions in Table 5-1 might be “Sure, I already know most of these ideas.” That reaction is a big part of why design patterns are valuable. Patterns are familiar to most experienced programmers, and assigning recognizable names to them supports efficient and effective communication about them.

One potential trap with patterns is force-fitting code to use a pattern. In some cases, shifting code slightly to conform to a well-recognized pattern will improve understandability of the code. But if the code has to be shifted too far, forcing it to look like a standard pattern can sometimes increase complexity.

Another potential trap with patterns is feature-itis: using a pattern because of a desire to try out a pattern rather than because the pattern is an appropriate design solution.

Overall, design patterns are a powerful tool for managing complexity. You can read more detailed descriptions in any of the good books that are listed at the end of this chapter.

Other Heuristics

The preceding sections describe the major software design heuristics. Following are a few other heuristics that might not be useful quite as often but are still worth mentioning.

Aim for Strong Cohesion

Cohesion arose from structured design and is usually discussed in the same context as coupling. Cohesion refers to how closely all the routines in a class or all the code in a routine support a central purpose—how focused the class is. Classes that contain strongly related functionality are described as having strong cohesion, and the heuristic goal is to make cohesion as strong as possible. Cohesion is a useful tool for managing complexity because the more that code in a class supports a central purpose, the more easily your brain can remember everything the code does.

Thinking about cohesion at the routine level has been a useful heuristic for decades and is still useful today. At the class level, the heuristic of cohesion has largely been subsumed by the broader heuristic of well-defined abstractions, which was discussed earlier in this chapter and in Chapter 6. Abstractions are useful at the routine level, too, but on a more even footing with cohesion at that level of detail.

Build Hierarchies

A hierarchy is a tiered information structure in which the most general or abstract representation of concepts is contained at the top of the hierarchy, with increasingly detailed, specialized representations at the hierarchy's lower levels. In software, hierarchies are found in class hierarchies, and, as Level 4 in Figure 5-2 illustrated, in routine-calling hierarchies as well.

Hierarchies have been an important tool for managing complex sets of information for at least 2000 years. Aristotle used a hierarchy to organize the animal kingdom. Humans frequently use outlines to organize complex information (like this book). Researchers have found that people generally find hierarchies to be a natural way to organize complex information. When they draw a complex object such as a house, they draw it hierarchically. First they draw the outline of the house, then the windows

and doors, and then more details. They don't draw the house brick by brick, shingle by shingle, or nail by nail (Simon 1996).

Hierarchies are a useful tool for achieving Software's Primary Technical Imperative because they allow you to focus on only the level of detail you're currently concerned with. The details don't go away completely; they're simply pushed to another level so that you can think about them when you want to rather than thinking about all the details all of the time.

Formalize Class Contracts

Cross-Reference For more on contracts, see "Use assertions to document and verify preconditions and postconditions" in Section 8.2.

At a more detailed level, thinking of each class's interface as a contract with the rest of the program can yield good insights. Typically, the contract is something like "If you promise to provide data x, y, and z and you promise they'll have characteristics a, b, and c, I promise to perform operations 1, 2, and 3 within constraints 8, 9, and 10." The promises the clients of the class make to the class are typically called "preconditions," and the promises the object makes to its clients are called the "postconditions."

Contracts are useful for managing complexity because, at least in theory, the object can safely ignore any noncontractual behavior. In practice, this issue is much more difficult.

Assign Responsibilities

Another heuristic is to think through how responsibilities should be assigned to objects. Asking what each object should be responsible for is similar to asking what information it should hide, but I think it can produce broader answers, which gives the heuristic unique value.

Design for Test

A thought process that can yield interesting design insights is to ask what the system will look like if you design it to facilitate testing. Do you need to separate the user interface from the rest of the code so that you can exercise it independently? Do you need to organize each subsystem so that it minimizes dependencies on other subsystems? Designing for test tends to result in more formalized class interfaces, which is generally beneficial.

Avoid Failure

Civil engineering professor Henry Petroski wrote an interesting book, *Design Paradigms: Case Histories of Error and Judgment in Engineering* (Petroski 1994), that chronicles the history of failures in bridge design. Petroski argues that many spectacular bridge failures have occurred because of focusing on previous successes and not adequately considering possible failure modes. He concludes that failures like the Tacoma Narrows bridge could have been avoided if the designers had carefully considered the ways the bridge might fail and not just copied the attributes of other successful designs.

The high-profile security lapses of various well-known systems the past few years make it hard to disagree that we should find ways to apply Petroski's design-failure insights to software.

Choose Binding Time Consciously

Cross-Reference For more on binding time, see Section 10.6, "Binding Time."

Binding time refers to the time a specific value is bound to a variable. Code that binds early tends to be simpler, but it also tends to be less flexible. Sometimes you can get a good design insight from asking questions like these: What if I bound these values earlier? What if I bound these values later? What if I initialized this table right here in the code? What if I read the value of this variable from the user at run time?

Make Central Points of Control

P.J. Plauger says his major concern is "The Principle of One Right Place—there should be One Right Place to look for any nontrivial piece of code, and One Right Place to make a likely maintenance change" (Plauger 1993). Control can be centralized in classes, routines, preprocessor macros, *#include* files—even a named constant is an example of a central point of control.

The reduced-complexity benefit is that the fewer places you have to look for something, the easier and safer it will be to change.

Consider Using Brute Force

When in doubt, use brute force.
—Butler Lampson

One powerful heuristic tool is brute force. Don't underestimate it. A brute-force solution that works is better than an elegant solution that doesn't work. It can take a long time to get an elegant solution to work. In describing the history of searching algorithms, for example, Donald Knuth pointed out that even though the first description of a binary search algorithm was published in 1946, it took another 16 years for someone to publish an algorithm that correctly searched lists of all sizes (Knuth 1998). A binary search is more elegant, but a brute-force, sequential search is often sufficient.

Draw a Diagram

Diagrams are another powerful heuristic tool. A picture is worth 1000 words—kind of. You actually want to leave out most of the 1000 words because one point of using a picture is that a picture can represent the problem at a higher level of abstraction. Sometimes you want to deal with the problem in detail, but other times you want to be able to work with more generality.

Keep Your Design Modular

Modularity's goal is to make each routine or class like a "black box": You know what goes in, and you know what comes out, but you don't know what happens inside. A

black box has such a simple interface and such well-defined functionality that for any specific input you can accurately predict the corresponding output.

The concept of modularity is related to information hiding, encapsulation, and other design heuristics. But sometimes thinking about how to assemble a system from a set of black boxes provides insights that information hiding and encapsulation don't, so the concept is worth having in your back pocket.

Summary of Design Heuristics

More alarming, the same programmer is quite capable of doing the same task himself in two or three ways, sometimes unconsciously, but quite often simply for a change, or to provide elegant variation.
—A. R. Brown and W. A. Sampson

Here's a summary of major design heuristics:

- Find Real-World Objects
- Form Consistent Abstractions
- Encapsulate Implementation Details
- Inherit When Possible
- Hide Secrets (Information Hiding)
- Identify Areas Likely to Change
- Keep Coupling Loose
- Look for Common Design Patterns

The following heuristics are sometimes useful too:

- Aim for Strong Cohesion
- Build Hierarchies
- Formalize Class Contracts
- Assign Responsibilities
- Design for Test
- Avoid Failure
- Choose Binding Time Consciously
- Make Central Points of Control
- Consider Using Brute Force
- Draw a Diagram
- Keep Your Design Modular

Guidelines for Using Heuristics

Approaches to design in software can learn from approaches to design in other fields. One of the original books on heuristics in problem solving was G. Polya's *How to Solve It* (1957). Polya's generalized problem-solving approach focuses on problem solving in mathematics. Figure 5-10 is a summary of his approach, adapted from a similar summary in his book (emphases his).

cc2e.com/0592

1. Understanding the Problem. You have to *understand* the problem.

What is the unknown? What are the data? What is the condition? Is it possible to satisfy the condition? Is the condition sufficient to determine the unknown? Or is it insufficient? Or redundant? Or contradictory?

Draw a figure. Introduce suitable notation. Separate the various parts of the condition. Can you write them down?

2. Devising a Plan. Find the connection between the data and the unknown. You might be obliged to consider auxiliary problems if you can't find an intermediate connection. You should eventually come up with a *plan* of the solution.

Have you seen the problem before? Or have you seen the same problem in a slightly different form? *Do you know a related problem?* Do you know a theorem that could be useful?

Look at the unknown! And try to think of a familiar problem having the same or a similar unknown. *Here is a problem related to yours and solved before. Can you use it?* Can you use its result? Can you use its method? Should you introduce some auxiliary element in order to make its use possible?

Can you restate the problem? Can you restate it still differently? Go back to definitions.

