

Table of Contents

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

Part I Laying the Foundation

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

**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

xii	Table of Contents	Copyrighted Material
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

xvi	Table of Contents	Copyrighted Material
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/

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

Copyrighted Material

- 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

Copyrighted Material

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

Copyrighted Material

- Figure 5-2** The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5). 82
- Figure 5-3** An example of a system with six subsystems. 83
- Figure 5-4** An example of what happens with no restrictions on intersubsystem communications. 83
- Figure 5-5** With a few communication rules, you can simplify subsystem interactions significantly. 84
- Figure 5-6** This billing system is composed of four major objects. The objects have been simplified for this example. 88
- Figure 5-7** Abstraction allows you to take a simpler view of a complex concept. 90
- Figure 5-8** Encapsulation says that, not only are you allowed to take a simpler view of a complex concept, you are not allowed to look at any of the details of the complex concept. What you see is what you get—it's all you get! 91
- Figure 5-9** A good class interface is like the tip of an iceberg, leaving most of the class unexposed. 93
- Figure 5-10** G. Polya developed an approach to problem solving in mathematics that's also useful in solving problems in software design (Polya 1957). 109
- Figure 8-1** Part of the Interstate-90 floating bridge in Seattle sank during a storm because the flotation tanks were left uncovered, they filled with water, and the bridge became too heavy to float. During construction, protecting yourself against the small stuff matters more than you might think. 189
- Figure 8-2** Defining some parts of the software that work with dirty data and some that work with clean data can be an effective way to relieve the majority of the code of the responsibility for checking for bad data. 204
- Figure 9-1** Details of class construction vary, but the activities generally occur in the order shown here. 216
- Figure 9-2** These are the major activities that go into constructing a routine. They're usually performed in the order shown. 217
- Figure 9-3** You'll perform all of these steps as you design a routine but not necessarily in any particular order. 225
- Figure 10-1** "Long live time" means that a variable is live over the course of many statements. "Short live time" means it's live for only a few statements. "Span" refers to how close together the references to a variable are. 246
- Figure 10-2** Sequential data is data that's handled in a defined order. 254
- Figure 10-3** Selective data allows you to use one piece or the other, but not both. 255

Copyrighted Material

- 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

Copyrighted Material

- Figure 24-2** Your code doesn't have to be messy just because the real world is messy. Conceive your system as a combination of ideal code, interfaces from the ideal code to the messy real world, and the messy real world. 583
- Figure 24-3** One strategy for improving production code is to refactor poorly written legacy code as you touch it, so as to move it to the other side of the "interface to the messy real world." 584
- Figure 27-1** The number of communication paths increases proportionate to the square of the number of people on the team. 650
- Figure 27-2** As project size increases, errors usually come more from requirements and design. Sometimes they still come primarily from construction (Boehm 1981, Grady 1987, Jones 1998). 652
- Figure 27-3** Construction activities dominate small projects. Larger projects require more architecture, integration work, and system testing to succeed. Requirements work is not shown on this diagram because requirements effort is not as directly a function of program size as other activities are (Albrecht 1979; Glass 1982; Boehm, Gray, and Seewaldt 1984; Boddie 1987; Card 1987; McGarry, Waligora, and McDermott 1989; Brooks 1995; Jones 1998; Jones 2000; Boehm et al. 2000). 654
- Figure 27-4** The amount of software construction work is a near-linear function of project size. Other kinds of work increase nonlinearly as project size increases. 655
- Figure 28-1** This chapter covers the software-management topics related to construction. 661
- Figure 28-2** Estimates created early in a project are inherently inaccurate. As the project progresses, estimates can become more accurate. Reestimate periodically throughout a project, and use what you learn during each activity to improve your estimate for the next activity. 673
- Figure 29-1** The football stadium add-on at the University of Washington collapsed because it wasn't strong enough to support itself during construction. It likely would have been strong enough when completed, but it was constructed in the wrong order—an integration error. 690
- Figure 29-2** Phased integration is also called "big bang" integration for a good reason! 691
- Figure 29-3** Incremental integration helps a project build momentum, like a snowball going down a hill. 692

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

Part I

Laying the Foundation

In this part:

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

1

2

Preface

3

4

5

6

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.

7

—Fred Brooks

8

9

10

11

12

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.

13

14

15

16

17

18

19

20

21

22

23

24

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

25

Who Should Read This Book?

26

27

28

29

30

31

32

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.

33 **Experienced Programmers**

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

39 **Self-Taught Programmers**

40 If you haven't had much formal training, you're in good company. About 50,000
41 new programmers enter the profession each year (BLS 2002), but only about
42 35,000 software-related degrees are awarded each year (NCES 2002). From
43 these figures it's a short hop to the conclusion that most programmers don't
44 receive a formal education in software development. Many self-taught
45 programmers are found in the emerging group of professionals—engineers,
46 accountants, teachers, scientists, and small-business owners—who program as
47 part of their jobs but who do not necessarily view themselves as programmers.
48 Regardless of the extent of your programming education, this handbook can give
49 you insight into effective programming practices.

50 **Students**

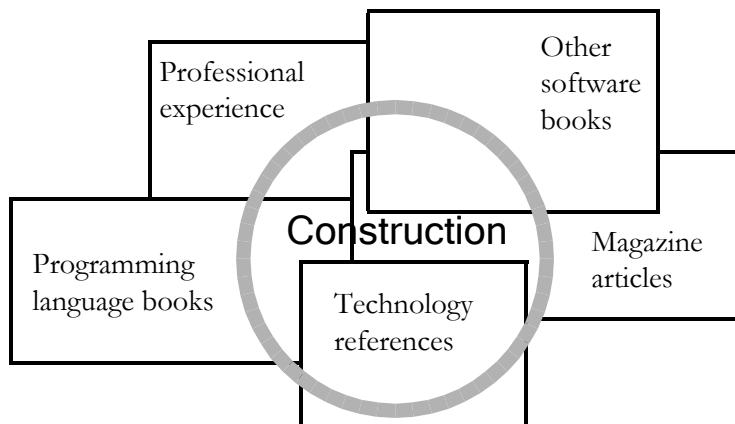
51 The counterpoint to the programmer with experience but little formal training is
52 the fresh college graduate. The recent graduate is often rich in theoretical
53 knowledge but poor in the practical know-how that goes into building production
54 programs. The practical lore of good coding is often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

62 **Where Else Can You Find This Information?**

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

69 share what they have learned. Consequently, programmers may have difficulty
70 finding a good source of programming information.

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



F00xx01

Figure 1

The information in this book is distilled from many sources.

82
83
84
85
86 The only other way to obtain the information you'll find in this handbook would
87 be to plow through a mountain of books and a few hundred technical journals
88 and then add a significant amount of real-world experience. If you've already
89 done all that, you can still benefit from this book's collecting the information in
90 one place for easy reference.

Key Benefits of This Handbook

91
92 Whatever your background, this handbook can help you write better programs in
93 less time and with fewer headaches.

94 ***Complete software-construction reference***

95 This handbook discusses general aspects of construction such as software quality
96 and ways to think about programming. It gets into nitty-gritty construction
97 details such as steps in building classes, ins and outs of using data and control
98 structures, debugging, refactoring, and code-tuning techniques and strategies.

99 You don't need to read it cover to cover to learn about these topics. The book is
100 designed to make it easy to find the specific information that interests you.

101 ***Ready-to-use checklists***

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

105 ***State-of-the-art information***

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

110 ***Larger perspective on software development***

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

118 ***Absence of hype***

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

124 ***Concepts applicable to most common languages***

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

128 ***Numerous code examples***

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

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

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

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

148 **Why This Handbook Was Written**

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

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

159 **The Topic of Construction Has Been Neglected**

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

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

173 **Construction Is Important**

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

178 Construction typically makes up about 80 percent of the effort on small projects
179 and 50 percent on medium projects. Construction accounts for about 75 percent
180 of the errors on small projects and 50 to 75 percent on medium and large
181 projects. Any activity that accounts for 50 to 75 percent of the errors presents a
182 clear opportunity for improvement. (Chapter 27 contains more details on this
183 topic.)

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

192 Small-scale coding errors might be less expensive to fix than errors in
193 requirements or architecture, but an inexpensive cost to fix obviously does not
194 imply that fixing them should be a low priority.

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

201

202 *When art critics get
203 together they talk about
204 Form and Structure and
205 Meaning. When artists
206 get together they talk
207 about where you can buy
208 cheap turpentine.*

209 —Pablo Picasso

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224 CC2E.COM/1234

225

226

227

228

229

230

231

232

233

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 ago or more and employed relatively esoteric languages such as ALGOL, PL/I, Ratfor, and Smalltalk. Some were written by professors who were not working on production code. The professors wrote about techniques that worked for student projects, but they often had little idea of how the techniques would play out in full-scale development environments. Still other books trumpeted the authors' newest favorite methodologies but ignored the huge repository of mature practices that have proven their effectiveness over time.

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

Book Website

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

Author Note

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

*Bellevue, Washington
New Year's Day, 2004*

Notes about the Second Edition

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

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

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

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

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

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

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

34 Does the second edition discuss object-oriented programming?

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

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

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

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

55 *CC1* addressed many topics that would be called “agile” today. For example,
56 here’s what I said about planning in the first edition:

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

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

65 *“Evolution during development is an issue that hasn’t
66 received much attention in its own right. With the rise of code-
67 centered approaches such as prototyping and evolutionary
68 delivery, it’s likely to receive an increasing amount of
69 attention.”*

70
71
72
73
74

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

75
76

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

77
78
79
80
81
82
83
84
85
86
87

What size project will benefit from Code Complete, Second Edition?

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

88
89
90

Have there been any improvements in programming in the past 10 years?

Programming tools have advanced by leaps and bounds. The tool that I described as a panacea in 1993 is commonplace today.

91
92
93
94

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

95
96
97
98
99
100

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

101
102
103
104
105
106

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

107 *Has anything moved backwards?*

108
109
110
111
112
113
114
115
116
There are still far more people who talk about good practices than who actually use good practices. I see far too many people using current buzzwords as a cloak for sloppy practices. When the first edition was published, people were claiming, “I don’t have to do requirements or design because I’m using object-oriented programming.” That was just an excuse. Most of those people weren’t really doing object-oriented programming—they were hacking, and the results were predictable, and poor. Right now, people are saying “I don’t have to do requirements or design because I’m doing agile development.” Again, the results are easy to predict, and poor.

117
118
119
120
121
122
Testing guru Boris Beizer said that his clients ask him, “How can I revolutionize and transform my software development without changing anything except the names and putting some slogans up on the walls?” (Johnson 1994b). Good programmers invest the effort to learn how to use current practices. Not-so-good programmers just learn the buzzwords, and that’s been a software industry constant for a half century.

123 *Which of the first edition’s ideas are you most protective of?*

124
125
126
127
I’m protective of the construction metaphor and the toolbox metaphor. Some writers have criticized the construction metaphor as not being well-suited to software, but most of those writers seem to have simplistic understandings of construction (You can see how I’ve responded to those criticisms in Chapter 2.)

128
129
130
131
132
133
134
135
136
The toolbox metaphor is becoming more critical as software continues to weave itself into every fiber of our lives. Understanding that different tools will work best for different kinds of jobs is critical to not using an axe to cut a stick of butter and not using a butter knife to chop down a tree. It’s silly to hear people criticize software axes for being too bureaucratic when they should have chosen butter knives instead. Axes are good, and so are butter knives, but you need to know what each is used for. In software, we still see people using practices that are good practices in the right context but that are not well suited for every single task.

137 *Will there be a third edition 10 years from now?*

138
I’m tired of answering questions. Let’s get on with the book!

1

Welcome to Software Construction

4 CC2E.COM/0178

Contents

- 5 1.1 What Is Software Construction?
- 6 1.2 Why Is Software Construction Important?
- 7 1.3 How to Read This Book

8 Related Topics

9 Who should read the book: Preface

10 Benefits of reading the book: Preface

11 Why the book was written: Preface

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

19 1.1 What Is Software Construction?