If you cannot solve the proposed problem, try to solve some related problem first. Can you imagine a more accessible related problem? A more general problem? A more special problem? An analogous problem? Can you solve a part of the problem? Keep only a part of the condition, drop the other part; how far is the unknown then determined, how can it vary? Can you derive something useful from the data? Can you think of other data appropriate for determining the unknown? Can you change the unknown or the data, or both if necessary, so that the new unknown and the new data are nearer to each other?

Did you use all the data? Did you use the whole condition? Have you taken into account all essential notions involved in the problem?

3. Carrying out the Plan. Carry out your plan.

Carrying out your plan of the solution, *check each step*. Can you see clearly that the step is correct? Can you prove that it's correct?

4. Looking Back. Examine the solution.

Can you *check the result*? Can you check the argument? Can you derive the result differently? Can you see it at a glance?

Can you use the result, or the method, for some other problem?

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).

One of the most effective guidelines is not to get stuck on a single approach. If diagramming the design in UML isn't working, write it in English. Write a short test program. Try a completely different approach. Think of a brute-force solution. Keep outlining and sketching with your pencil, and your brain will follow. If all else fails, walk away from the problem. Literally go for a walk, or think about something else before returning to the problem. If you've given it your best and are getting nowhere, putting it out of your mind for a time often produces results more quickly than sheer persistence can.

You don't have to solve the whole design problem at once. If you get stuck, remember that a point needs to be decided but recognize that you don't yet have enough information to resolve that specific issue. Why fight your way through the last 20 percent of the design when it will drop into place easily the next time through? Why make bad decisions based on limited experience with the design when you can make good decisions based on more experience with it later? Some people are uncomfortable if they don't come to closure after a design cycle, but after you have created a few designs without resolving issues prematurely, it will seem natural to leave issues unresolved until you have more information (Zahniser 1992, Beck 2000).

5.4 Design Practices

The preceding section focused on heuristics related to design attributes—what you want the completed design to look like. This section describes *design practice* heuristics, steps you can take that often produce good results.

Iterate

You might have had an experience in which you learned so much from writing a program that you wished you could write it again, armed with the insights you gained from writing it the first time. The same phenomenon applies to design, but the design cycles are shorter and the effects downstream are bigger, so you can afford to whirl through the design loop a few times.



Design is an iterative process. You don't usually go from point A only to point B; you go from point A to point B and back to point A.

As you cycle through candidate designs and try different approaches, you'll look at both high-level and low-level views. The big picture you get from working with high-level issues will help you to put the low-level details in perspective. The details you get from working with low-level issues will provide a foundation in solid reality for the high-level decisions. The tug and pull between top-level and bottom-level

considerations is a healthy dynamic; it creates a stressed structure that's more stable than one built wholly from the top down or the bottom up.

Many programmers—many people, for that matter—have trouble ranging between high-level and low-level considerations. Switching from one view of a system to another is mentally strenuous, but it's essential to creating effective designs. For entertaining exercises to enhance your mental flexibility, read *Conceptual Blockbusting* (Adams 2001), described in the “Additional Resources” section at the end of the chapter.

Cross-Reference Refactoring is a safe way to try different alternatives in code. For more on this, see Chapter 24, “Refactoring.”

When you come up with a first design attempt that seems good enough, don't stop! The second attempt is nearly always better than the first, and you learn things on each attempt that can improve your overall design. After trying a thousand different materials for a light bulb filament with no success, Thomas Edison was reportedly asked if he felt his time had been wasted since he had discovered nothing. “Nonsense,” Edison is supposed to have replied. “I have discovered a thousand things that don't work.” In many cases, solving the problem with one approach will produce insights that will enable you to solve the problem using another approach that's even better.

Divide and Conquer

As Edsger Dijkstra pointed out, no one's skull is big enough to contain all the details of a complex program, and that applies just as well to design. Divide the program into different areas of concern, and then tackle each of those areas individually. If you run into a dead end in one of the areas, iterate!

Incremental refinement is a powerful tool for managing complexity. As Polya recommended in mathematical problem solving, understand the problem, devise a plan, carry out the plan, and then *look back* to see how you did (Polya 1957).

Top-Down and Bottom-Up Design Approaches

“Top down” and “bottom up” might have an old-fashioned sound, but they provide valuable insight into the creation of object-oriented designs. Top-down design begins at a high level of abstraction. You define base classes or other nonspecific design elements. As you develop the design, you increase the level of detail, identifying derived classes, collaborating classes, and other detailed design elements.

Bottom-up design starts with specifics and works toward generalities. It typically begins by identifying concrete objects and then generalizes aggregations of objects and base classes from those specifics.

Some people argue vehemently that starting with generalities and working toward specifics is best, and some argue that you can't really identify general design principles until you've worked out the significant details. Here are the arguments on both sides.

Argument for Top Down

The guiding principle behind the top-down approach is the idea that the human brain can concentrate on only a certain amount of detail at a time. If you start with general classes and decompose them into more specialized classes step by step, your brain isn't forced to deal with too many details at once.

The divide-and-conquer process is iterative in a couple of senses. First, it's iterative because you usually don't stop after one level of decomposition. You keep going for several levels. Second, it's iterative because you don't usually settle for your first attempt. You decompose a program one way. At various points in the decomposition, you'll have choices about which way to partition the subsystems, lay out the inheritance tree, and form compositions of objects. You make a choice and see what happens. Then you start over and decompose it another way and see whether that works better. After several attempts, you'll have a good idea of what will work and why.

How far do you decompose a program? Continue decomposing until it seems as if it would be easier to code the next level than to decompose it. Work until you become somewhat impatient at how obvious and easy the design seems. At that point, you're done. If it's not clear, work some more. If the solution is even slightly tricky for you now, it'll be a bear for anyone who works on it later.

Argument for Bottom Up

Sometimes the top-down approach is so abstract that it's hard to get started. If you need to work with something more tangible, try the bottom-up design approach. Ask yourself, "What do I know this system needs to do?" Undoubtedly, you can answer that question. You might identify a few low-level responsibilities that you can assign to concrete classes. For example, you might know that a system needs to format a particular report, compute data for that report, center its headings, display the report on the screen, print the report on a printer, and so on. After you identify several low-level responsibilities, you'll usually start to feel comfortable enough to look at the top again.

In some other cases, major attributes of the design problem are dictated from the bottom. You might have to interface with hardware devices whose interface requirements dictate large chunks of your design.

Here are some things to keep in mind as you do bottom-up composition:

- Ask yourself what you know the system needs to do.
- Identify concrete objects and responsibilities from that question.
- Identify common objects, and group them using subsystem organization, packages, composition within objects, or inheritance, whichever is appropriate.
- Continue with the next level up, or go back to the top and try again to work down.

No Argument, Really

The key difference between top-down and bottom-up strategies is that one is a decomposition strategy and the other is a composition strategy. One starts from the general problem and breaks it into manageable pieces; the other starts with manageable pieces and builds up a general solution. Both approaches have strengths and weaknesses that you'll want to consider as you apply them to your design problems.

The strength of top-down design is that it's easy. People are good at breaking something big into smaller components, and programmers are especially good at it.

Another strength of top-down design is that you can defer construction details. Since systems are often perturbed by changes in construction details (for example, changes in a file structure or a report format), it's useful to know early on that those details should be hidden in classes at the bottom of the hierarchy.

One strength of the bottom-up approach is that it typically results in early identification of needed utility functionality, which results in a compact, well-factored design. If similar systems have already been built, the bottom-up approach allows you to start the design of the new system by looking at pieces of the old system and asking "What can I reuse?"

A weakness of the bottom-up composition approach is that it's hard to use exclusively. Most people are better at taking one big concept and breaking it into smaller concepts than they are at taking small concepts and making one big one. It's like the old assemble-it-yourself problem: I thought I was done, so why does the box still have parts in it? Fortunately, you don't have to use the bottom-up composition approach exclusively.

Another weakness of the bottom-up design strategy is that sometimes you find that you can't build a program from the pieces you've started with. You can't build an airplane from bricks, and you might have to work at the top before you know what kinds of pieces you need at the bottom.

To summarize, top down tends to start simple, but sometimes low-level complexity ripples back to the top, and those ripples can make things more complex than they really needed to be. Bottom up tends to start complex, but identifying that complexity early on leads to better design of the higher-level classes—if the complexity doesn't torpedo the whole system first!

In the final analysis, top-down and bottom-up design aren't competing strategies—they're mutually beneficial. Design is a heuristic process, which means that no solution is guaranteed to work every time. Design contains elements of trial and error. Try a variety of approaches until you find one that works well.

Experimental Prototyping

cc2e.com/0599

Sometimes you can't really know whether a design will work until you better understand some implementation detail. You might not know if a particular database organization will work until you know whether it will meet your performance goals. You might not know whether a particular subsystem design will work until you select the specific GUI libraries you'll be working with. These are examples of the essential "wickedness" of software design—you can't fully define the design problem until you've at least partially solved it.

A general technique for addressing these questions at low cost is experimental prototyping. The word "prototyping" means lots of different things to different people (McConnell 1996). In this context, prototyping means writing the absolute minimum amount of throwaway code that's needed to answer a specific design question.

Prototyping works poorly when developers aren't disciplined about writing the *absolute minimum* of code needed to answer a question. Suppose the design question is, "Can the database framework we've selected support the transaction volume we need?" You don't need to write any production code to answer that question. You don't even need to know the database specifics. You just need to know enough to approximate the problem space—number of tables, number of entries in the tables, and so on. You can then write very simple prototyping code that uses tables with names like *Table1*, *Table2*, and *Column1*, and *Column2*, populate the tables with junk data, and do your performance testing.

Prototyping also works poorly when the design question is not *specific* enough. A design question like "Will this database framework work?" does not provide enough direction for prototyping. A design question like "Will this database framework support 1,000 transactions per second under assumptions X, Y, and Z?" provides a more solid basis for prototyping.

A final risk of prototyping arises when developers do not treat the code as *throwaway* code. I have found that it is not possible for people to write the absolute minimum amount of code to answer a question if they believe that the code will eventually end up in the production system. They end up implementing the system instead of prototyping. By adopting the attitude that once the question is answered the code will be thrown away, you can minimize this risk. One way to avoid this problem is to create prototypes in a different technology than the production code. You could prototype a Java design in Python or mock up a user interface in Microsoft PowerPoint. If you do create prototypes using the production technology, a practical standard that can help is requiring that class names or package names for prototype code be prefixed with *prototype*. That at least makes a programmer think twice before trying to extend prototype code (Stephens 2003).

Used with discipline, prototyping is the workhorse tool a designer has to combat design wickedness. Used without discipline, prototyping adds some wickedness of its own.

Collaborative Design

Cross-Reference For more details on collaborative development, see Chapter 21, “Collaborative Construction.”

In design, two heads are often better than one, whether those two heads are organized formally or informally. Collaboration can take any of several forms:

- You informally walk over to a co-worker’s desk and ask to bounce some ideas around.
- You and your co-worker sit together in a conference room and draw design alternatives on a whiteboard.
- You and your co-worker sit together at the keyboard and do detailed design in the programming language you’re using—that is, you can use pair programming, described in Chapter 21, “Collaborative Construction.”
- You schedule a meeting to walk through your design ideas with one or more co-workers.
- You schedule a formal inspection with all the structure described in Chapter 21.
- You don’t work with anyone who can review your work, so you do some initial work, put it into a drawer, and come back to it a week later. You will have forgotten enough that you should be able to give yourself a fairly good review.
- You ask someone outside your company for help: send questions to a specialized forum or newsgroup.

If the goal is quality assurance, I tend to recommend the most structured review practice, formal inspections, for the reasons described in Chapter 21. But if the goal is to foster creativity and to increase the number of design alternatives generated, not just to find errors, less structured approaches work better. After you’ve settled on a specific design, switching to a more formal inspection might be appropriate, depending on the nature of your project.

How Much Design Is Enough?

We try to solve the problem by rushing through the design process so that enough time is left at the end of the project to uncover the errors that were made because we rushed through the design process.
—Glenford Myers

Sometimes only the barest sketch of an architecture is mapped out before coding begins. Other times, teams create designs at such a level of detail that coding becomes a mostly mechanical exercise. How much design should you do before you begin coding?

A related question is how formal to make the design. Do you need formal, polished design diagrams, or would digital snapshots of a few drawings on a whiteboard be enough?

Deciding how much design to do before beginning full-scale coding and how much formality to use in documenting that design is hardly an exact science. The experience of the team, expected lifetime of the system, desired level of reliability, and size of project and team should all be considered. Table 5-2 summarizes how each of these factors influence the design approach.

Table 5-2 Design Formality and Level of Detail Needed

Factor	Level of Detail Needed in Design Before Construction	Documentation Formality
Design/construction team has deep experience in applications area.	Low Detail	Low Formality
Design/construction team has deep experience but is inexperienced in the applications area.	Medium Detail	Medium Formality
Design/construction team is inexperienced.	Medium to High Detail	Low-Medium Formality
Design/construction team has moderate-to-high turnover.	Medium Detail	—
Application is safety-critical.	High Detail	High Formality
Application is mission-critical.	Medium Detail	Medium-High Formality
Project is small.	Low Detail	Low Formality
Project is large.	Medium Detail	Medium Formality
Software is expected to have a short lifetime (weeks or months).	Low Detail	Low Formality
Software is expected to have a long lifetime (months or years).	Medium Detail	Medium Formality

Two or more of these factors might come into play on any specific project, and in some cases the factors might provide contradictory advice. For example, you might have a highly experienced team working on safety critical software. In that case, you’d probably want to err on the side of the higher level of design detail and formality. In such cases, you’ll need to weigh the significance of each factor and make a judgment about what matters most.

If the level of design is left to each individual, then, when the design descends to the level of a task that you’ve done before or to a simple modification or extension of such a task, you’re probably ready to stop designing and begin coding.

If I can't decide how deeply to investigate a design before I begin coding, I tend to err on the side of going into more detail. The biggest design errors arise from cases in which I thought I went far enough, but it later turns out that I didn't go far enough to realize there were additional design challenges. In other words, the biggest design problems tend to arise not from areas I knew were difficult and created bad designs for, but from areas I thought were easy and didn't create any designs for at all. I rarely encounter projects that are suffering from having done too much design work.

I've never met a human being who would want to read 17,000 pages of documentation, and if there was, I'd kill him to get him out of the gene pool.
—Joseph Costello

On the other hand, occasionally I have seen projects that are suffering from too much design *documentation*. Gresham's Law states that “programmed activity tends to drive out nonprogrammed activity” (Simon 1965). A premature rush to polish a design description is a good example of that law. I would rather see 80 percent of the design effort go into creating and exploring numerous design alternatives and 20 percent go into creating less polished documentation than to have 20 percent go into creating mediocre design alternatives and 80 percent go into polishing documentation of designs that are not very good.

Capturing Your Design Work

cc2e.com/0506

The traditional approach to capturing design work is to write up the designs in a formal design document. However, you can capture designs in numerous alternative ways that work well on small projects, informal projects, or projects that need a lightweight way to record a design:

The bad news is that, in our opinion, we will never find the philosopher's stone. We will never find a process that allows us to design software in a perfectly rational way. The good news is that we can fake it.
—David Parnas and Paul Clements

Insert design documentation into the code itself Document key design decisions in code comments, typically in the file or class header. When you couple this approach with a documentation extractor like JavaDoc, this assures that design documentation will be readily available to a programmer working on a section of code, and it improves the chance that programmers will keep the design documentation reasonably up to date.

Capture design discussions and decisions on a Wiki Have your design discussions in writing, on a project Wiki (that is, a collection of Web pages that can be edited easily by anyone on your project using a Web browser). This will capture your design discussions and decision automatically, albeit with the extra overhead of typing rather than talking. You can also use the Wiki to capture digital pictures to supplement the text discussion, links to websites that support the design decision, white papers, and other materials. This technique is especially useful if your development team is geographically distributed.

Write e-mail summaries After a design discussion, adopt the practice of designating someone to write a summary of the discussion—especially what was decided—and send it to the project team. Archive a copy of the e-mail in the project's public e-mail folder.

Use a digital camera One common barrier to documenting designs is the tedium of creating design drawings in some popular drawing tools. But the documentation choices are not limited to the two options of “capturing the design in a nicely formatted, formal notation” vs. “no design documentation at all.”

Taking pictures of whiteboard drawings with a digital camera and then embedding those pictures into traditional documents can be a low-effort way to get 80 percent of the benefit of saving design drawings by doing about 1 percent of the work required if you use a drawing tool.

Save design flip charts There’s no law that says your design documentation has to fit on standard letter-size paper. If you make your design drawings on large flip chart paper, you can simply archive the flip charts in a convenient location—or, better yet, post them on the walls around the project area so that people can easily refer to them and update them when needed.

cc2e.com/0513

Use CRC (Class, Responsibility, Collaborator) cards Another low-tech alternative for documenting designs is to use index cards. On each card, designers write a class name, responsibilities of the class, and collaborators (other classes that cooperate with the class). A design group then works with the cards until they’re satisfied that they’ve created a good design. At that point, you can simply save the cards for future reference. Index cards are cheap, unintimidating, and portable, and they encourage group interaction (Beck 1991).

Create UML diagrams at appropriate levels of detail One popular technique for diagramming designs is called Unified Modeling Language (UML), which is defined by the Object Management Group (Fowler 2004). Figure 5-6 earlier in this chapter was one example of a UML class diagram. UML provides a rich set of formalized representations for design entities and relationships. You can use informal versions of UML to explore and discuss design approaches. Start with minimal sketches and add detail only after you’ve zeroed in on a final design solution. Because UML is standardized, it supports common understanding in communicating design ideas and it can accelerate the process of considering design alternatives when working in a group.