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

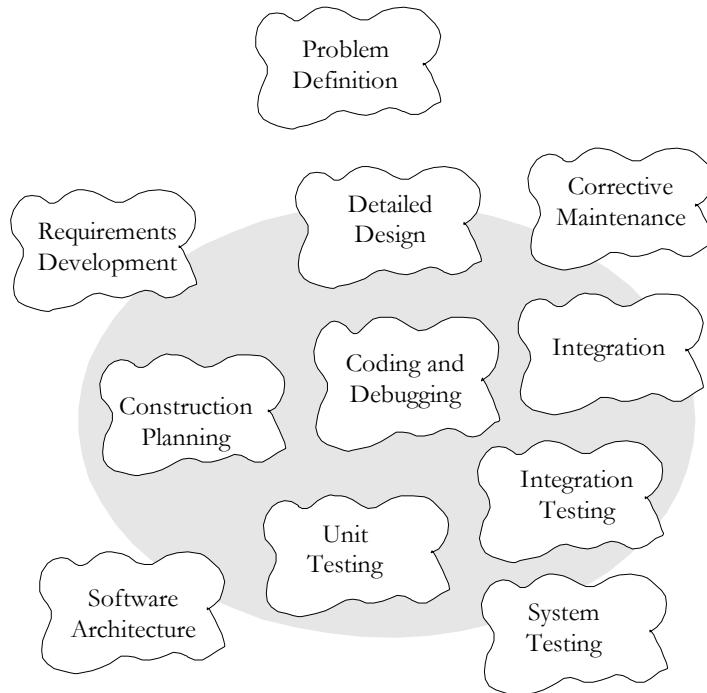
- 23 • Problem definition
- 24 • Requirements development
- 25 • Construction planning
- 26 • Software architecture, or high-level design

- 27 • Detailed design
28 • Coding and debugging
29 • Unit testing
30 • Integration testing
31 • Integration
32 • System testing
33 • Corrective maintenance

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

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

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



49

50

51

52

53

54

F01xx01

Figure 1-1

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

55 | **KEY POINT**

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

63

64

65

66

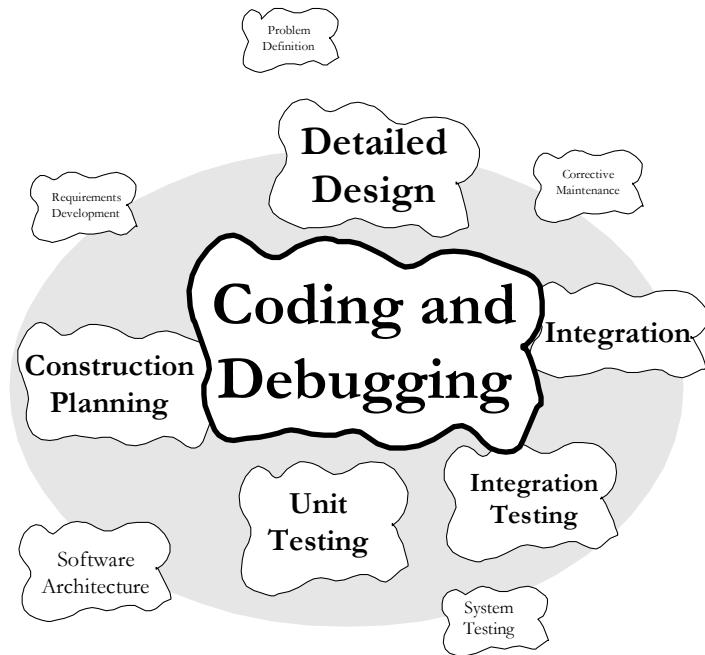
67

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

68

69

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

**F01xx02****Figure 1-2**

This book focuses on detailed design, coding, debugging, and unit testing in roughly these proportions.

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

- 78 ● Verifying that the groundwork has been laid so that construction can proceed
79 successfully
- 80 ● Determining how your code will be tested
- 81 ● Designing and writing classes and routines
- 82 ● Creating and naming variables and named constants
- 83 ● Selecting control structures and organizing blocks of statements
- 84 ● Unit testing, integration testing, and debugging your own code
- 85 ● Reviewing other team members' low-level designs and code and having
86 them review yours
- 87 ● Polishing code by carefully formatting and commenting it
- 88 ● Integrating software components that were created separately
- 89 ● Tuning code to make it smaller and faster

70
71
72
73
74

75
76
77

78
79
80
81
82
83
84
85
86
87
88
89

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

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

102 103 **1.2 Why Is Software Construction Important?**

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

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

114 Here’s why:

115 **CROSS-REFERENCE** For
116 details on the relationship
117 between project size and the
118 percentage of time consumed
119 by construction, see “Activity
Proportions and Size” in
Section 27.5.

120
121
122
123

Construction is a large part of software development

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

Construction is the central activity in software development

Requirements and architecture are done before construction so that you can do construction effectively. System testing is done after construction to verify that construction has been done correctly. Construction is at the center of the software development process.

124 **CROSS-REFERENCE** For
125 data on variations among
126 programmers, see “Individual
127 Variation” in Section 28.5.

128
129
130
131

132
133
134
135
136
137
138
139
140

141 **KEY POINT**

142
143
144
145
146
147
148
149
150

With a focus on construction, the individual programmer’s productivity can improve enormously

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

Construction’s product, the source code, is often the only accurate description of the software

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

Construction is the only activity that’s guaranteed to be done

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

1.3 How to Read This Book

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

159

160

161

162

163

164

165

166

167

168

169

170

171

Key Points

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

2

Metaphors for a Richer Understanding of Software Development

CC2E.COM/0278

Contents

- 2.1 The Importance of Metaphors
- 2.2 How to Use Software Metaphors
- 2.3 Common Software Metaphors

Related Topic

Heuristics in design: “Design is a Heuristic Process” in Section 5.1.

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

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

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

2.1 The Importance of Metaphors

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

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

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

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

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

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

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

57 The suggestive powers of the two models are quite different. The Aristotelian
58 who saw the swinging stone as an object falling would observe the stone’s
59 weight, the height to which it had been raised, and the time it took to come to
60 rest. For Galileo’s pendulum model, the prominent factors were different.
61 Galileo observed the stone’s weight, the radius of the pendulum’s swing, the
62 angular displacement, and the time per swing. Galileo discovered laws the

63 Aristotelians could not discover because their model led them to look at different
64 phenomena and ask different questions.

65 Metaphors contribute to a greater understanding of software-development issues
66 in the same way that they contribute to a greater understanding of scientific
67 questions. In his 1973 Turing Award lecture, Charles Bachman described the
68 change from the prevailing earth-centered view of the universe to a sun-centered
69 view. Ptolemy's earth-centered model had lasted without serious challenge for
70 1400 years. Then in 1543, Copernicus introduced a heliocentric theory, the idea
71 that the sun rather than the earth was the center of the universe. This change in
72 mental models led ultimately to the discovery of new planets, the reclassification
73 of the moon as a satellite rather than a planet, and a different understanding of
74 humankind's place in the universe.

75 ***The value of metaphors***
76 ***should not be***
77 ***underestimated.***
78 ***Metaphors have the***
79 ***virtue of an expected***
80 ***behavior that is***
81 ***understood by all.***
82 ***Unnecessary***
83 ***communication and***
84 ***misunderstandings are***
85 ***reduced. Learning and***
86 ***education are quicker. In***
87 ***effect, metaphors are a***
88 ***way of internalizing and***
89 ***abstracting concepts***
90 ***allowing one's thinking***
91 ***to be on a higher plane***
92 ***and low-level mistakes to***
93 ***be avoided.***

— Fernando J. Corbató

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

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

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

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

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

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

109 2.2 How to Use Software Metaphors

110 KEY POINT

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

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

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

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

128 **CROSS-REFERENCE** For
129 details on how to use
130 heuristics in designing
131 software, see "Design is a
Heuristic Process" in Section
5.1.

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

133 The difference between an algorithm and a heuristic is subtle, and the two terms
134 overlap somewhat. For the purposes of this book, the main difference between
the two is the level of indirection from the solution. An algorithm gives you the

135 instructions directly. A heuristic tells you how to discover the instructions for
136 yourself, or at least where to look for them.

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

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

154 2.3 Common Software Metaphors

155 A confusing abundance of metaphors has grown up around software
156 development. Fred Brooks says that writing software is like farming, hunting
157 werewolves, or drowning with dinosaurs in a tar pit (1995). David Gries says it's
158 a science (1981). Donald Knuth says it's an art (1998). Watts Humphrey says it's
159 a process (1989). P.J. Plauger and Kent Beck say it's like driving a car (Plauger
160 1993, Beck 2000). Alistair Cockburn says it's a game (2001). Eric Raymond
161 says it's like a bazaar (2000). Paul Heckel says it's like filming Snow White and
162 the Seven Dwarfs (1994). Which are the best metaphors?

163 Software Penmanship: Writing Code

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

169 Many ideas derive from the writing metaphor. Jon Bentley says you should be
170 able to sit down by the fire with a glass of brandy, a good cigar, and your

171 favorite hunting dog to enjoy a “literate program” the way you would a good
172 novel. Brian Kernighan and P. J. Plauger named their programming-style book
173 *The Elements of Programming Style* (1978) after the writing-style book *The*
174 *Elements of Style* (Strunk and White 2000). Programmers often talk about
175 “program readability.”

176 **KEY POINT**

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

190 **Plan to throw one away;**
191 **you will, anyhow.**

192 — Fred Brooks

193 **If you plan to throw one**
194 **away, you will throw**
195 **away two.**

196 — Craig Zerouni

197

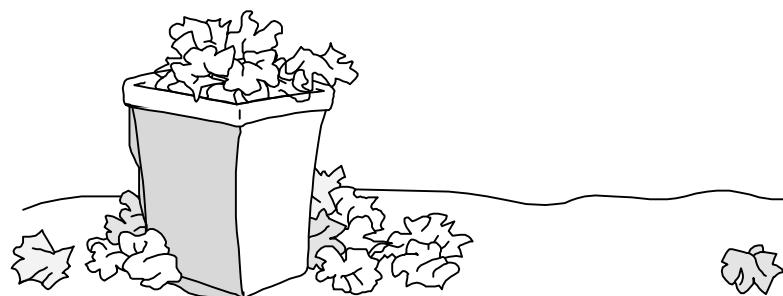
198

199

200

201

Unfortunately, the letter-writing metaphor has been perpetuated by one of the
most popular software books on the planet, Fred Brooks’s *The Mythical Man-
Month* (Brooks 1995). Brooks says, “Plan to throw one away; you will,
anyhow.” This conjures up an image of a pile of half-written drafts thrown into a
wastebasket. Planning to throw one away might be practical when you’re writing
a polite how-do-you-do to your aunt, and it might have been state-of-the-art
software engineering practice in 1975, when Brooks wrote his book.



F02xx01

Figure 2-1

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

202 But extending the metaphor of “writing” software to a plan to throw one away is
203 poor advice for software development in the twenty-first century, when a major
204 system already costs as much as a 10-story office building or an ocean liner. It’s
205 easy to grab the brass ring if you can afford to sit on your favorite wooden pony
206 for an unlimited number of spins around the carousel. The trick is to get it the
207 first time around—or to take several chances when they’re cheapest. Other
208 metaphors better illuminate ways of attaining such goals.

209 **Software Farming: Growing a System**

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

215 **KEY POINT**

216
217

218 **FURTHER READING** For an
219 illustration of a different
220 farming metaphor, one that’s
221 applied to software
222 maintenance, see the chapter
223 “On the Origins of Designer
224 Intuition” in *Rethinking
225 Systems Analysis and Design*
(Weinberg 1988).

226

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

227
228
229
230

**F02xx02****Figure 2-2**

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

Software Oyster Farming: System Accretion

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

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

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

After you've formed the skeleton, little by little you lay on the muscle and skin. You change each of the dummy classes to real classes. Instead of having your program pretend to accept input, you drop in code that accepts real input. Instead of having your program pretend to produce output, you drop in code that

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245 **CROSS-REFERENCE** For
246 details on how to apply
247 incremental strategies to
248 system integration, see
249 Section 29.2, "Integration
250 Frequency—Phased or
Incremental?"

251

252

253

254

255

256

257

258

259

260

261

262 produces real output. You add a little bit of code at a time until you have a fully
263 working system.

264 The anecdotal evidence in favor of this approach is impressive. Fred Brooks,
265 who in 1975 advised building one to throw away, said that nothing in the decade
266 after he wrote his landmark book *The Mythical Man-Month* so radically changed
267 his own practice or its effectiveness as incremental development (1995). Tom
268 Gilb made the same point in his breakthrough book *Principles of Software*
269 *Engineering Management* (1988), which introduced Evolutionary Delivery and
270 laid the groundwork for much of today's Agile programming approach.
271 Numerous current methodologies are based on this idea (Beck 2000, Cockburn
272 2001, Highsmith 2002, Reifer 2002, Martin 2003, Larman 2004).

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

277 **Software Construction: Building Software**

278 **KEY POINT**

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

284 Building a 4-foot tower requires a steady hand, a level surface, and 10
285 undamaged beer cans. Building a tower 100 times that size doesn't merely
286 require 100 times as many beer cans. It requires a different kind of planning and
287 construction altogether.