These techniques can work in various combinations, so feel free to mix and match these approaches on a project-by-project basis or even within different areas of a single project.

5.5 Comments on Popular Methodologies

The history of design in software has been marked by fanatic advocates of wildly conflicting design approaches. When I published the first edition of *Code Complete* in the early 1990s, design zealots were advocating dotting every design *i* and crossing every design *t* before beginning coding. That recommendation didn’t make any sense.

People who preach software design as a disciplined activity spend considerable energy making us all feel guilty. We can never be structured enough or object-oriented enough to achieve nirvana in this lifetime. We all truck around a kind of original sin from having learned Basic at an impressionable age. But my bet is that most of us are better designers than the purists will ever acknowledge.
—P. J. Plauger

As I write this edition in the mid-2000s, some software swamis are arguing for not doing any design at all. “Big Design Up Front is *BDUF*,” they say. “*BDUF* is bad. You’re better off not doing any design before you begin coding!”

In ten years the pendulum has swung from “design everything” to “design nothing.” But the alternative to *BDUF* isn’t no design up front, it’s a Little Design Up Front (*LDUF*) or Enough Design Up Front—*ENUF*.

How do you tell how much is enough? That’s a judgment call, and no one can make that call perfectly. But while you can’t know the exact right amount of design with any confidence, two amounts of design are guaranteed to be wrong every time: designing every last detail and not designing anything at all. The two positions advocated by extremists on both ends of the scale turn out to be the only two positions that are always wrong!

As P.J. Plauger says, “The more dogmatic you are about applying a design method, the fewer real-life problems you are going to solve” (Plauger 1993). Treat design as a wicked, sloppy, heuristic process. Don’t settle for the first design that occurs to you. Collaborate. Strive for simplicity. Prototype when you need to. Iterate, iterate, and iterate again. You’ll be happy with your designs.

Additional Resources

cc2e.com/0520

Software design is a rich field with abundant resources. The challenge is identifying which resources will be most useful. Here are some suggestions.

Software Design, General

Weisfeld, Matt. *The Object-Oriented Thought Process*, 2d ed. SAMS, 2004. This is an accessible book that introduces object-oriented programming. If you’re already familiar with object-oriented programming, you’ll probably want a more advanced book, but if you’re just getting your feet wet in object orientation, this book introduces fundamental object-oriented concepts, including objects, classes, interfaces, inheritance, polymorphism, overloading, abstract classes, aggregation and association, constructors/destructors, exceptions, and others.

Riel, Arthur J. *Object-Oriented Design Heuristics*. Reading, MA: Addison-Wesley, 1996. This book is easy to read and focuses on design at the class level.

Plauger, P. J. *Programming on Purpose: Essays on Software Design*. Englewood Cliffs, NJ: PTR Prentice Hall, 1993. I picked up as many tips about good software design from reading this book as from any other book I’ve read. Plauger is well-versed in a wide-variety of design approaches, he’s pragmatic, and he’s a great writer.

Meyer, Bertrand. *Object-Oriented Software Construction*, 2d ed. New York, NY: Prentice Hall PTR, 1997. Meyer presents a forceful advocacy of hard-core object-oriented programming.

Raymond, Eric S. *The Art of UNIX Programming*. Boston, MA: Addison-Wesley, 2004. This is a well-researched look at software design through UNIX-colored glasses. Section 1.6 is an especially concise 12-page explanation of 17 key UNIX design principles.

Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2d ed. Englewood Cliffs, NJ: Prentice Hall, 2001. This book is a popular introduction to object-oriented design in the context of the Unified Process. It also discusses object-oriented analysis.

Software Design Theory

Parnas, David L., and Paul C. Clements. “A Rational Design Process: How and Why to Fake It.” *IEEE Transactions on Software Engineering* SE-12, no. 2 (February 1986): 251–57. This classic article describes the gap between how programs are really designed and how you sometimes wish they were designed. The main point is that no one ever really goes through a rational, orderly design process but that aiming for it makes for better designs in the end.

I’m not aware of any comprehensive treatment of information hiding. Most software-engineering textbooks discuss it briefly, frequently in the context of object-oriented techniques. The three Parnas papers listed below are the seminal presentations of the idea and are probably still the best resources on information hiding.

Parnas, David L. “On the Criteria to Be Used in Decomposing Systems into Modules.” *Communications of the ACM* 5, no. 12 (December 1972): 1053–58.

Parnas, David L. “Designing Software for Ease of Extension and Contraction.” *IEEE Transactions on Software Engineering* SE-5, no. 2 (March 1979): 128–38.

Parnas, David L., Paul C. Clements, and D. M. Weiss. “The Modular Structure of Complex Systems.” *IEEE Transactions on Software Engineering* SE-11, no. 3 (March 1985): 259–66.

Design Patterns

Gamma, Erich, et al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995. This book by the “Gang of Four” is the seminal book on design patterns.

Shalloway, Alan, and James R. Trott. *Design Patterns Explained*. Boston, MA: Addison-Wesley, 2002. This book contains an easy-to-read introduction to design patterns.

Design in General

Adams, James L. *Conceptual Blockbusting: A Guide to Better Ideas*, 4th ed. Cambridge, MA: Perseus Publishing, 2001. Although not specifically about software design, this book was written to teach design to engineering students at Stanford. Even if you never design anything, the book is a fascinating discussion of creative thought processes. It includes many exercises in the kinds of thinking required for effective design. It also contains a well-annotated bibliography on design and creative thinking. If you like problem solving, you'll like this book.

Polya, G. *How to Solve It: A New Aspect of Mathematical Method*, 2d ed. Princeton, NJ: Princeton University Press, 1957. This discussion of heuristics and problem solving focuses on mathematics but is applicable to software development. Polya's book was the first written about the use of heuristics in mathematical problem solving. It draws a clear distinction between the messy heuristics used to discover solutions and the tidier techniques used to present them once they've been discovered. It's not easy reading, but if you're interested in heuristics, you'll eventually read it whether you want to or not. Polya's book makes it clear that problem solving isn't a deterministic activity and that adherence to any single methodology is like walking with your feet in chains. At one time, Microsoft gave this book to all its new programmers.

Michalewicz, Zbigniew, and David B. Fogel. *How to Solve It: Modern Heuristics*. Berlin: Springer-Verlag, 2000. This is an updated treatment of Polya's book that's quite a bit easier to read and that also contains some nonmathematical examples.

Simon, Herbert. *The Sciences of the Artificial*, 3d ed. Cambridge, MA: MIT Press, 1996. This fascinating book draws a distinction between sciences that deal with the natural world (biology, geology, and so on) and sciences that deal with the artificial world created by humans (business, architecture, and computer science). It then discusses the characteristics of the sciences of the artificial, emphasizing the science of design. It has an academic tone and is well worth reading for anyone intent on a career in software development or any other "artificial" field.

Glass, Robert L. *Software Creativity*. Englewood Cliffs, NJ: Prentice Hall PTR, 1995. Is software development controlled more by theory or by practice? Is it primarily creative or is it primarily deterministic? What intellectual qualities does a software developer need? This book contains an interesting discussion of the nature of software development with a special emphasis on design.

Petroski, Henry. *Design Paradigms: Case Histories of Error and Judgment in Engineering*. Cambridge: Cambridge University Press, 1994. This book draws heavily from the field of civil engineering (especially bridge design) to explain its main argument that successful design depends at least as much upon learning from past failures as from past successes.

Standards

IEEE Std 1016-1998, Recommended Practice for Software Design Descriptions. This document contains the IEEE-ANSI standard for software-design descriptions. It describes what should be included in a software-design document.

IEEE Std 1471-2000. Recommended Practice for Architectural Description of Software Intensive Systems. Los Alamitos, CA: IEEE Computer Society Press. This document is the IEEE-ANSI guide for creating software architecture specifications.

cc2e.com/0527

CHECKLIST: Design in Construction

Design Practices

- ☐ Have you iterated, selecting the best of several attempts rather than the first attempt?
- ☐ Have you tried decomposing the system in several different ways to see which way will work best?
- ☐ Have you approached the design problem both from the top down and from the bottom up?
- ☐ Have you prototyped risky or unfamiliar parts of the system, creating the absolute minimum amount of throwaway code needed to answer specific questions?
- ☐ Has your design been reviewed, formally or informally, by others?
- ☐ Have you driven the design to the point that its implementation seems obvious?
- ☐ Have you captured your design work using an appropriate technique such as a Wiki, e-mail, flip charts, digital photography, UML, CRC cards, or comments in the code itself?

Design Goals

- ☐ Does the design adequately address issues that were identified and deferred at the architectural level?
- ☐ Is the design stratified into layers?
- ☐ Are you satisfied with the way the program has been decomposed into subsystems, packages, and classes?
- ☐ Are you satisfied with the way the classes have been decomposed into routines?
- ☐ Are classes designed for minimal interaction with each other?

- ❑ Are classes and subsystems designed so that you can use them in other systems?
- ❑ Will the program be easy to maintain?
- ❑ Is the design lean? Are all of its parts strictly necessary?
- ❑ Does the design use standard techniques and avoid exotic, hard-to-understand elements?
- ❑ Overall, does the design help minimize both accidental and essential complexity?

Key Points

- Software's Primary Technical Imperative is *managing complexity*. This is greatly aided by a design focus on simplicity.
- Simplicity is achieved in two general ways: minimizing the amount of essential complexity that anyone's brain has to deal with at any one time, and keeping accidental complexity from proliferating needlessly.
- Design is heuristic. Dogmatic adherence to any single methodology hurts creativity and hurts your programs.
- Good design is iterative; the more design possibilities you try, the better your final design will be.
- Information hiding is a particularly valuable concept. Asking "What should I hide?" settles many difficult design issues.
- Lots of useful, interesting information on design is available outside this book. The perspectives presented here are just the tip of the iceberg.

Index

Symbols and Numbers

* (pointer declaration symbol), 332, 334–335, 763
&x (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 instantiation, 132
 files as, 130
 guidelines, 130–131
 hiding information with, 127
 instantiating, 132
 implicit instantiation, 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

- code tuning, *continued*
 - subexpression elimination, 638–639
 - summary of approach for, 606
 - system calls, 599–600, 633–634
 - tools, 720
 - unrolling loops, 618–620
 - unswitching loops, 616–617
 - variations in environments for, 594
 - when to tune, 596
- code-generation wizards, 718
- coding. *See also* construction; software construction overview
 - conventions. *See* conventions, coding
 - practices checklist, 69
 - sequential. *See* straight-line code
 - software construction as, 5
 - style. *See* layout
- cohesion
 - interfaces, class, 138
 - routines, designing with, 168–171
 - strength reduction, 623–624, 630–632
- coincidental cohesion, 170
- collaboration
 - code reading, 494
 - collective ownership benefits, 482
 - comparisons of techniques, table of, 495–496
 - cost advantage, 480–481
 - defined, 479, 480
 - design phase, 115
 - development time benefit, 480
 - dog-and-pony shows, 495
 - extending beyond construction, 483
 - Extreme Programming method, 482
 - formal inspections. *See* formal inspections
 - General Principle of Software Quality, 481
 - inspections. *See* formal inspections
 - key points, 497
 - mentoring aspect of, 482
 - pair programming. *See* pair programming
 - purpose of, 480
 - standards, IEEE, 497
 - testing, compared to, 481
 - walk-throughs, 492–493
- collections, refactoring, 572
- collective ownership, 482. *See also* collaboration
- comments. *See also* documentation
 - /** vs. *//*, 790
 - abbreviations in, 799
 - algorithms, 809
 - argument against, 782
 - authorship, 811
 - bad code, on, 568
 - blank lines around, 765–766
 - Book Paradigm for, 812–813
 - categories of, 786–788
 - checklists, 774, 816–817
 - classes, 810
 - coded meanings, 802–803
 - control structures, 804–805, 817
 - declarations with, 794, 802–803, 816
 - descriptions of code intent, 787
 - distance to code guideline, 806
 - efficient creation of, 788–791
 - endline comments, 793–795
 - errors, marking workarounds, 800
 - explanatory, 786
 - files, 810–811
 - flags, bit level, 803
 - global variables, 803, 809
 - indentation guidelines, 764–765
 - individual lines with, 792–795
 - input data, 803, 808
 - integrating into development, 791
 - interfaces, class, 810
 - interfaces, routine, 808
 - Javadoc, 807, 815
 - key points, 817
 - layout guidelines, 763–766
 - legal notices, 811
 - length of descriptions, 806
 - level of code intent, 795–796
 - loops, 804–805
 - maintenance of, 220, 788–791, 794
 - major vs. minor, 799–800
 - markers, 787
 - non-code essential information, 788
 - numerical data, 802
 - optimum density of, 792
 - output data, 808
 - paragraphs of code with, 795–801, 816
 - parameter declarations, 806–807
 - parts of programs, 809
 - performance considerations, 791
 - preceding code rule, 798
 - proportionality of, 806
 - pseudocode, deriving from, 220, 784, 791
 - purpose of, 782
 - repeating code with, 786
 - resources on, 815
 - routines with, 805–809, 817
 - self-commenting code, 796–797
 - Socratic dialog about, 781–785
 - standards, IEEE, 813–814
 - style differences, managing, 683
 - style violations, 801
 - summaries of code, 787
 - surprises, 798
 - tricky code, 798, 801
 - undocumented features, 800
 - variables, 803
 - version control, 811
 - why vs. how, 797–798
 - workarounds, 800
- commitment delay strategy, 52
- communication skills, importance of, 828
- communicational cohesion, 169
- communications, development team, 650
- comparisons
 - boolean. *See* boolean tests
 - floating-point equality, 295–296
 - mixed data types, 293
- compilers
 - binding during compilation, 252–253
 - broken builds, 703
 - data type warnings, 293
 - debugging tools, as, 557, 827
 - errors, finding in routines, 230–231
 - line numbers, debugging with, 549
 - messages, treatment of, 549, 826–827
 - multiple error messages, 550
 - optimizations by, 596–597
 - performance tuning considerations, 590
 - project-wide standards for, 557
 - speeds from optimization, table of, 597
 - tools for, 716
 - tricky code optimization, 597
 - validators with, 231
 - warnings, 293, 557

- completeness of requirements
 - checklist, 43
- complex data types. *See* structures
- complexity
 - abstraction for handling, 839
 - classes for reducing, 152
 - coding conventions for reducing, 839
 - control structure contributions to, 456–459
 - conventions for managing, 844–845
 - decision points, counting, 458
 - importance of, 457
 - isolation, classes for, 153
 - live time, 459
 - management, 77–79, 844–845
 - McCabe's metric, 457–458
 - mental objects held, measure of, 457
 - methods for handling, 837–839
 - minimization goal, 80
 - patterns, reducing with, 103
 - problem domain, working at, 845
 - reliability correlated with, 457
 - routines for reducing, 164
 - size of projects, effect on, 656–657
 - span, 459
- component testing, 499
- components, buying, 18, 51
- Composite pattern, 104
- compound boundaries, 514
- compound statements. *See* blocks
- computed-value qualifiers of
 - variable names, 263–264
- computer-aided software
 - engineering (CASE) tools, 710
- conditional statements
 - boolean function calls with, 359
 - boolean variables recommended, 301–302
 - case statements. *See* case statements
 - chained if-then-else statements, 358–360
 - checklist, 365
 - common cases first guideline, 359–360
 - comparing performance of, 614
 - covering all cases, 360
 - defined, 355
 - eliminating testing redundancy, 610–611
 - else clauses, 358–360
 - equality, branching on, 355
 - error processing examples, 356–357
 - frequency, testing in order of, 612–613
 - if statements. *See* if statements
 - key points, 366
 - lookup tables, substituting, 614–615
 - looping, conditional. *See* loops
 - normal case first guideline, 356–357
 - normal path first guideline, 355
 - null if clauses, 357
 - plain if-then statements, 355–357
 - refactoring, 573
 - short-circuit evaluation, 610
 - switch statements. *See* case statements
- confessional debugging, 547–548
- configuration management
 - architectural anticipation of change, 52
 - backup plans, 669, 670
 - boards, change-control, 667
 - bureaucratic considerations, 667
 - checklist, 669–670
 - code changes, 667–668
 - cost, estimating, 666
 - defined, 664
 - design changes, 666–667
 - estimating change costs, 666
 - grouping change requests, 666
 - high change volumes, 666
 - identifying areas of change, 97–99
 - machine configurations, reproducing, 668
 - purpose of, 664–665
 - requirements changes, 41, 664, 666–667
 - resources on, 670
 - SCM, 665
 - tool version control, 668
 - version-control software, 668
- const keyword, C++, 176, 177, 243, 274, 333
- constants
 - checklist, 317
 - consistency rule, 309
 - declarations using, 308
 - defined, 307
 - emulation by global variables, 338
 - initializing, 243
 - literals, avoiding with, 308–309
 - naming, 270, 273, 277–279
 - purpose of, 307
 - refactoring, 571
 - simulating in languages lacking, 309
- construction. *See also* software
 - construction overview
 - collaborative. *See* collaboration
 - decisions. *See* construction decisions
 - decisions
 - guidelines, 66
 - managing. *See* managing construction
 - percent of total activity, by size of project, 654–655
 - prerequisites. *See* prerequisites, upstream
 - quality of. *See* quality of software
 - resources on, 856
 - schedules, estimating. *See* construction schedules, estimating
 - size of projects, effects on. *See* size of projects
 - tools for. *See* programming tools
- construction decisions
 - checklist of major construction practices, 69–70
 - coding practices checklist, 69
 - early-wave environments, 67
 - key points for, 70
 - major construction practices, selecting, 69–70
 - mature technology environments, 67
 - programming conventions, 66–66
 - programming into languages, 68–69
 - programming languages. *See* programming language choice
 - quality assurance checklist, 70
 - teamwork checklist, 69
 - technology waves, determining your location in, 66–69
 - tools checklist, 70
- construction schedules, estimating
 - approaches to, list of, 671
 - catching up from behind, 675–676
 - controlling vs. estimating, 675
 - factors influencing, 674–675
 - level of detail for, 672
 - multiple techniques with comparisons, 672
 - objectives, establishing, 671
 - optimism, 675