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

**F02xx03****Figure 2-3**

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

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

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

295

296

297

298

299

300

301

302

303

304

305

306

307

308

309

310

311

312

313

314

315

316

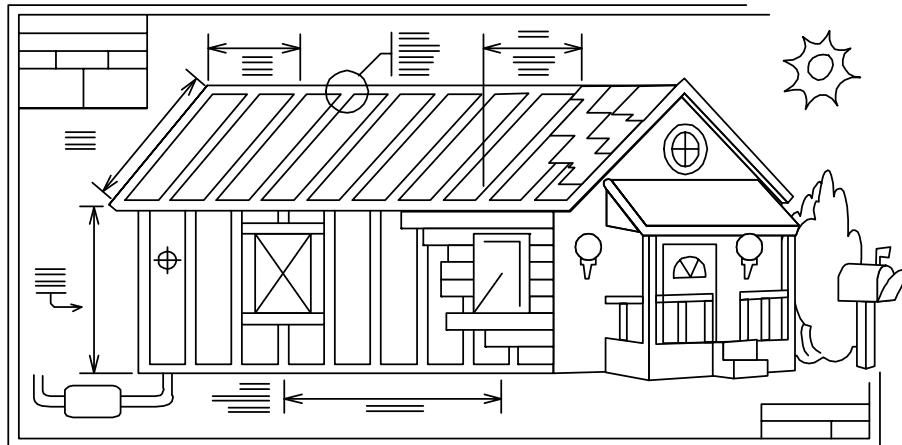
317

318

319

320

321

**F02xx04****Figure 2-4**

More complicated structures require more careful planning.

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

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

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

322
323
324
325

326
327
328
329
330
331
332
333
334
335

336
337
338
339
340
341
342
343

344
345
346
347
348

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

357 The construction analogy also helps explain why different software projects
358 benefit from different development approaches. In building, you'd use different
359 levels of planning, design, and quality assurance if you're building a warehouse
360 or a shopping mall than if you're building a medical center or a nuclear reactor.
361 You'd use still different approaches for building a school, a skyscraper, or a
362 three bedroom home. Likewise, in software you might generally use flexible,
363 lightweight development approaches, but sometimes rigid, heavyweight
364 approaches are required to achieve safety goals and other goals.

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

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

377 Likewise, for extremely large software projects, planning of a higher order is
378 needed than for projects that are merely large. Capers Jones reports that a one-
379 million line of code software system requires an average of 69 *kinds* of
380 documentation (1998). The requirements specification for a 1,000,000 line of
381 code system would typically be about 4,000-5,000 pages long, and the design
382 documentation can easily be two or three times as extensive as the requirements.
383 It's unlikely that an individual would be able to understand the complete design
384 for a project of this size—or even read it. A greater degree of preparation is
385 appropriate.

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

388 **FURTHER READING** For
389 some good comments about
390 extending the construction
391 metaphor, see “What
392 Supports the Roof?” (Starr
392 2003).

393

394

395 **KEY POINT**

396

397

398

399

400

401

402 **CROSS-REFERENCE** For
403 details on selecting and
404 combining methods in
405 design, see Section 5.3,
406 “Design Building Blocks:
406 Heuristics.”

407

408

409

410

411

412

413

414

415

416

417

418

419

CC2E.COM/0285

420

421

422

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

Applying Software Techniques: The Intellectual Toolbox

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

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

Combining Metaphors

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

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

Additional Resources

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

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

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

433 Key Points

- 434 • Metaphors are heuristics, not algorithms. As such, they tend to be a little
435 sloppy.
- 436 • Metaphors help you understand the software-development process by
437 relating it to other activities you already know about.
- 438 • Some metaphors are better than others.
- 439 • Treating software construction as similar to building construction suggests
440 that careful preparation is needed and illuminates the difference between
441 large and small projects.
- 442 • Thinking of software-development practices as tools in an intellectual
443 toolbox suggests further that every programmer has many tools and that no
444 single tool is right for every job. Choosing the right tool for each problem is
445 one key to being an effective programmer.

3

Measure Twice, Cut Once: Upstream Prerequisites

CC2E.COM/0309

Contents

- 3.1 Importance of Prerequisites
- 3.2 Determine the Kind of Software You're Working On
- 3.3 Problem-Definition Prerequisite
- 3.4 Requirements Prerequisite
- 3.5 Architecture Prerequisite
- 3.6 Amount of Time to Spend on Upstream Prerequisites

Related Topics

- Key construction decisions: Chapter 4
 - Effect of project size on construction and prerequisites: Chapter 27
 - Relationship between quality goals and construction activities: Chapter 20
 - Managing construction: Chapter 28
 - Design: Chapter 5
- Before beginning construction of a house, a builder reviews blueprints, checks that all permits have been obtained, and surveys the house's foundation. A builder prepares for building a skyscraper one way, a housing development a different way, and a doghouse a third way. No matter what the project, the preparation is tailored to the project's specific needs and done conscientiously before construction begins.

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

28 jewel, you have to start with a diamond in the rough. If you start with plans for a
29 brick, the best you can create is a fancy brick.

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

35 Although this chapter lays the groundwork for successful software construction,
36 it doesn’t discuss construction directly. If you’re feeling carnivorous or you’re
37 already well versed in the software-engineering life cycle, look for the construc-
38 tion meat beginning in Chapter 5. If you don’t like the idea of prerequisites to
39 construction, review Section 3.2, “Determine the Kind of Software You’re
40 Working On,” to see how prerequisites apply to your situation, and then take a
41 look at the data in Section 3.1 which describes the cost of not doing prerequi-
42 sites.

43 3.1 Importance of Prerequisites

44 **CROSS-REFERENCE** Pay-
45 ing attention to quality is also
46 the best way to improve pro-
ductivity. For details, see
47 Section 20.5, “The General
48 Principle of Software Qual-
ity.”

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

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

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

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

53 KEY POINT

54

55
56
57
58
59
60
61

62
63
64
65
66
67
68

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

69
70

Do Prerequisites Apply to Modern Software Projects?

71 *The methodology used
72 should be based on choice
73 of the latest and best, and
74 not based on ignorance.
75 It should also be laced
76 liberally with the old and
77 dependable.*
78 — Harlan Mills
79

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

KEY POINT

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

Preparation for construction is not an exact science, and the specific approach to risk reduction must be decided project by project. Details can vary greatly among projects. For more on this, see Section 3.2, “Determine the Kind of Software You’re Working On.”

Causes of Incomplete Preparation

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

A common cause of incomplete preparation is that the developers who are assigned to work on the upstream activities do not have the expertise to carry out their assignments. The skills needed to plan a project, create a compelling business case, develop comprehensive and accurate requirements, and create high-quality architectures are far from trivial, but most developers have not received training in how to perform these activities. When developers don't know how to

93 **FURTHER READING** For a
94 description of a professional
95 development program that
96 that cultivates these skills,
97 see Chapter 16 of *Profes-*
98 *sional Software Development*
(McConnell 2004).

99 do upstream work, the recommendation to “do more upstream work” sounds like
100 nonsense: If the work isn’t being done well in the first place, doing *more* of it
101 will not be useful! Explaining how to perform these activities is beyond the
102 scope of this book, but the “Additional Resources” sections at the end of this
103 chapter provide numerous options for gaining that expertise.

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

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

116 **FURTHER READING** For
117 many entertaining variations
118 on this theme, read Gerald
119 Weinberg’s classic, *The Psychology of Computer Pro-*
120 *gramming* (Weinberg 1998).
121
122
123
124
125
126
127

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

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

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

136 Second, you can pretend to be coding when you're not. Put an old program list-
137 ing on the corner of your desk. Then go right ahead and develop your require-
138 ments and architecture, with or without your boss's approval. You'll do the pro-
139 ject faster and with higher-quality results. From your boss's perspective, igno-
140 rance is bliss.

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

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

149

150 **Utterly Compelling and Foolproof Argument for Doing Prerequisites Before Construction**

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

156 KEY POINT

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

163

164 **Appeal to Logic**

165 One of the key ideas in effective programming is that preparation is important. It
166 makes sense that before you start working on a big project, you should plan the
167 project. Big projects require more planning; small projects require less. From a
168 management point of view, planning means determining the amount of time,
169 number of people, and number of computers the project will need. From a tech-
170 nical point of view, planning means understanding what you want to build so
171 that you don't waste money building the wrong thing. Sometimes users aren't
entirely sure what they want at first, so it might take more effort than seems ideal

172 to find out what they really want. But that's cheaper than building the wrong
173 thing, throwing it away, and starting over.

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

177 **Appeal to Analogy**

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

183 You don't start decorating the Christmas tree until you've put it in the stand.
184 You don't start a fire until you've opened the flue. You don't go on a long trip
185 with an empty tank of gas. You don't get dressed before you take a shower, and
186 you don't put your shoes on before your socks. You have to do things in the right
187 order in software too.

188 Programmers are at the end of the software food chain. The architect consumes
189 the requirements; the designer consumes the architecture; and the coder con-
190 sumes the design.

191 Compare the software food chain to a real food chain. In an ecologically sound
192 environment, seagulls eat fresh salmon. That's nourishing to them because the
193 salmon ate fresh herring, and they in turn ate fresh water bugs. The result is a
194 healthy food chain. In programming, if you have healthy food at each stage in
195 the food chain, the result is healthy code written by happy programmers.

196 In a polluted environment, the water bugs have been swimming in nuclear waste.
197 The herring are contaminated by PCBs, and the salmon that eat the herring swam
198 through oil spills. The seagulls are, unfortunately, at the end of the food chain, so
199 they don't eat just the oil in the bad salmon. They also eat the PCBs and the nu-
200 clear waste from the herring and the water bugs. In programming, if your re-
201 quirements are contaminated, they contaminate the architecture, and the architec-
202 ture in turn contaminates construction. This leads to grumpy, malnourished pro-
203 grammers and radioactive, polluted software that's riddled with defects.

204 If you are planning a highly iterative project, you will need to identify the critical
205 requirements and architectural elements that apply to each piece you're con-
206 structing before you begin construction. A builder who is building a housing de-
207 velopment doesn't need to know every detail of every house in the development
208 before beginning construction on the first house. But the builder will survey the

209 site, map out sewer and electrical lines, and so on. If the builder doesn't prepare
210 well, construction may be delayed when a sewer line needs to be dug under a
211 house that's already been constructed.

212 **Appeal to Data**

213 Studies over the last 25 years have proven conclusively that it pays to do things
214 right the first time. Unnecessary changes are expensive.

215 **HARD DATA**
216 Researchers at Hewlett-Packard, IBM, Hughes Aircraft, TRW, and other organizations
217 have found that purging an error by the beginning of construction allows
218 rework to be done 10 to 100 times less expensively than when it's done in the
219 last part of the process, during system test or after release (Fagan 1976; Humphrey,
220 Snyder, and Willis 1991; Leffingwell 1997; Willis et al 1998; Grady 1999; Shull, et al, 2002; Boehm and Turner 2004).

221 In general, the principle is to find an error as close as possible to the time at
222 which it was introduced. The longer the defect stays in the software food chain,
223 the more damage it causes further down the chain. Since requirements are done
224 first, requirements defects have the potential to be in the system longer and to be
225 more expensive. Defects inserted into the software upstream also tend to have
226 broader effects than those inserted further downstream. That also makes early
227 defects more expensive.

228 Table 3-1 shows the relative expense of fixing defects depending on when
229 they're introduced and when they're found.

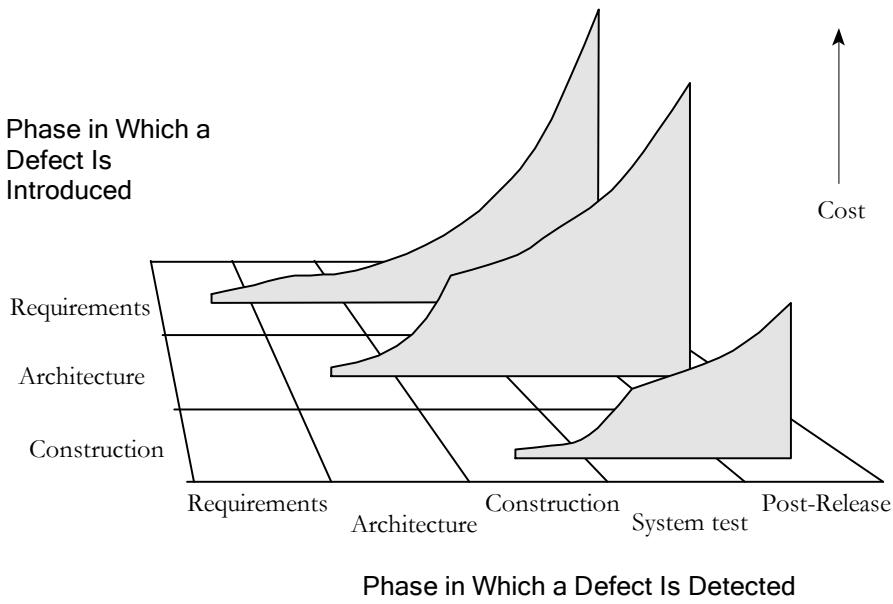
230 **HARD DATA**
231 **Table 3-1. Average Cost of Fixing Defects Based on When They're Introduced and When They're Detected**

Time Introduced	Time Detected				
	Re- quire- ments	Archite- cture	Con- struc- tion	System Test	Post- Re- lease
Requirements	1	3	5-10	10	10-100
Architecture	—	1	10	15	25-100
Construction	—	—	1	10	10-25

232 *Source: Adapted from "Design and Code Inspections to Reduce Errors in Program
233 Development" (Fagan 1976), Software Defect Removal (Dunn 1984), "Software
234 Process Improvement at Hughes Aircraft" (Humphrey, Snyder, and Willis 1991),
235 "Calculating the Return on Investment from More Effective Requirements Manage-
236 ment" (Leffingwell 1997), "Hughes Aircraft's Widespread Deployment of a Con-
237 tinuously Improving Software Process" (Willis et al 1998), "An Economic Release
238 Decision Model: Insights into Software Project Management" (Grady 1999), "What*

239 *We Have Learned About Fighting Defects" (Shull et al 2002), and Balancing Agility*
 240 *and Discipline: A Guide for the Perplexed (Boehm and Turner 2004).*

241 The data in Table 3-1 shows that, for example, an architecture defect that costs
 242 \$1000 to fix when the architecture is being created can cost \$15,000 to fix during
 243 system test. Figure 3-1 illustrates the same phenomenon.



244 **F03xx01**

245 **Figure 3-1**

246 *The cost to fix a defect rises dramatically as the time from when it's introduced to*
 247 *when it's detected increases. This remains true whether the project is highly sequen-*
 248 *tial (doing 100 percent of requirements and design up front) or highly iterative (do-*
 249 *ing 5 percent of requirements and design up front).*

250

251 **HARD DATA**

252 The average project still exerts most of its defect-correction effort on the right
 253 side of Figure 3-1, which means that debugging and associated rework takes
 254 about 50 percent of the time spent in a typical software development cycle (Mills
 255 1983; Boehm 1987a; Cooper and Mullen 1993; Fishman 1996; Haley 1996;
 256 Wheeler, Bryczynski, and Meeson 1996; Jones 1998, Shull et al 2002, Wiegers
 257 2002). Dozens of companies have found that simply focusing on correcting de-
 258 fects earlier rather than later in a project can cut development costs and sched-
 259 ules by factors of two or more (McConnell 2004). This is a healthy incentive to
 fix your problems as early as you can.

260 **Boss-Readiness Test**

261 When you think your boss understands the importance of completing princi-
 262 ples before moving into construction, try the test below to be sure.

- 263 Which of these statements are self-fulfilling prophecies?
- 264 • We'd better start coding right away because we're going to have a lot of
265 debugging to do.
- 266 • We haven't planned much time for testing because we're not going to find
267 many defects.
- 268 • We've investigated requirements and design so much that I can't think of
269 any major problems we'll run into during coding or debugging.

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

271 If you're still not convinced that prerequisites apply to your project, the next sec-
272 tion will help you decide.

273 **3.2 Determine the Kind of Software You're 274 Working On**

275 Capers Jones, Chief Scientist at Software Productivity Research, summarized 20
276 years of software research by pointing out that he and his colleagues have seen
277 40 different methods for gathering requirements, 50 variations in working on
278 software designs, and 30 kinds of testing applied to projects in more than 700
279 different programming languages (Jones 2003).

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

285 **Table 3-2. Typical good practices for three common kinds of software
286 projects**

Kind of Software	Typical Good Practices		
	Business Systems	Mission-Critical Systems	Embedded Life-Critical Systems

Typical Good Practices			
Kind of Software	Business Systems	Mission-Critical Systems	Embedded Life-Critical Systems
Typical applications	Internet site	Embedded software	Avionics software
	Intranet site	Games	Embedded software
	Inventory management	Internet site	Medical devices
	Games	Packaged software	Operating systems
	Management information systems	Software tools	Packaged software
	Payroll system	Web services	
Lifecycle models	Agile development (extreme programming, scrum, time-box development, and so on)	Staged delivery	Staged delivery
		Evolutionary delivery	Spiral development
	Evolutionary prototyping	Spiral development	Evolutionary delivery
Planning and management	Incremental project planning	Basic up-front planning	Extensive up-front planning
	As-needed test and QA planning	Basic test planning	Extensive test planning
	Informal change control	As-needed QA planning	Extensive QA planning
		Formal change control	Rigorous change control
Requirements	Informal requirements specification	Semi-formal requirements specification	Formal requirements specification
		As-needed requirements reviews	Formal requirements inspections
Design	Design and coding are combined	Architectural design	Architectural design
		Informal detailed design	Formal architecture inspections
		As-needed design reviews	Formal detailed design
Construction	Pair programming or individual coding	Pair programming or individual coding	Pair programming or individual coding
	Informal check-in procedure or no check-in procedure	Informal check-in procedure	Formal check-in procedure
		As-needed code reviews	Formal code inspections

Typical Good Practices			
Kind of Software	Business Systems	Mission-Critical Systems	Embedded Life-Critical Systems
Testing and QA	Developers test their own code	Developers test their own code	Developers test their own code
	Test-first development	Test-first development	Test-first development
	Little or no testing by a separate test group	Separate testing group	Separate testing group Separate QA group
Deployment	Informal deployment procedure	Formal deployment procedure	Formal deployment procedure

287

288 On real projects, you'll find infinite variations on the three themes presented in
 289 this table, however the generalities in the table are illuminating. Business sys-
 290 tems projects tend to benefit from highly iterative approaches, in which plan-
 291 ning, requirements, and architecture are interleaved with construction, system
 292 testing and quality assurance activities. Life-critical systems tend to require more
 293 sequential approaches—requirements stability is part of what's needed to ensure
 294 ultra-high levels of reliability.

295 Some writers have asserted that projects that use iterative techniques don't need
 296 to focus on prerequisites much at all, but that point of view is misinformed. Itera-
 297 tive approaches tend to reduce the impact of inadequate upstream work, but they
 298 don't eliminate it. Consider the example shown in Table 3-3 of a project that's
 299 conducted sequentially and that relies solely on testing to discover defects. In
 300 this approach, the defect correction (rework) costs will be clustered at the end of
 301 the project.

302 **Table 3-3. Effect of short-changing prerequisites on sequential and it-
 303 erative projects. This data is for purposes of illustration only**

	Approach #1		Approach #2	
	Sequential Approach without Prerequisites		Iterative Approach without Prerequisites	
Project comple- tion status	Cost of Work	Cost of Rework	Cost of Work	Cost of Rework
10%	\$100,000	\$0	\$100,000	\$75,000
20%	\$100,000	\$0	\$100,000	\$75,000
30%	\$100,000	\$0	\$100,000	\$75,000
40%	\$100,000	\$0	\$100,000	\$75,000

50%	\$100,000	\$0	\$100,000	\$75,000
60%	\$100,000	\$0	\$100,000	\$75,000
70%	\$100,000	\$0	\$100,000	\$75,000
80%	\$100,000	\$0	\$100,000	\$75,000
90%	\$100,000	\$0	\$100,000	\$75,000
100%	\$100,000	\$0	\$100,000	\$75,000
End-of-Project Rework	\$0	\$1,000,000	\$0	\$0
TOTAL	\$1,000,000	\$1,000,000	\$1,000,000	\$750,000
GRAND TOTAL		\$2,000,000		\$1,750,000

304

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

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

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

322 **Table 3-4. Effect of focusing on prerequisites on sequential and iterative projects. This data is for purposes of illustration only**

	Approach #3		Approach #4	
	Sequential Approach with Prerequisites		Iterative Approach with Prerequisites	
Project completion status	Cost of Work	Cost of Rework	Cost of Work	Cost of Rework
10%	\$100,000	\$20,000	\$100,000	\$10,000
20%	\$100,000	\$20,000	\$100,000	\$10,000

30%	\$100,000	\$20,000	\$100,000	\$10,000
40%	\$100,000	\$20,000	\$100,000	\$10,000
50%	\$100,000	\$20,000	\$100,000	\$10,000
60%	\$100,000	\$20,000	\$100,000	\$10,000
70%	\$100,000	\$20,000	\$100,000	\$10,000
80%	\$100,000	\$20,000	\$100,000	\$10,000
90%	\$100,000	\$20,000	\$100,000	\$10,000
100%	\$100,000	\$20,000	\$100,000	\$10,000
End-of-Project Rework	\$0	\$0	\$0	\$0
TOTAL	\$1,000,000	\$200,000	\$1,000,000	\$100,000
GRAND TOTAL		\$1,200,000		\$1,100,000

324 **KEY POINT**

325

326

327

328

329

330

331

332

333

334

335

336

337

338

339

340

341

342

343

344

345

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

One realistic approach is to plan to specify about 80 percent of the requirements up front, allocate time for additional requirements to be specified later, and then practice systematic change control to accept only the most valuable new requirements as the project progresses.

Error! Objects cannot be created from editing field codes.

F03xx02

Figure 3-2

Activities will overlap to some degree on most projects, even those that are highly sequential.

Another alternative is to specify only the most important 20 percent of the requirements up front and plan to develop the rest of the software in small increments, specifying additional requirements and designs as you go.

Error! Objects cannot be created from editing field codes.

F03xx03

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.

346 **CROSS-REFERENCE** For
347 details on how to adapt your
348 development approach for
349 programs of different sizes,
350 see Chapter 27, "How Pro-
gram Size Affects Construc-
tion."

351

352

353

354

355

356

357

358

359

360

361

362

363

364

365

366

367

368

369

370

371

372

373

374

375

376

377

378

379

The extent to which prerequisites need to be satisfied up front will vary with the project type indicated in Table 3-2, project formality, technical environment, staff capabilities, and project business goals. You might choose a more sequential (up-front) approach when:

- The requirements are fairly stable
- The design is straightforward and fairly well understood
- The development team is familiar with the applications area
- The project contains little risk
- Long-term predictability is important
- The cost of changing requirements, design, and code downstream is likely to be high

You might choose a more iterative (as-you-go) approach when:

- The requirements are not well understood or you expect them to be unstable for other reasons
- The design is complex, challenging, or both
- The development team is unfamiliar with the applications area
- The project contains a lot of risk
- Long-term predictability is not important
- The cost of changing requirements, design, and code downstream is likely to be low

You can adapt the prerequisites to your specific project by making them more or less formal and more or less complete, as you see fit. For a detailed discussion of different approaches to large and small projects (also known as the different approaches to formal and informal projects), see Chapter 27, "How Program Size Affects Construction."

The net impact on construction prerequisites is that you should first determine what construction prerequisites are well-suited to your project. Some projects spend too little time on prerequisites, which exposes construction to an unnecessarily high rate of destabilizing changes and prevents the project from making consistent progress. Some project do too much up front; they doggedly adhere to requirements and plans that have been invalidated by downstream discoveries, and that can also impede progress during construction.

Now that you've studied Table 3-2 and determined what prerequisites are appropriate for your project, the rest of this chapter describes how to determine

380
381

382

383 **If the 'box' is the boundary of constraints and
384 conditions, then the trick
385 is to find the box.... Don't
386 think outside the box—
387 find the box."**

388 —Andy Hunt and Dave Thomas

390
391
392
393
394
395
396

397
398

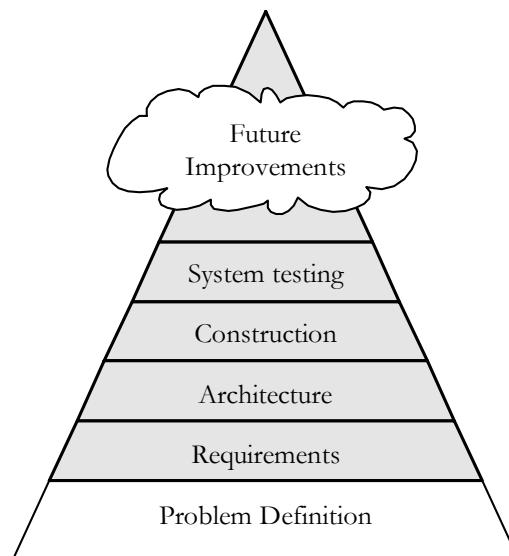
whether each specific construction prerequisite has been “prereq’d” or “pre-wrecked.”

3.3 Problem-Definition Prerequisite

The first prerequisite you need to fulfill before beginning construction is a clear statement of the problem that the system is supposed to solve. This is sometimes called “product vision,” “mission statement,” and “product definition.” Here it’s called “problem definition.” Since this book is about construction, this section doesn’t tell you how to write a problem definition; it tells you how to recognize whether one has been written at all and whether the one that’s written will form a good foundation for construction.

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

Problem definition comes before detailed requirements work, which is a more in-depth investigation of the problem.



F03xx02

Figure 3-2

The problem definition lays the foundation for the rest of the programming process.