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* 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

- files
 - ADTs, treating as, 130
 - authorship records for, 811
 - C++ source file order, 773
 - deleting multiple example, 401–402
 - documenting, 810–811
 - layout within, 771–773
 - naming, 772, 811
 - routines in, 772
- final keyword, Java, 243
- finally statements, 404–405
- fixing defects
 - checking fixes, 553
 - checklist, 560
 - diagnosis confirmation, 551
 - hurrying, impact of, 551
 - initialization defects, 553
 - maintenance issues, 553
 - one change at a time rule, 553
 - reasoning for changes, 553
 - saving unfixed code, 552
 - similar defects, looking for, 554
 - special cases, 553
 - symptoms, fixing instead of problems, 552–553
 - understand first guideline, 550–551
 - unit tests for, 554
- flags
 - change, identifying areas of, 98–99
 - comments for bit-level meanings, 803
 - enumerated types for, 266–267
 - gotos, rewriting with, 403–404
 - names for, 266–267
 - semantic coupling with, 102
- flexibility
 - coupling criteria for, 100–101
 - defined, 464
- floating-point data types
 - accuracy limitations, 295
 - BCD, 297
 - checklist, 316
 - costs of operations, 602
 - equality comparisons, 295–296
 - magnitudes, greatly different, operations with, 295
 - rounding errors, 297
 - Visual Basic types, 297
- for loops
 - advantages of, 374
 - formatting, 732–733, 746–747
 - indexes, 377–378
 - purpose of, 372
- foreach loops, 367, 372
- formal inspections
 - author role, 486
 - benefit summary, 491
 - blame game, 490
 - checklist, 491–492
 - CMM, 491
 - Code Complete example, 490–491
 - compared to other collaboration, 495–496
 - defined, 485
 - egos in, 490
 - error checklists, 489
 - expected results from, 485–486
 - fine-tuning, 489
 - follow-up stage, 489
 - inspection meetings, 488
 - key points, 497
 - management role, 486–487
 - moderator role, 486
 - overview stage, 487
 - performance appraisals from, 487
 - planning stage, 487
 - preparation stage, 487–488
 - procedure for, 487–489
 - rate of code review, 488
 - reports, 488–489
 - resources for, 496–497
 - reviewer role, 486
 - reviews, compared to, 485
 - rework stage, 489
 - roles in, 486–487
 - scenarios approach, 488
 - scribe role, 486
 - stages of, 487–489
 - three-hour solutions meeting, 489
- formal technical reviews, 467
- formatting code. *See* layout
- Fortran, 64
- functional cohesion, 168–169
- functional specification. *See* requirements
- functions. *See also* routines
 - calculations converted to example, 166–167
 - defined, 181
 - disallowing, 150
 - key point for, 186
 - naming conventions for, 172, 181
 - private, overriding, 146
 - return values, setting, 182
 - status as return value, 181
 - when to use, 181–182
- Fundamental Theorem of Formatting, 732
- G**
 - General Principle of Software Quality
 - collaboration effects, 481
 - costs, 522
 - debugging, 537
 - defined, 474–475
 - global variables
 - access routines for. *See* access routines
 - aliasing problems with, 336–337
 - alternatives to, 339–342
 - annotating, 343
 - changes to, inadvertent, 336
 - checklist for, 343–344
 - class variable alternatives, 339
 - code reuse problems, 337
 - commenting, 803, 809
 - enumerated types emulation by, 338
 - g_ prefix guideline, 340
 - hiding implementation in classes, 153
 - information hiding problems with, 95–96
 - initialization problems, 337
 - intermediate results, avoiding, 343
 - key points, 344
 - local first guideline, 339
 - locking, 341
 - modularity damaged by, 337–338
 - named constants emulation by, 338
 - naming, 263, 273, 277, 278, 279, 342
 - objects for, monster, 343
 - overview of, 335–336
 - persistence of, 251
 - preservation of values with, 338
 - re-entrant code problems, 337
 - refactoring, 568
 - risk reduction strategies, 342–343
 - routines using as parameters, 336
 - semantic coupling with, 102
 - streamlining data use with, 338
 - tramp data, eliminating with, 338
 - god classes, 155
 - gonzo programming, 832
 - good data, testing, 515–516
 - goto statements
 - Ada, inclusion in, 399
 - advantages of, 399
 - alternatives compared with, 405
 - checklist, 410

goto statements, *continued*
 deallocation with, 399
 disadvantages of, 398–399
 duplicate code, eliminating with, 399
 else clauses with, 406–407
 error processing with, 401–402
 Fortran's use of, 399
 forward direction guideline, 408
 guidelines, 407–408
 indentation problem with, 398
 key points, 410
 layout guidelines, 750–751
 legitimate uses of, 407–408
 optimization problem with, 398
 phony debating about, 400–401
 readability issue, 398
 resources for, 409–410
 rewritten with nested ifs, 402–403
 rewritten with status variables, 403–404
 rewritten with try-finally, 404–405
 trivial rewrite example, 400–401
 unused labels, 408
 graphical design tools, 710
 grep, 711
 growing a system metaphor, 14–15
 GUIs (graphical user interfaces)
 architecture prerequisites, 47
 refactoring data from, 576
 subsystem design, 85

H

habits of programmers, 833–834
 hacking approach to design, 233
 hardware
 dependencies, changing, 98
 performance enhancement with, 591
 has a relationships, 143
 heuristic design
 abstractions, forming consistent, 89–90
 alternatives from patterns, 103
 avoiding failure, 106–107
 binding time considerations, 107
 bottom-up approach to design, 112–113
 brute force, 107
 capturing work, 117–118
 central points of control, 107

change, identifying areas of, 97–99
 checklist for, 122–123
 collaboration, 115
 communications benefit from patterns, 104
 completion of, determining, 115–117
 coupling considerations, 100–102
 diagrams, drawing, 107
 divide and conquer technique, 111
 encapsulation, 90–91
 error reduction with patterns, 103
 formality of, determining, 115–117
 formalizing class contracts, 106
 goals checklist, 122–123
 guidelines for using, 109–110
 hierarchies for, 105–106
 information hiding, 92–97, 120
 inheritance, 91–92
 interfaces, formalizing as contracts, 106
 iteration practice, 111–117
 key points, 123
 level of detail needed, 115–117
 modularity, 107
 multiple approach suggestion, 110
 nature of design process, 76
 nondeterministic basis for, 87
 object-oriented, resource for, 119
 objects, real world, finding, 87–89
 patterns, 103–105, 120
 practices, 110–118, 122
 prototyping, 114–115
 resources for, 121
 responsibilities, assigning to objects, 106
 strong cohesion, 105
 summary list of rules, 108
 testing, anticipating, 106
 top-down approach, 111–112, 113
 heuristics
 algorithms compared to, 12
 design with. *See* heuristic design
 error guessing, 513
 hiding. *See* information hiding
 hierarchies, benefits of, 105–106
 high fan-in design goal, 80
 human aspects of software development. *See* character, personal

humility, role in character, 821, 826, 834
 Hungarian naming convention, 279
 hybrid coupling of variables, 256–257

I

I/O (input/output)
 architecture prerequisites, 49
 change, identifying areas of, 98
 performance considerations, 598–599
 IDEs (Integrated Development Environments), 710–711
 IEEE (Institute for Electric and Electrical Engineers), 813
 if statements
 boolean function calls with, 359
 break blocks, simplification with, 446–447
 case statements, compared to, 360, 614
 case statements, converting to, 448–449, 451
 chains of, 358–360
 checklist, 365
 common cases first guideline, 359–360
 continuation lines in, 757
 covering all cases, 360
 else clauses, 358–360, 406–407
 equality, branching on, 355
 error processing examples, 356–357
 factoring to routines, 449–451
 flipped, 358
 frequency, testing in order of, 612–613
 gotos rewritten with, 402–403, 406–407
 if-then-else statements, converting to, 447–448
 key points, 366
 lookup tables, substituting, 614–615
 multiple returns nested in, 392–393
 negatives in, making positive, 435–436
 nested. *See* nested if statements
 normal case first guideline, 356–357
 normal path first guideline, 355
 null if clauses, 357

- 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 instantiating, 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
- instanting 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
- programmatic 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
 - τ_ 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

objects, *continued*

- class names, differentiating from, 272–273
 - classes, contrasted to, 86
 - containment, identifying, 88
 - deleting objects, 206
 - factory methods, 103–104, 452–453, 577
 - identifying, 88
 - inheritance, identifying, 88. *See also* inheritance
 - interfaces, designing, 89. *See also* interfaces, class
 - operations, identifying, 88
 - parameters, using as, 179, 574
 - protected interfaces, designing, 89
 - public vs. private members, designing, 89
 - real world, finding, 87–89
 - refactoring, 574–576
 - reference objects, 574
 - responsibilities, assigning to, 106
 - singleton property, enforcing, 151
 - steps in designing, 87–89
- Observer pattern, 104
- off-by-one errors
- boundary analysis, 513–514
 - fixing, approaches to, 553
- offensive programming, 206
- one-in, one-out control constructs, 454
- operating systems, 590
- operations, costs of common, 601–603
- opposites for variable names, 264
- optimization, premature, 840. *See also* performance tuning
- oracles, software, 851
- out keyword creation, 175–176
- overengineering, 51
- overflows, integer, 293–295
- overlay linkers, 716
- overridable routines, 145–146, 156
- oyster farming metaphor, 15–16

P

- packages, 156–157
- paging operations, 599
- pair programming
 - benefits of, 484
 - checklist, 484
 - coding standards support for, 483
 - compared to other collaboration, 495–496

- defined, 483
 - inexperienced pairs, 484
 - key points, 497
 - pace, matching, 483
 - personality conflicts, 484
 - resources, 496
 - rotating pairs, 483
 - team leaders, 484
 - visibility of monitor, 484
 - watching, 483
 - when not to use, 483
- parameters of routines
- abstraction and object parameters, 179
 - actual, matching to formal, 180
 - asterisk (*) rule for pointers, 334–335
 - behavior dependence on, 574
 - by reference vs. by value, 333
 - checklist for, 185
 - C-library order, 175
 - commenting, 806–807
 - const prefix, 176, 177, 274
 - dependencies, clarifying, 349
 - documentation, 178
 - enumerated types for, 303
 - error variables, 176
 - formal, matching to actual, 180
 - global variables for, 336
 - guidelines for use in routines, 174–180
 - in keyword creation, 175–176
 - input-modify-output order, 174–175
 - Java, 176–177
 - list size as refactoring indicator, 566
 - matching actual to formal, 180
 - naming, 178, 180, 274, 277, 278, 279
 - number of, limiting, 178
 - objects, passing, 179
 - order for, 174–176
 - out keyword creation, 175–176
 - passing, types of, 333
 - refactoring, 571, 573
 - status, 176
 - structures as, 322
 - using all of rule, 176
 - variables, using as, 176–177
 - Visual Basic, 180
- parentheses
- balancing technique, 437–438
 - layout with, 738
- Pareto Principle, 592

passing parameters, 333

patterns

- advantages of, 103–104
 - alternatives suggested by, 103
 - communications benefit, 104
 - complexity reduction with, 103
 - disadvantages of, 105
 - error reduction benefit, 103
 - Factory Method, 103–104
 - resource for, 120
 - table of, 104
- people first theme. *See* readability
- performance appraisals, 487
- performance tuning
- algorithm choice, 590
 - architecture prerequisites, 48
 - arrays, 593–594, 603–604
 - checklist, 607–608
 - code tuning for. *See* code tuning
 - comments, effects on, 791
 - competing objectives dilemma, 595, 605
 - compiler considerations, 590, 596–597
 - correctness, importance of, 595–596
 - database indexing, 601
 - defects in code, 601
 - DES example, 605–606
 - design view, 589–590
 - feature specific, 595
 - hardware considerations, 591
 - inefficiency, sources of, 598–601
 - information hiding
 - considerations of, 96
 - input/output, 598–599
 - interpreted vs. compiled languages, 600–601
 - key points, 608
 - lines of code, minimizing number of, 593–594
 - measurement of, 603–604
 - memory vs. file operations, 598–599
 - old wives' tales, 593–596
 - operating system considerations, 590
 - operations, costs of common, 601–603
 - overview of, 643–644
 - paging operations, 599
 - premature optimization, 840
 - program requirements view of, 589
 - purpose of, 587

- quality of code, impact on, 588
- resource goals, 590
- resources, 606–607
- routine design, 165, 222–223, 590
- speed, importance of, 595–596
- summary of approach for, 606
- system calls, 599–600
- timing issues, 604
- user view of coding, 588
- when to tune, 596
- periodicals on programming, 859–860
- Perl, 65
- persistence of variables, 251–252, 831
- personal character. *See* character, personal
- perturbbers. *See* system perturbbers
- phased integration, 691–692
- phonic abbreviations of names, 283
- PHP (PHP Hypertext Processor), 65, 600
- physical environment for programmers, 684–685
- planning
 - analogy argument for, 27–28
 - building metaphor for, 18–19
 - data arguing for, 28–30
 - good practices table for, 31–32
 - logical argument for, 27
- pointers
 - * (pointer declaration symbol), 332, 334–335, 763
 - & (pointer reference symbol), 332
 - > (pointer symbol), 328
 - address of, 323, 326
 - allocation of, 326, 330, 331
 - alternatives to, 332
 - as function return values, 182
 - asterisk (*) rule, 334–335
 - auto_ptrs, 333
 - bounds checking tools, 527
 - C language, 334–335
 - C++ examples, 325, 328–334
 - C++ guidelines, 332–334
 - checking before using, 326, 331
 - checklist for, 344
 - comparisons with, 441
 - contents, interpretation of, 324–325
 - cover routines for, 331–332
 - dangers of, 323, 325
 - data types pointed to, 324–325
 - deallocation of, 326, 330, 332
 - debugging aids, 208–209
 - declaring, 325–326, 763
 - deleting, 330–331, 332
 - diagramming, 329
 - dog tag fields, 326–327
 - explicit typing of, 334
 - explicitly redundant fields, 327
 - extra variables for clarity, 327–329
 - hiding operations with routines, 165
 - initializing, 241, 244, 325–326
 - interpretation of address contents, 324–325
 - isolating operations of, 325
 - key points, 344
 - languages not providing, 323
 - linked lists, deleting in, 330
 - location in memory, 323
 - memory corruption by, 325–327
 - memory parachutes, 330
 - null, setting to after deleting, 330
 - null, using as warnings, 849
 - overwriting memory with junk, 330
 - parts of, 323
 - passing by reference, 333
 - references, C++, 332
 - resources for, 343
 - SAFE_ routines for, 331–332
 - simplifying complicated expressions, 329
 - sizeof(), 335
 - smart, 334
 - string operations in C, 299
 - type casting, avoiding, 334
 - variables referenced by, checking, 326
- polymorphism
 - case statements, replacing with, 147–148
 - defined, 92
 - language-specific rules, 156
 - nested ifs, converting to, 452–453
- polynomial expressions, 631–632
- portability
 - data types, defining for, 315–316
 - defined, 464
 - routines for, 165
- postconditions
 - routine design with, 221
 - verification, 192–193
- PPP (Pseudocode Programming Process)
 - algorithms, researching, 223
 - alternates to, 232–233
 - checking for errors, 230–231
 - checklist for, 233–234
 - cleanup steps, 232
 - coding below comments, 227–229
 - coding routines from, 225–229
 - 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
 - header comments for routines, 223
 - high-level comments from, 226–227
 - iterating, 225
 - key points for, 234
 - naming routines, 222
 - performance considerations, 222–223
 - prerequisites, 221
 - problem definition, 221
 - refactoring, 229
 - removing errors, 231
 - repeating steps, 232
 - reviewing pseudocode, 224–225
 - stepping through code, 231
 - testing the code, 222, 231
 - writing pseudocode step, 223–224
- precedence, misleading, 733
- preconditions
 - routine design with, 221
 - verification, 192–193
- prefixes, standardization of, 279–281
- premature optimization, 840
- preparation. *See* prerequisites, upstream
- preprocessors
 - C++, 207–208
 - debugging aids, removing with, 207–208
 - purpose of, 718–719
 - writing, 208
- prerequisites, upstream
 - analogy argument for, 27–28
 - architectural. *See* architecture
 - boss readiness test, 30–31
 - checklist for, 59