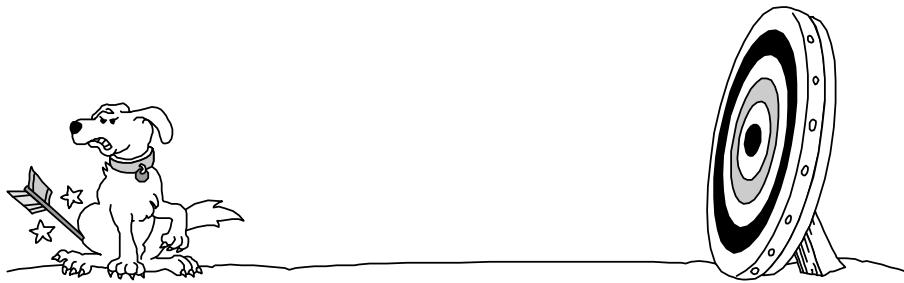
399
400
401
402

403
404
405
406
407
408
409
410
411
412

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

413
414
415

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



416
417
418
419
420

421 **KEY POINT**

422
423

424

425
426
427
428
429

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

3.4 Requirements Prerequisite

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

430
431

Why Have Official Requirements?

An explicit set of requirements is important for several reasons.

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

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

441 **KEY POINT**

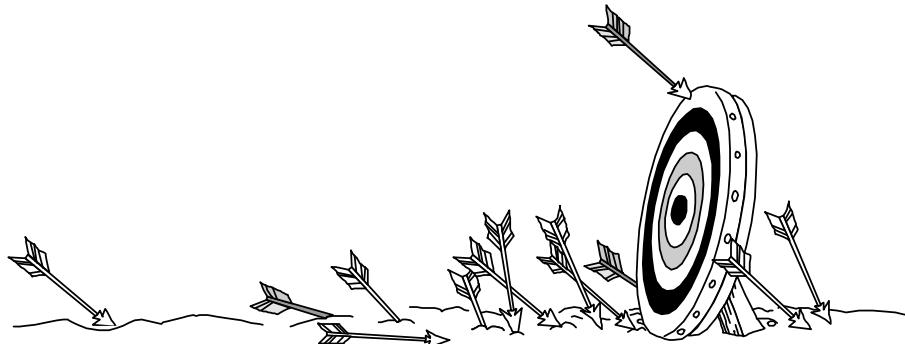
442
443
444
445
446
447
448
449
450

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

451 **HARD DATA**

452
453
454
455
456
457
458
459

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

**F03xx04****Figure 3-4**

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

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

The Myth of Stable Requirements

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

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

How much change is typical? Studies at IBM and other companies have found that the average project experiences about a 25 percent change in requirements during development (Boehm 1981, Jones 1994, Jones 2000), which typically

460

461

462

463

464

465

466

467

468

469

470

471

472 **Requirements are like
473 water. They're easier to
474 build on when they're
475 frozen.**

476 —Anon.
477

478

479

480

481

482

483

484

485

486

487 **HARD DATA**

488

489

490 accounts for 70 to 85 percent of the rework on a typical project (Leffingwell
491 1997, Wiegers 2003).

492 Maybe you think the Pontiac Aztek was the greatest car ever made, belong to the
493 Flat Earth Society, and vote for Ross Perot every four years. If you do, go ahead
494 and believe that requirements won't change on your projects. If, on the other
495 hand, you've stopped believing in Santa Claus and the Tooth Fairy, or at least
496 have stopped admitting it, you can take several steps to minimize the impact of
497 requirements changes.

498 **Handling Requirements Changes During Construc- 499 tion**

500 **KEY POINT**
501 Here are several things you can do to make the best of changing requirements
during construction.

502 ***Use the requirements checklist at the end of the section to assess the quality 503 of your requirements***

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

509 ***Make sure everyone knows the cost of requirements changes***

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

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

523 ***CROSS-REFERENCE*** For 524 details on handling changes 525 to design and code, see Sec- 526 tion 28.2, "Configuration Management."

523 ***Set up a change-control procedure***
If your client's excitement persists, consider establishing a formal change-
control board to review such proposed changes. It's all right for customers to
change their minds and to realize that they need more capabilities. The problem

527
528
529
530
531

532
533 **FURTHER READING** For
534 details on development ap-
535 proaches that support flexible
536 requirements, see *Rapid De-
537 velopment* (McConnell
538 1996).
539

540 **CROSS-REFERENCE** For
541 details on iterative develop-
542 ment approaches, see “Iter-
543 ate” in Section 5.4 and Sec-
544 tion 29.3, “Incremental Inte-
545 gration Strategies.”
546

547 **CROSS-REFERENCE** For
548 details on the differences
549 between formal and informal
550 projects (often caused by
551 differences in project size),
552 see Chapter 27, “How Pro-
553 gram Size Affects Construc-
554 tion.”
555

556
557

558
559
560
561
562
563

is their suggesting changes so frequently that you can’t keep up. Having a built-in procedure for controlling changes makes everyone happy. You’re happy because you know that you’ll have to work with changes only at specific times. Your customers are happy because they know that you have a plan for handling their input.

Use development approaches that accommodate changes

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

Dump the project

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

Checklist: Requirements

The requirements checklist contains a list of questions to ask yourself about your project’s requirements. This book doesn’t tell you how to do good requirements development, and the list won’t tell you how to do one either. Use the list as a sanity check at construction time to determine how solid the ground that you’re standing on is—where you are on the requirements Richter scale.

Not all of the checklist questions will apply to your project. If you’re working on an informal project, you’ll find some that you don’t even need to think about. You’ll find others that you need to think about but don’t need to answer formally. If you’re working on a large, formal project, however, you may need to consider every one.

Specific Functional Requirements

- Are all the inputs to the system specified, including their source, accuracy, range of values, and frequency?
- Are all the outputs from the system specified, including their destination, accuracy, range of values, frequency, and format?
- Are all output formats specified for web pages, reports, and so on?

- 564 Are all the external hardware and software interfaces specified?
565 Are all the external communication interfaces specified, including handshak-
566 ing, error-checking, and communication protocols?
567 Are all the tasks the user wants to perform specified?
568 Is the data used in each task and the data resulting from each task specified?

569 **Specific Non-Functional (Quality) Requirements**

- 570 Is the expected response time, from the user's point of view, specified for all
571 necessary operations?
572 Are other timing considerations specified, such as processing time, data-
573 transfer rate, and system throughput?
574 Is the level of security specified?
575 Is the reliability specified, including the consequences of software failure,
576 the vital information that needs to be protected from failure, and the strategy
577 for error detection and recovery?
578 Is maximum memory specified?
579 Is the maximum storage specified?
580 Is the maintainability of the system specified, including its ability to adapt to
581 changes in specific functionality, changes in the operating environment, and
582 changes in its interfaces with other software?
583 Is the definition of success included? Of failure?

584 **Requirements Quality**

- 585 Are the requirements written in the user's language? Do the users think so?
586 Does each requirement avoid conflicts with other requirements?
587 Are acceptable trade-offs between competing attributes specified—for ex-
588 ample, between robustness and correctness?
589 Do the requirements avoid specifying the design?
590 Are the requirements at a fairly consistent level of detail? Should any re-
591 quirement be specified in more detail? Should any requirement be specified
592 in less detail?
593 Are the requirements clear enough to be turned over to an independent group
594 for construction and still be understood?
595 Is each item relevant to the problem and its solution? Can each item be
596 traced to its origin in the problem environment?
597 Is each requirement testable? Will it be possible for independent testing to
598 determine whether each requirement has been satisfied?
599 Are all possible changes to the requirements specified, including the likeli-
600 hood of each change?

Requirements Completeness

- Where information isn't available before development begins, are the areas of incompleteness specified?
 - Are the requirements complete in the sense that if the product satisfies every requirement, it will be acceptable?
 - Are you comfortable with all the requirements? Have you eliminated requirements that are impossible to implement and included just to appease your customer or your boss?
-

3.5 Architecture Prerequisite

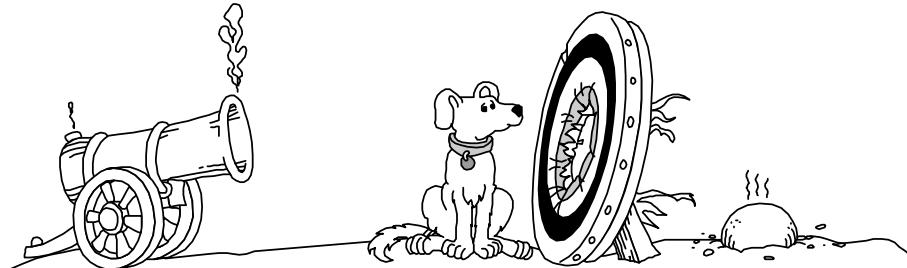
Software architecture is the high-level part of software design, the frame that holds the more detailed parts of the design (Buschman, et al, 1996; Fowler 2002; Bass Clements, Kazman 2003; Clements et al, 2003). Architecture is also known as "system architecture," "high-level design," and "top-level design." Typically, the architecture is described in a single document referred to as the "architecture specification" or "top-level design." Some people make a distinction between architecture and high-level design—architecture refers to design constraints that apply system-wide, whereas high-level design refers to design constraints that apply at the subsystem or multiple-class level, but not necessarily system wide.

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

KEY POINT

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

Good architecture makes construction easy. Bad architecture makes construction almost impossible.

**F03xx05****Figure 3-5**

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

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

Typical Architectural Components

Many components are common to good system architectures. If you’re building the whole system yourself, your work on the architecture, will overlap your work on the more detailed design. In such a case, you should at least think about each architectural component. If you’re working on a system that was architected by someone else, you should be able to find the important components without a bloodhound, a deerstalker cap, and a magnifying glass. In either case, here are the architectural components to consider.

Program Organization

A system architecture first needs an overview that describes the system in broad terms. Without such an overview, you’ll have a hard time building a coherent picture from a thousand details or even a dozen individual classes. If the system were a little 12-piece jigsaw puzzle, your two-year-old could solve it between spoonfuls of strained asparagus. A puzzle of 12 software classes or 12 subsystems is harder to put together, and if you can’t put it together, you won’t understand how a class you’re developing contributes to the system.

In the architecture, you should find evidence that alternatives to the final organization were considered and find the reasons the organization used was chosen over the alternatives. It’s frustrating to work on a class when it seems as if the class’s role in the system has not been clearly conceived. By describing the or-

635
636
637
638
639

640 | **HARD DATA**

641
642
643
644
645
646

647

648 **CROSS-REFERENCE** For
649 details on lower-level pro-
650 gram design, see Chapters 5
through 9.
651

652
653
654

655

656 *If you can’t explain
657 something to a six-year-
658 old, you really don’t un-
659 derstand it yourself. —*
660 *Albert Einstein*

661
662

663
664
665
666

667
668
669
670

671 **CROSS-REFERENCE** For
672 details on different size build-
673 ing blocks in design, see
674 “Levels of Design” in Section
5.2.

675
676
677
678
679

680 **CROSS-REFERENCE** Mini-
681 mizing what each building
block knows about other
682 building blocks is a key part
683 of information hiding. For
684 details, see “Hide Secrets
(Information Hiding)” in
685 Section 5.3.

686
687

688

689 **CROSS-REFERENCE** For
690 details on class design, see
691 Chapter 6, “Working
Classes.”

692
693

694
695
696
697
698

699

700 **CROSS-REFERENCE** For
701 details on working with vari-
ables, see Chapters 10
702 through 13.

703
704

ganizational alternatives, the architecture provides the rationale for the system organization and shows that each class has been carefully considered. One review of design practices found that the design rationale is at least as important for maintenance as the design itself (Rombach 1990).

The architecture should define the major building blocks in a program. Depending on the size of the program, each building block might be a single class, or it might be a subsystem consisting of many classes. Each building block is a class, or a collection of classes or routines that work together on high-level functions such as interacting with the user, displaying web pages, interpreting commands, encapsulating business rules, or accessing data. Every feature listed in the requirements should be covered by at least one building block. If a function is claimed by two or more building blocks, their claims should cooperate, not conflict.

What each building block is responsible for should be well defined. A building block should have one area of responsibility, and it should know as little as possible about other building blocks’ areas of responsibility. By minimizing what each building block knows about each other building block, you localize information about the design into single building blocks.

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

Major Classes

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

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

Data Design

The architecture should describe the major files and table designs to be used. It should describe alternatives that were considered and justify the choices that were made. If the application maintains a list of customer IDs and the architects have chosen to represent the list of IDs using a sequential-access list, the document should explain why a sequential-access list is better than a random-access

705 list, stack, or hash table. During construction, such information gives you insight
706 into the minds of the architects. During maintenance, the same insight is an in-
707 valuable aid. Without it, you're watching a foreign movie with no subtitles.

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

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

717 **Business Rules**

718 If the architecture depends on specific business rules, it should identify them and
719 describe the impact the rules have on the system's design. For example, suppose
720 the system is required to follow a business rule that customer information should
721 be no more than 30 seconds out of date. In that case, the impact that has on the
722 architecture's approach to keeping customer information up to date and synchro-
723 nized should be described.

724 **User Interface Design**

725 Sometimes the user interface is specified at requirements time. If it isn't, it
726 should be specified in the software architecture. The architecture should specify
727 major elements of web page formats, GUIs, command line interfaces, and so on.
728 Careful architecture of the user interface makes the difference between a well-
729 liked program and one that's never used.

730 The architecture should be modularized so that a new user interface can be sub-
731 stituted without affecting the business rules and output parts of the program. For
732 example, the architecture should make it fairly easy to lop off a group of interac-
733 tive interface classes and plug in a group of command line classes. This ability is
734 often useful, especially since command line interfaces are convenient for soft-
735 ware testing at the unit or subsystem level.

736 The design of user interfaces deserves its own book-length discussion but is out-
737 side the scope of this book.

738 **Input/Output**

739 Input/output is another area that deserves attention in the architecture. The archi-
740 tecture should specify a look-ahead, look-behind, or just-in-time reading scheme.

741
742

And it should describe the level at which I/O errors are detected: at the field, record, stream, or file level.

743
744
745
746
747
748
749
750
751
752
753

754 CC2E.COM/0330

755 **FURTHER READING** For an
756 excellent discussion of soft-
ware security, see *Writing
757 Secure Code*, 2d Ed. (Howard
758 and LeBlanc 2003) as well as
759 the January 2002 issue of
760 *IEEE Software*.

761

762

763 **FURTHER READING** For
764 additional information on
designing systems for per-
765 formance, see Connie
766 Smith's *Performance Engi-*
767 neering of Software Systems
768 (1990).

769

770

771

772

773

774

775

776

Resource Management

The architecture should describe a plan for managing scarce resources such as database connections, threads, and handles. Memory management is another important area for the architecture to treat in memory-constrained applications areas like driver development and embedded systems. The architecture should estimate the resources used for nominal and extreme cases. In a simple case, the estimates should show that the resources needed are well within the capabilities of the intended implementation environment. In a more complex case, the application might be required to more actively manage its own resources. If it is, the resource manager should be architected as carefully as any other part of the system.

Security

The architecture should describe the approach to design-level and code-level security. If a threat model has not previously been built, it should be built at architecture time. Coding guidelines should be developed with security implications in mind, including approaches to handling buffers; rules for handling untrusted data (data input from users, cookies, configuration data, other external interfaces); encryption; level of detail contained in error messages; protecting secret data that's in memory; and other issues.

Performance

If performance is a concern, performance goals should be specified in the requirements. Performance goals can include both speed and memory use.

The architecture should provide estimates and explain why the architects believe the goals are achievable. If certain areas are at risk of failing to meet their goals, the architecture should say so. If certain areas require the use of specific algorithms or data types to meet their performance goals, the architecture should say so. The architecture can also include space and time budgets for each class or object.

Scalability

Scalability is the ability of a system to grow to meet future demands. The architecture should describe how the system will address growth in number of users, number of servers, number of network nodes, database size, transaction volume, and so on. If the system is not expected to grow and scalability is not an issue, the architecture should make that assumption explicit.

777

778

779

780

781

782

783

784

785

786

787

788

789

790

791

792

793

794

795

796

797

798

799

800

801 **HARD DATA**

802

803

804

805

806

807

808

809

810

811

812

813

814

Interoperability

If the system is expected to share data or resources with other software or hardware, the architecture should describe how that will be accomplished.

Internationalization/Localization

“Internationalization” is the technical activity of preparing a program to support multiple locales. Internationalization is often known as “I18N” because the first and last characters in “internationalization” are “I” and “N” and because there are 18 letters in the middle of the word. “Localization” (known as “L10n” for the same reason) is the activity of translating a program to support a specific local language.

Internationalization issues deserve attention in the architecture for an interactive system. Most interactive systems contain dozens or hundreds of prompts, status displays, help messages, error messages, and so on. Resources used by the strings should be estimated. If the program is to be used commercially, the architecture should show that the typical string and character-set issues have been considered, including character set used (ASCII, DBCS, EBCDIC, MBCS, Unicode, ISO 8859, and so on), kinds of strings used (C strings, Visual Basic Strings, and so on) maintaining the strings without changing code, and translating the strings into foreign languages with minimal impact on the code and the user interface. The architecture can decide to use strings in line in the code where they’re needed, keep the strings in a class and reference them through the class interface, or store the strings in a resource file. The architecture should explain which option was chosen and why.

Error Processing

Error processing is turning out to be one of the thorniest problems of modern computer science, and you can’t afford to deal with it haphazardly. Some people have estimated that as much as 90 percent of a program’s code is written for exceptional, error-processing cases or housekeeping, implying that only 10 percent is written for nominal cases (Shaw in Bentley 1982). With so much code dedicated to handling errors, a strategy for handling them consistently should be spelled out in the architecture.

Error handling is often treated as a coding-convention-level issue, if it’s treated at all. But because it has system-wide implications, it is best treated at the architectural level. Here are some questions to consider:

- Is error processing corrective or merely detective? If corrective, the program can attempt to recover from errors. If it’s merely detective, the program can continue processing as if nothing had happened, or it can quit. In either case, it should notify the user that it detected an error.

- 815 ● Is error detection active or passive? The system can actively anticipate errors—for example, by checking user input for validity—or it can passively
816 respond to them only when it can't avoid them—for example, when a com-
817 bination of user input produces a numeric overflow. It can clear the way or
818 clean up the mess. Again, in either case, the choice has user-interface impli-
819 cations.
820
821 ● How does the program propagate errors? Once it detects an error, it can im-
822 mediately discard the data that caused the error, it can treat the error as an
823 error and enter an error-processing state, or it can wait until all processing is
824 complete and notify the user that errors were detected (somewhere).
825
826 ● What are the conventions for handling error messages? If the architecture
827 doesn't specify a single, consistent strategy, the user interface will appear to
828 be a confusing macaroni-and-dried-bean collage of different interfaces in
829 different parts of the program. To avoid such an appearance, the architecture
 should establish conventions for error messages.
830 **CROSS-REFERENCE** A
831 consistent method of han-
832 dling bad parameters is an-
other aspect of error-
833 processing strategy that
834 should be addressed architec-
835 turally. For examples, see
836 Chapter 8, "Defensive Pro-
gramming."
837
838
839
840

- 841
842 **FURTHER READING** For a
843 good introduction to fault
844 tolerance, see the July 2001
845 issue of *IEEE Software*. In
846 addition to providing a good
introduction, the articles cite
847 many key books and key
articles on the topic.

Fault Tolerance

The architecture should also indicate the kind of fault tolerance expected. Fault tolerance is a collection of techniques that increase a system's reliability by detecting errors, recovering from them if possible, and containing their bad effects if not.

For example, a system could make the computation of the square root of a number fault tolerant in any of several ways:

- The system might back up and try again when it detects a fault. If the first answer is wrong, it would back up to a point at which it knew everything was all right and continue from there.

- 851 • The system might have auxiliary code to use if it detects a fault in the primary code. In the example, if the first answer appears to be wrong, the system switches over to an alternative square-root routine and uses it instead.
- 852 • The system might use a voting algorithm. It might have three square-root classes that each use a different method. Each class computes the square root, and then the system compares the results. Depending on the kind of fault tolerance built into the system, it then uses the mean, the median, or the mode of the three results.
- 853 • The system might replace the erroneous value with a phony value that it knows to have a benign effect on the rest of the system.

854
855
856
857
858
859
860
861 Other fault-tolerance approaches include having the system change to a state of
862 partial operation or a state of degraded functionality when it detects an error. It
863 can shut itself down or automatically restart itself. These examples are necessarily
864 simplistic. Fault tolerance is a fascinating and complex subject—
865 unfortunately, one that's outside the scope of this book.

866 **Architectural Feasibility**

867 The designers might have concerns about a system's ability to meet its performance
868 targets, work within resource limitations, or be adequately supported by the
869 implementation environments. The architecture should demonstrate that the system
870 is technically feasible. If infeasibility in any area could render the project
871 unworkable, the architecture should indicate how those issues have been investigated—through proof-of-concept prototypes, research, or other means. These
872 risks should be resolved before full-scale construction begins.

874 **Overengineering**

875 Robustness is the ability of a system to continue to run after it detects an error.
876 Often an architecture specifies a more robust system than that specified by the
877 requirements. One reason is that a system composed of many parts that are
878 minimally robust might be less robust than is required overall. In software, the
879 chain isn't as strong as its weakest link; it's as weak as all the weak links multiplied
880 together. The architecture should clearly indicate whether programmers
881 should err on the side of overengineering or on the side of doing the simplest
882 thing that works.

883 Specifying an approach to over-engineering is particularly important because
884 many programmers over-engineer their classes automatically, out of a sense of
885 professional pride. By setting expectations explicitly in the architecture, you can
886 avoid the phenomenon in which some classes are exceptionally robust and others
887 are barely adequate.

888

889 **CROSS-REFERENCE** For
890 a list of kinds of commer-
891 cially available software
892 components and libraries, see
893 “Code Libraries” in Section
894 30.3.

895

896

897

898

899

900

901

902

903

904

905 **CROSS-REFERENCE** For
906 details on handling changes
907 systematically, see Section
908 28.2, “Configuration Man-
agement.”

909

910

911

912

913 ***Design bugs are often***
914 ***subtle and occur by***
915 ***evolution with early***
916 ***assumptions being***
917 ***forgotten as new features***
918 ***or uses are added to a***
919 ***system.***

920

921

922

923

Buy-vs.-Build Decisions

The most radical solution to building software is not to build it at all—to buy it instead. You can buy GUI controls, database managers, image processors, graphics and charting components, Internet communications components, security and encryption components, spreadsheet tools, text processing tools—the list is nearly endless. One of the greatest advantages of programming in modern GUI environments is the amount of functionality you get automatically: graphics classes, dialog box managers, keyboard and mouse handlers, code that works automatically with any printer or monitor, and so on.

If the architecture isn’t using off-the-shelf components, it should explain the ways in which it expects custom-built components to surpass ready-made libraries and components.

Reuse Decisions

If the plan calls for using pre-existing software, the architecture should explain how the reused software will be made to conform to the other architectural goals—if it will be made to conform.

Change Strategy

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

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

924 **CROSS-REFERENCE** For
925 a full explanation of delaying
926 commitment, see “Choose
927 Binding Time Consciously”
928 in Section 5.3.

929

930 **CROSS-REFERENCE** For
931 more information about how
932 quality attributes interact, see
933 Section 20.1, “Characteristics
934 of Software Quality.”

935

936

937

938

939

940

941

942

943

944

945

946

947

948

949

950

951

952

953

954

955

956

957

958

959

960

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

General Architectural Quality

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

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

You might know of ways in which the architecture was changed during its development. Each change should fit in cleanly with the overall concept. The architecture shouldn’t look like a House appropriations bill complete with pork-barrel, boondoggle riders for each representative’s home district.

The architecture’s objectives should be clearly stated. A design for a system with a primary goal of modifiability will be different from one with a goal of uncompromised performance, even if both systems have the same function.

The architecture should describe the motivations for all major decisions. Be wary of “we’ve always done it that way” justifications. One story goes that Beth wanted to cook a pot roast according to an award-winning pot roast recipe handed down in her husband’s family. Her husband, Abdul, said that his mother had taught him to sprinkle it with salt and pepper, cut both ends off, put it in the pan, cover it, and cook it. Beth asked, “Why do you cut both ends off?” Abdul said, “I don’t know. I’ve always done it that way. Let me ask my mother.” He called her, and she said, “I don’t know. I’ve always done it that way. Let me ask your grandmother.” She called his grandmother, who said, “I don’t know why you do it that way. I did it that way because it was too big to fit in my pan.”

Good software architecture is largely machine and language independent. Admittedly, you can’t ignore the construction environment. By being as independent of the environment as possible, however, you avoid the temptation to over-architect the system or to do a job that you can do better during construction. If the pur-

961 pose of a program is to exercise a specific machine or language, this guideline
962 doesn't apply.

963 The architecture should tread the line between under-specifying and over-
964 specifying the system. No part of the architecture should receive more attention
965 than it deserves, or be over-designed. Designers shouldn't pay attention to one
966 part at the expense of another. The architecture should address all requirements
967 without gold-plating (without containing elements that are not required).

968 The architecture should explicitly identify risky areas. It should explain why
969 they're risky and what steps have been taken to minimize the risk.

970 Finally, you shouldn't be uneasy about any parts of the architecture. It shouldn't
971 contain anything just to please the boss. It shouldn't contain anything that's hard
972 for you to understand. You're the one who'll implement it; if it doesn't make
973 sense to you, how can you implement it?

CC2E.COM/0337

974 Checklist: Architecture

975 Here's a list of issues that a good architecture should address. The list isn't in-
976 tended to be a comprehensive guide to architecture but to be a pragmatic way of
977 evaluating the nutritional content of what you get at the programmer's end of the
978 software food chain. Use this checklist as a starting point for your own checklist.
979 As with the requirements checklist, if you're working on an informal project,
980 you'll find some items that you don't even need to think about. If you're work-
981 ing on a larger project, most of the items will be useful.