- 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

- specific functional requirements
 - checklist, 42
- specific nonfunctional requirements
 - checklist, 42
- specification. *See* requirements
- speed improvement checklist, 642–643. *See also* code tuning; performance tuning
- SQL, 65
- stabilizing errors, 542–543
- stair-step access tables, 426–429
- standards, overview of, 814
- state variables. *See* status variables
- statements
 - checklist, 774
 - closely-related elements, 755–756
 - continuation layout, 754–758
 - ends of continuations, 756–757
 - incomplete, 754–755
 - length of, 753
 - refactoring, 572–573, 577–578
 - sequential. *See* straight-line code
- status reporting, 827
- status variables
 - bit-level meanings, 803
 - change, identifying areas of, 98–99
 - enumerated types for, 266–267
 - gotos rewritten with, 403–404
 - names for, 266–267
 - semantic coupling of, 102
- straight-line code
 - checklist, 353
 - clarifying dependencies, 348–350
 - dependencies concept, 347
 - documentation, 350
 - error checking, 350
 - grouping related statements, 352–353
 - hidden dependencies, 348
 - initialization order, 348
 - naming routines, 348–349
 - non-obvious dependencies, 348
 - organization to show dependencies, 348
 - parameters, effective, 349
 - proximity principle, 351
 - specific order, required, 347–350
 - top to bottom readability guideline, 351–352
- Strategy pattern, 104
- stratification design goal, 81
- strcpy(), 301
- streams, 206
- strength. *See* cohesion
- string data types
 - C language, 299–301
 - character sets, 298
 - checklist, 316–317
 - conversion strategies, 299
 - indexes, 298, 299–300, 627
 - initializing, 300
 - localization, 298
 - magic (literal) strings, 297–298
 - memory concerns, 298, 300
 - pointers vs. character arrays, 299
 - Unicode, 298, 299
- string pointers, 299
- strncpy(), 301
- strong cohesion, 105
- structs. *See* structures
- structured basis testing
 - recommended, 503
 - theory of, 505–509
- structured programming
 - core thesis of, 456
 - iteration, 456
 - overview, 454
 - selections, 455
 - sequences, 454
- structures
 - blocks of data, operations on, 320–322
 - checklist for, 343
 - clarifying data relationships with, 320
 - classes performing as, 319
 - defined, 319
 - key points, 344
 - maintenance reduction with, 323
 - overdoing, 322
 - parameter simplification with, 322
 - relationships, clear example of, 320
 - routine calls with, 322
 - simplifying data operations with, 320–322
 - swapping data, 321–322
 - unstructured data example, 320
 - Visual Basic examples, 320–322
- stub objects, testing with, 523
- stubs as integration aids, 694, 696
- stubs with debugging aids, 208–209
- style issues
 - formatting. *See* layout
 - self-documenting code, 778–781
 - human aspects of, 683–684
- sub procedures, 161. *See also* routines
- subsystem design level, 82–85
- subtraction, 295
- swapping data using structures, 321–322
- switch statements. *See* case statements
- symbolic debuggers, 526–527
- syntax, errors in, 549–550, 560, 713–714
- system architecture. *See* architecture
- system calls
 - code tuning, 633–634
 - performance issues, 599–600
- system dependencies, 85
- system perturbers, 527
- system testing, 500
- system-level refactoring, 576–577, 579

T

- table-driven methods
 - advantages of, 420
 - binary searches with, 428
 - case statement approach, 421–422
 - checklist, 429
 - code-tuning with, 614–615
 - creating from expressions, 435
 - days-in-month example, 413–414
 - defined, 411
 - design method, 420
 - direct access. *See* direct access tables
 - endpoints of ranges, 428
 - flexible-message-format example, 416–423
 - fudging keys for, 423–424
 - indexed access tables, 425–426, 428–429
 - insurance rates example, 415–416
 - issues in, 412–413
 - key points, 430
 - keys for, 423–424
 - lookup issue, 412
 - miscellaneous examples, 429
 - object approach, 422–423
 - precomputing calculations, 635
 - purpose of, 411–412
 - stair-step access tables, 426–429
 - storage issue, 413
 - transforming keys, 424
- Tacoma Narrows bridge, 74
- takedown code, refactoring, 568–569
- Team Software Process (TSP), 521

- teams. *See also* managing
 - construction
 - build groups, 704
 - checklist, 69
 - development processes used by, 840
 - expanding to meet schedules, 676
 - managers, 686
 - physical environment, 684–685
 - privacy of offices, 684
 - process, importance to, 839–840
 - religious issues, 683–684
 - resources on, 685–686
 - size of projects, effects of, 650–653
 - style issues, 683–684
 - time allocations, 681
 - variations in performance, 681–683
 - technology waves, determining your
 - location in, 66–69
 - Template Method pattern, 104
 - template tools, 713
 - temporal cohesion, 169
 - temporary variables, 267–268
 - testability
 - defined, 465
 - strategies for, 467
 - test-data generators, 524–525
 - test-first development, 233
 - testing
 - automated testing, 528–529
 - bad data classes, 514–515
 - black-box testing, 500
 - boundary analysis, 513–514
 - bounds checking tools, 527
 - cases, creating, 506–508, 522–525, 532
 - characteristics of, troublesome, 501
 - checklist, 532
 - classes prone to error, 517–518
 - classifications of errors, 518–520
 - clean test limitation, 504
 - clerical errors (typos), 519
 - code coverage testing, 506
 - component testing, 499
 - compound boundaries, 514
 - construction defects, proportion of, 520–521
 - coverage of code, 505–509, 526
 - data flow testing, 509–512
 - data generators for, 524–525
 - data recorder tools, 526
 - debuggers, 526–527
 - debugging, compared to, 500
 - defined-used data paths, 510–512
 - design concerns, 503
 - designs, misunderstanding, 519
 - developer-view limitations, 504
 - developing tests, 522
 - diff tools for, 524
 - driver routines, 523
 - dummy classes, 523
 - dummy files for, 524
 - during construction, 502–503
 - ease of fixing defects, 519
 - equivalence partitioning, 512
 - error checklists for, 503
 - error databases, 527
 - error guessing, 513
 - error presence assumption, 501
 - errors in testing itself, 522
 - expected defect rate, 521–522
 - first or last recommendation, 503–504, 531
 - frameworks for, 522, 524
 - goals of, 501
 - good data classes, 515–516
 - integration testing, 499
 - JUnit for, 531
 - key points, 533
 - limitations on developer testing, 504
 - logging tools for, 526
 - logic coverage testing, 506
 - maximum normal configurations, 515
 - measurement of, 520, 529
 - memory tools, 527
 - minimum normal configurations, 515
 - mock objects, 523
 - nominal case errors, 515
 - old data, compatibility with, 516
 - optimistic programmers
 - limitation, 504
 - outside of construction domain
 - defects, 519
 - planning for, 528
 - prioritizing coverage, 505
 - provability of correctness, 501, 505
 - quality not affected by, 501
 - random-data generators, 525
 - recommended approach to, 503–504
 - record keeping for, 529–530
 - regression testing, 500, 528
 - requirements, 503
 - resources for, 530–531
 - results, uses for, 502
 - role in software quality assurance, 500–502
 - routines, black-box testing of, 502
 - scaffolding, 523–524, 531
 - scope of defects, 519
 - selecting cases for convenience, 516
 - stabilizing errors, 542
 - standards, IEEE, 532
 - structured basis testing, 503, 505–509
 - stub objects, 523
 - symbolic debuggers, 526–527
 - system perturbers, 527
 - system testing, 500
 - testability, 465, 467
 - test case errors, 522
 - time commitment to, 501–502
 - test-first development, 233
 - tools, list of, 719
 - unit testing, 499, 545
 - varying cases, 545
 - white-box testing, 500, 502
 - threading, 337
 - throwaway code, 114
 - throwing one away metaphor, 13–14
 - time allowances, 55–56
 - tool version control, 668
 - toolbox approach, 20
 - tools
 - checklist, 70
 - debugging. *See* debugging
 - editing. *See* editing tools
 - programming. *See* programming tools
 - source code. *See* source-code tools
 - top-down approach to design, 111–113
 - top-down integration, 694–696
 - transcendental functions, 602, 634
 - translator tools, 715
 - try-finally statements, 404–405
 - T-shaped integration, 701
 - type casting, avoiding, 334
 - type creation
 - C++, 312
 - centralization benefit, 314
 - checklist, 318
 - classes, compared to, 316
 - example of, 313–315
 - guidelines for, 315–316
 - information hiding aspect of, 313–314

- 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

- 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

Steve McConnell

Steve McConnell is Chief Software Engineer at Construx Software where he oversees Construx's software engineering practices. Steve is the lead for the Construction Knowledge Area of the Software Engineering Body of Knowledge (SWEBOK) project. Steve has worked on software projects at Microsoft, Boeing, and other Seattle-area companies.

Steve is the author of *Rapid Development* (1996), *Software Project Survival Guide* (1998), and *Professional Software Development* (2004). His books have twice won *Software Development* magazine's Jolt Excellence award for outstanding software development book of the year. Steve was also the lead developer of SPC Estimate Professional, winner of a Software Development Productivity award. In 1998, readers of *Software Development* magazine named Steve one of the three most influential people in the software industry, along with Bill Gates and Linus Torvalds.

Steve earned a Bachelor's degree from Whitman College and a Master's degree in software engineering from Seattle University. He lives in Bellevue, Washington.

If you have any comments or questions about this book, please contact Steve at stevemcc@construx.com or via www.stevemccconnell.com.