982 Specific Architectural Topics

- 983 Is the overall organization of the program clear, including a good architec-
984 tural overview and justification?
- 985 Are major building blocks well defined, including their areas of responsibil-
986 ity and their interfaces to other building blocks?
- 987 Are all the functions listed in the requirements covered sensibly, by neither
988 too many nor too few building blocks?
- 989 Are the most critical classes described and justified?
- 990 Is the data design described and justified?
- 991 Is the database organization and content specified?
- 992 Are all key business rules identified and their impact on the system de-
993 scribed?
- 994 Is a strategy for the user interface design described?
- 995 Is the user interface modularized so that changes in it won't affect the rest of
996 the program?

- 997 Is a strategy for handling I/O described and justified?
- 998 Are resource-use estimates and a strategy for resource management de-
- 999 scribed and justified?
- 1000 Are the architecture's security requirements described?
- 1001 Does the architecture set space and speed budgets for each class, subsystem,
- 1002 or functionality area?
- 1003 Does the architecture describe how scalability will be achieved?
- 1004 Does the architecture address interoperability?
- 1005 Is a strategy for internationalization/localization described?
- 1006 Is a coherent error-handling strategy provided?
- 1007 Is the approach to fault tolerance defined (if any is needed)?
- 1008 Has technical feasibility of all parts of the system been established?
- 1009 Is an approach to overengineering specified?
- 1010 Are necessary buy-vs.-build decisions included?
- 1011 Does the architecture describe how reused code will be made to conform to
- 1012 other architectural objectives?
- 1013 Is the architecture designed to accommodate likely changes?
- 1014 Does the architecture describe how reused code will be made to conform to
- 1015 other architectural objectives?

1016 **General Architectural Quality**

- 1017 Does the architecture account for all the requirements?
- 1018 Is any part over- or under-architected? Are expectations in this area set out
- 1019 explicitly?
- 1020 Does the whole architecture hang together conceptually?
- 1021 Is the top-level design independent of the machine and language that will be
- 1022 used to implement it?
- 1023 Are the motivations for all major decisions provided?
- 1024 Are you, as a programmer who will implement the system, comfortable with
- 1025 the architecture?

1027

1028

1029 **CROSS-REFERENCE** The
1030 amount of time you spend on
1031 prerequisites will depend on
1032 your project type. For details
1033 on adapting prerequisites to
1034 your specific project, see
1035 Section 3.2, "Determine the
Kind of Software You're
1036 Working On," earlier in this
1037 chapter.

1038

1039

1040

1041

1042

1043 **CROSS-REFERENCE** For
1044 approaches to handling
1045 changing requirements, see
1046 "Handling Requirements
1047 Changes During Construc-
1048 tion" in Section 3.4, earlier in
1049 this chapter.

1050

1051

1052

1053

1054

1055

1056

1057

1058

1059

1060

1061

1062

1063

3.6 Amount of Time to Spend on Upstream Prerequisites

The amount of time to spend on problem definition, requirements, and software architecture varies according to the needs of your project. Generally, a well-run project devotes about 10 to 20 percent of its effort and about 20 to 30 percent of its schedule to requirements, architecture, and up-front planning (McConnell 1998, Kruchten 2000). These figures don't include time for detailed design—that's part of construction.

If requirements are unstable and you're working on a large, formal project, you'll probably have to work with a requirements analyst to resolve requirements problems that are identified early in construction. Allow time to consult with the requirements analyst and for the requirements analyst to revise the requirements before you'll have a workable version of the requirements.

If requirements are unstable and you're working on a small, informal project, allow time for defining the requirements well enough that their volatility will have a minimal impact on construction.

If the requirements are unstable on any project—formal or informal—treat requirements work as its own project. Estimate the time for the rest of the project after you've finished the requirements. This is a sensible approach since no one can reasonably expect you to estimate your schedule before you know what you're building. It's as if you were a contractor called to work on a house. Your customer says, "What will it cost to do the work?" You reasonably ask, "What do you want me to do?" Your customer says, "I can't tell you, but how much will it cost?" You reasonably thank the customer for wasting your time and go home.

With a building, it's clear that it's unreasonable for clients to ask for a bid before telling you what you're going to build. Your clients wouldn't want you to show up with wood, hammer, and nails and start spending their money before the architect had finished the blueprints. People tend to understand software development less than they understand two-by-fours and sheetrock, however, so the clients you work with might not immediately understand why you want to plan requirements development as a separate project. You might need to explain your reasoning to them.

When allocating time for software architecture, use an approach similar to the one for requirements development. If the software is a kind that you haven't worked with before, allow more time for the uncertainty of designing in a new area. Ensure that the time you need to create a good architecture won't take away

1064 from the time you need for good work in other areas. If necessary, plan the architecture work as a separate project too.
1065

CC2E.COM/0344

1066 Additional Resources

1067 Requirements

1068 CC2E.COM/0351 Here are a few books that give much more detail on requirements development.

1069 Wiegers, Karl. *Software Requirements*, 2d Ed. Redmond, WA: Microsoft Press,
1070 2003. This is a practical, practitioner-focused book that describes the nuts and
1071 bolts of requirements activities including requirements elicitation, requirements
1072 analysis, requirements specification, requirements validation, and requirements
1073 management.

1074 Robertson, Suzanne and James Robertson. *Mastering the Requirements Process*,
1075 Reading, MA: Addison Wesley, 1999. This is a good alternative to Wiegers'
1076 book for the more advanced requirements practitioner.

1077 CC2E.COM/0358 Gilb, Tom. *Competitive Engineering*, Reading, Mass.: Addison Wesley, 2004.
1078 This book describes Gilb's requirements language known as "Planguage." The
1079 book covers Gilb's specific approach to requirements engineering, design and
1080 design evaluation, and evolutionary project management. This book can be
1081 downloaded from Gilb's website at www.gilb.com.

1082 IEEE Std 830-1998. *IEEE Recommended Practice for Software Requirements*
1083 *Specifications*, Los Alamitos, CA: IEEE Computer Society Press. This document
1084 is the IEEE-ANSI guide for writing software requirements specifications. It de-
1085 scribes what should be included in the specification document and shows several
1086 alternative outlines for one.

1087 CC2E.COM/0365 Abran, Alain, et al. *Swebok: Guide to the Software Engineering Body of Knowl-*
1088 *edge*, Los Alamitos, CA: IEEE Computer Society Press, 2001. This contains a
1089 detailed description of the body of software-requirements knowledge. It may
1090 also be downloaded from www.swebok.org.

1091 Other good alternatives include:

1092 Lauesen, Soren. *Software Requirements: Styles and Techniques*, Boston, Mass.:
1093 Addison Wesley, 2002.

1094 Kovitz, Benjamin, L. *Practical Software Requirements: A Manual of Content*
1095 *and Style*, Manning Publications Company, 1998.

1096 Cockburn, Alistair. *Writing Effective Use Cases*, Boston, Mass.: Addison
1097 Wesley, 2000.

Software Architecture

1098 Numerous books on software architecture have been published in the past few
1099 CC2E.COM/0372 years. Here are some of the best:

1100
1101 Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*,
1102 Second Edition, Boston, Mass.: Addison Wesley, 2003.

1103 Buschman, Frank, et al. *Pattern-Oriented Software Architecture, Volume 1: A*
1104 *System of Patterns*, New York: John Wiley & Sons, 1996.

1105 Clements, Paul, ed.. *Documenting Software Architectures: Views and Beyond*,
1106 Boston, Mass.: Addison Wesley, 2003.

1107 Clements, Paul, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*, Boston, Mass.: Addison Wesley, 2002.

1109 Fowler, Martin. *Patterns of Enterprise Application Architecture*, Boston, Mass.:
1110 Addison Wesley, 2002.

1111 Jacobson, Ivar, Grady Booch, James Rumbaugh, 1999. *The Unified Software*
1112 *Development Process*, Reading, Mass.: Addison Wesley, 1999.

1113 *IEEE Std 1471-2000. Recommended Practice for Architectural Description of*
1114 *Software Intensive Systems*, Los Alamitos, CA: IEEE Computer Society Press.
1115 This document is the IEEE-ANSI guide for creating software architecture speci-
1116 fications.

General Software Development Approaches

1117
1118 CC2E.COM/0379 Many books are available that map out different approaches to conducting a
1119 software project. Some are more sequential, and some are more iterative.

1120 McConnell, Steve. *Software Project Survival Guide*. Redmond, WA: Microsoft
1121 Press, 1998. This book presents one particular way to conduct a project. The ap-
1122 proach presented emphasizes deliberate up-front planning, requirements devel-
1123 opment, and architecture work followed by careful project execution. It provides
1124 long-range predictability of costs and schedules, high quality, and a moderate
1125 amount of flexibility.

1126 Kruchten, Philippe. *The Rational Unified Process: An Introduction, 2d Ed.*,
1127 Reading, Mass.: Addison Wesley, 2000. This book presents a project approach
1128 that is “architecture centric and use-case driven.” Like *Software Project Survival*

1129
1130 Guide, it focuses on up-front work that provides good long-range predictability
1131 of costs and schedules, high quality, and moderate flexibility. This book's ap-
1132 proach requires somewhat more sophisticated use than the approaches described
1133 in *Software Project Survival Guide* and *Extreme Programming Explained: Em-
brace Change*.

1134
1135 Jacobson, Ivar, Grady Booch, James Rumbaugh. *The Unified Software Devel-
opment Process*, Reading, Mass.: Addison Wesley, 1999. This book is a more in-
1136 depth treatment of the topics covered in *The Rational Unified Process: An Intro-
duction*, 2d Ed.

1138
1139 Beck, Kent. *Extreme Programming Explained: Embrace Change*, Reading,
1140 Mass.: Addison Wesley, 2000. Beck describes a highly iterative approach that
1141 focuses on developing requirements and designs iteratively, in conjunction with
1142 construction. The extreme programming approach offers little long-range pre-
dictability but provides a high degree of flexibility.

1143
1144 Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, Eng-
1145 land: Addison-Wesley. Gilb's approach explores critical planning, requirements,
1146 and architecture issues early in a project, then continuously adapts the project
1147 plans as the project progresses. This approach provides a combination of long-
1148 range predictability, high quality, and a high degree of flexibility. It requires
1149 more sophistication than the approaches described in *Software Project Survival
Guide* and *Extreme Programming: Embrace Change*.

1150
1151 McConnell, Steve. *Rapid Development*. Redmond, WA: Microsoft Press, 1996.
1152 This book presents a toolbox approach to project planning. An experienced pro-
1153 ject planner can use the tools presented in this book to create a project plan that
is highly adapted to a project's unique needs.

1154
1155 Boehm, Barry and Richard Turner. *Balancing Agility and Discipline: A Guide
for the Perplexed*, Boston, Mass.: Addison Wesley, 2003. This book explores the
1156 contrast between agile development and plan-driven development styles. Chapter
1157 3 has 4 especially revealing sections: A Typical Day using PSP/TSP, A Typical
1158 Day using Extreme Programming, A Crisis Day using PSP/TSP, and A Crisis
1159 Day using Extreme Programming. Chapter 5 is on using risk to balance agility,
1160 which provides incisive guidance for selecting between agile and plan-driven
1161 methods. Chapter 6, Conclusions, is also well balanced and gives great perspec-
1162 tive. Appendix E is a gold mine of empirical data on agile practices.

1163
1164 Larman, Craig. *Agile and Iterative Development: A Manager's Guide*, Boston,
1165 Mass.: Addison Wesley, 2004. This is a well-researched introduction to flexible,
1166 evolutionary development styles. It overviews Scrum, Extreme Programming,
the Unified Process, and Evo.

CC2E.COM/0386

1167

1168

1169

1170

1171

1172

1173

1174

1175

1176

Checklist: Upstream Prerequisites

- Have you identified the kind of software project you're working on and tailored your approach appropriately?
 - Are the requirements sufficiently well-defined and stable enough to begin construction (see the requirements checklist for details)?
 - Is the architecture sufficiently well defined to begin construction (see the architecture checklist for details)?
 - Have other risks unique to your particular project been addressed, such that construction is not exposed to more risk than necessary?
-

1177

Key Points

- The overarching goal of preparing for construction is risk reduction. Be sure your preparation activities are reducing risks, not increasing them.
- If you want to develop high-quality software, attention to quality must be part of the software-development process from the beginning to the end. Attention to quality at the beginning has a greater influence on product quality than attention at the end.
- Part of a programmer's job is to educate bosses and coworkers about the software-development process, including the importance of adequate preparation before programming begins.
- The kind of project you're working significantly affects construction prerequisites—many projects should be highly iterative, and some should be more sequential.
- If a good problem definition hasn't been specified, you might be solving the wrong problem during construction.
- If a good requirements work hasn't been done, you might have missed important details of the problem. Requirements changes cost 20 to 100 times as much in the stages following construction as they do earlier, so be sure the requirements are right before you start programming.
- If a good architectural design hasn't been done, you might be solving the right problem the wrong way during construction. The cost of architectural changes increases as more code is written for the wrong architecture, so be sure the architecture is right too.
- Understand what approach has been taken to the construction prerequisites on your project and choose your construction approach accordingly.

4

Key Construction Decisions

Contents

- 4.1 Choice of Programming Language
- 4.2 Programming Conventions
- 4.3 Your Location on the Technology Wave
- 4.4 Selection of Major Construction Practices

Related Topics

Upstream prerequisites: Chapter 3

Determine the kind of software you're working on: Section 3.1

Formality needed with programs of different sizes: Chapter 27

Managing construction: Chapter 28

Software design: Chapter 5, and Chapters 6 through 9

Once you're sure an appropriate groundwork has been laid for construction, preparation turns toward more construction-specific decisions. Chapter 3 discussed the software equivalent of blueprints and construction permits. You might not have had much control over those preparations, and so the focus of that chapter was on assessing what you've got to work with at the time construction begins. This chapter focuses on preparations that individual programmers and technical leads are responsible for, directly or indirectly. It discusses the software equivalent of how to select specific tools for your tool belt and how to load your truck before you head out to the jobsite.

If you feel you've read enough about construction preparations already, you might skip ahead to Chapter 5.

4.1 Choice of Programming Language

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced

28 *problems, and in effect increases the mental power of the race.*
29 *Before the introduction of the Arabic notation, multiplication*
30 *was difficult, and the division even of integers called into play*
31 *the highest mathematical faculties. Probably nothing in the*
32 *modern world would have more astonished a Greek*
33 *mathematician than to learn that ... a huge proportion of the*
34 *population of Western Europe could perform the operation of*
35 *division for the largest numbers. This fact would have seemed*
36 *to him a sheer impossibility.... Our modern power of easy*
37 *reckoning with decimal fractions is the almost miraculous*
38 *result of the gradual discovery of a perfect notation.*

39 —Alfred North Whitehead

40 The programming language in which the system will be implemented should be
41 of great interest to you since you will be immersed in it from the beginning of
42 construction to the end.

43 Studies have shown that the programming-language choice affects productivity
44 and code quality in several ways.

45 Programmers are more productive using a familiar language than an unfamiliar
46 one. Data from the Cocomo II estimation model shows that programmers
47 working in a language they've used for three years or more are about 30 percent
48 more productive than programmers with equivalent experience who are new to a
49 language (Boehm, et al 2000). An earlier study at IBM found that programmers
50 who had extensive experience with a programming language were more than
51 three times as productive as those with minimal experience (Walston and Felix
52 1977).

53 **HARD DATA**

54 Programmers working with high-level languages achieve better productivity and
55 quality than those working with lower-level languages. Languages such as C++,
56 Java, Smalltalk, and Visual Basic have been credited with improving
57 productivity, reliability, simplicity, and comprehensibility by factors of 5 to 15
58 over low-level languages such as assembly and C (Brooks 1987, Jones 1998,
59 Boehm 2000). You save time when you don't need to have an awards ceremony
60 every time a C statement does what it's supposed to. Moreover, higher-level
61 languages are more expressive than lower-level languages. Each line of code
62 says more. Table 4-1 shows typical ratios of source statements in several high-
63 level languages to the equivalent code in C. A higher ratio means that each line
64 of code in the language listed accomplishes more than does each line of code in
C.

65
66**Table 4-1. Ratio of High-Level-Language Statements to Equivalent C Code**

Language	Level relative to C
C	1 to 1
C++	1 to 2.5
Fortran 95	1 to 2
Java	1 to 2.5
Perl	1 to 6
Smalltalk	1 to 6
SQL	1 to 10
Visual Basic	1 to 4.5

67
68

Source: Adapted from Estimating Software Costs (Jones 1998) and Software Cost Estimation with Cocomo II (Boehm 2000).

69
70
71
72
73
74

Data from IBM points to another language characteristic that influences productivity: Developers working in interpreted languages tend to be more productive than those working in compiled languages (Jones 1986a). In languages that are available in both interpreted and compiled forms (such as Visual Basic), you can productively develop programs in the interpreted form and then release them in the better-performing compiled form.

75
76
77
78
79
80
81
82

Some languages are better at expressing programming concepts than others. You can draw a parallel between natural languages such as English and programming languages such as Java and C++. In the case of natural languages, the linguists Sapir and Whorf hypothesize a relationship between the expressive power of a language and the ability to think certain thoughts. The Sapir-Whorf hypothesis says that your ability to think a thought depends on knowing words capable of expressing the thought. If you don't know the words, you can't express the thought, and you might not even be able to formulate it (Whorf 1956).

83
84
85
86

Programmers may be similarly influenced by their languages. The words available in a programming language for expressing your programming thoughts certainly determine how you express your thoughts and might even determine what thoughts you can express.

87
88
89
90
91
92

Evidence of the effect of programming languages on programmers' thinking is common. A typical story goes like this: "We were writing a new system in C++, but most of our programmers didn't have much experience in C++. They came from Fortran backgrounds. They wrote code that compiled in C++, but they were really writing disguised Fortran. They stretched C++ to emulate Fortran's bad features (such as gotos and global data) and ignored C++'s rich set of object-

93 oriented capabilities.” This phenomenon has been reported throughout the
94 industry for many years (Hanson 1984, Yourdon 1986a).

95 **Language Descriptions**

96 The development histories of some languages are interesting, as are their general
97 capabilities. Here are descriptions of the most common languages in use today.

98 **Ada**

99 Ada is a general-purpose, high-level programming language based on Pascal. It
100 was developed under the aegis of the Department of Defense and is especially
101 well suited to real-time and embedded systems. Ada emphasizes data abstraction
102 and information hiding and forces you to differentiate between the public and
103 private parts of each class and package. “Ada” was chosen as the name of the
104 language in honor of Ada Lovelace, a mathematician who is considered to have
105 been the world’s first programmer. Today Ada is used primarily in military,
106 space, and avionics systems.

107 **Assembly Language**

108 Assembly language, or “assembler,” is a kind of low-level language in which
109 each statement corresponds to a single machine instruction. Because the
110 statements use specific machine instructions, an assembly language is specific to
111 a particular processor—for example, specific Intel or Motorola CPUs. Assembler
112 is regarded as the second-generation language. Most programmers avoid it
113 unless they’re pushing the limits in execution speed or code size.

114 **C**

115 C is a general-purpose, mid-level language that is originally associated with the
116 UNIX operating system. C has some high-level language features, such as
117 structured data, structured control flow, machine independence, and a rich set of
118 operators. It has also been called a “portable assembly language” because it
119 makes extensive use of pointers and addresses, has some low-level constructs
120 such as bit manipulation, and is weakly typed.

121 C was developed in the 1970s at Bell Labs. It was originally designed for and
122 used on the DEC PDP-11—whose operating system, C compiler, and UNIX
123 application programs were all written in C. In 1988, an ANSI standard was
124 issued to codify C, which was revised in 1999. C was the de facto standard for
125 microcomputer and workstation programming in the 1980s and 1990s.

126 **C++**

127 C++, an object-oriented language founded on C, was developed at Bell
128 Laboratories in the 1980s. In addition to being compatible with C, C++ provides

129 classes, polymorphism, exception handling, templates, and it provides more
130 robust type checking than C does.

131 **C#**

132 C# is a general-purpose, object-oriented language and programming
133 environment developed by Microsoft with syntax similar to C, C++, and Java
134 and provides extensive tools that aid development on Microsoft platforms.

135 **Cobol**

136 Cobol is an English-like programming language that was originally developed in
137 1959-1961 for use by the Department of Defense. Cobol is used primarily for
138 business applications and is still one of the most widely used languages today,
139 second only to Visual Basic in popularity (Feiman and Driver 2002). Cobol has
140 been updated over the years to include mathematical functions and object-
141 oriented capabilities. The acronym “Cobol” stands for Common Business-
142 Oriented Language.

143 **Fortran**

144 Fortran was the first high-level computer language, introducing the ideas of
145 variables and high-level loops. “Fortran” stands for FORmula TRANslator.
146 Fortran was originally developed in the 1950s and has seen several significant
147 revisions, including Fortran 77 in 1977, which added block structured *if-then-*
148 *else* statements and character-string manipulations. Fortran 90 added user-
149 defined data types, pointers, classes, and a rich set of operations on arrays.
150 Fortran is used mainly in scientific and engineering applications.

151 **Java**

152 Java is an object-oriented language with syntax similar to C and C++ that was
153 developed by Sun Microsystems, Inc. Java was designed to run on any platform
154 by converting Java source code to byte code, which is then run in each platform
155 within an environment known as a virtual machine. Java is in widespread use for
156 programming Web applications.

157 **JavaScript**

158 JavaScript is an interpreted scripting language that is loosely related to Java. It is
159 used primarily for adding simple functions and online applications to web pages.

160 **Perl**

161 Perl is a string-handling language that is based on C and several Unix utilities,
162 created at Jet Propulsion Laboratories. Perl is often used for system
163 administration tasks such as creating build scripts as well as for report generation
164 and processing. The acronym “Perl” stands for Practical Extraction and Report
165 Language.

166
167
168
169
170
171

PHP

PHP is an open-source scripting language with a simple syntax similar to Perl, Bourne Shell, JavaScript, and C. PHP runs on all major operating systems to execute server-side interactive functions. It can be embedded in web pages to access and present database information. The acronym “PHP” originally stood for Personal Home Page, but now stands for PHP: Hypertext Processor.

172
173
174
175
176

Python

Python is an interpreted, interactive, object-oriented language that focuses on working with strings. It is used most commonly for writing scripts and small Web applications and also contains some support for creating larger programs. It runs in numerous environments.

177
178
179
180
181
182

SQL

SQL is the de facto standard language for querying, updating, and managing relational databases. SQL stands for Structured Query Language. Unlike other languages listed in this section, SQL is a “declarative language”—meaning that it does not define a sequence of operations, but rather the result of some operations.

183
184
185
186
187
188
189
190
191
192

Visual Basic

The original version of Basic was a high-level language developed at Dartmouth College in the 1960s. The acronym BASIC stands for Beginner’s All-purpose Symbolic Instruction Code. Visual Basic is a high-level, object-oriented, visual programming version of Basic developed by Microsoft that was originally designed for creating Windows applications. It has since been extended to support customization of desktop applications such as Microsoft Office, creation of web programs, and other applications. Experts report that by the early 2000s more professional developers are working in Visual Basic than in any other language (Feiman and Driver 2002).

193
194
195
196
197
198

Language-Selection Quick Reference

199
200

Table 4-2 provides a thumbnail sketch of languages suitable for various purposes. It can point you to languages you might be interested in learning more about. But don’t use it as a substitute for a careful evaluation of a specific language for your particular project. The classifications are broad, so take them with a grain of salt, particularly if you know of specific exceptions.

Table 4-2. The Best and Worst Languages for Particular Kinds of Programs

Kind of Program	Best Languages	Worst Languages
Command-line	Cobol, Fortran, SQL	-

processing		
Cross-platform development	Java, Perl, Python	Assembler, C#, Visual Basic
Database manipulation	SQL, Visual Basic	Assembler, C
Direct memory manipulation	Assembler, C, C++	C#, Java, Visual Basic
Distributed system	C#, Java	-
Dynamic memory use	C, C++, Java	-
Easy-to-maintain program	C++, Java, Visual Basic	Assembler, Perl
Fast execution	Assembler, C, C++, Visual Basic	JavaScript, Perl, Python
For environments with limited memory	Assembler, C	C#, Java, Visual Basic
Mathematical calculation	Fortran	Assembler
Quick-and-dirty project	Perl, PHP, Python, Visual Basic	Assembler
Real-time program	C, C++, Assembler	C#, Java, Python, Perl, Visual Basic
Report writing	Cobol, Perl, Visual Basic	Assembler, Java
Secure program	C#, Java	C, C++
String manipulation	Perl, Python	C
Web development	C#, Java, JavaScript, PHP, Visual Basic	Assembler, C

201

202

203

204

205 **CROSS-REFERENCE** For
 206 more details on the power of
 207 conventions, see Sections
 208 11.3 through 11.5.

209

210

211

212

Some languages simply don't support certain kinds of programs, and those have not been listed as "worst" languages. For example, Perl is not listed as a "worst language" for mathematical calculations.

4.2 Programming Conventions

In high-quality software, you can see a relationship between the conceptual integrity of the architecture and its low-level implementation. The implementation must be consistent with the architecture that guides it and consistent internally. That's the point of construction guidelines for variable names, class names, routine names, formatting conventions, and commenting conventions.

In a complex program, architectural guidelines give the program structural balance and construction guidelines provide low-level harmony, articulating

213 each class as a faithful part of a comprehensive design. Any large program
214 requires a controlling structure that unifies its programming-language details.
215 Part of the beauty of a large structure is the way in which its detailed parts bear
216 out the implications of its architecture. Without a unifying discipline, your
217 creation will be a jumble of poorly coordinated classes and sloppy variations in
218 style.

219 What if you had a great design for a painting, but one part was classical, one
220 impressionist, and one cubist? It wouldn't have conceptual integrity no matter
221 how closely you followed its grand design. It would look like a collage. A
222 program needs low-level integrity too.

223 **KEY POINT**
224 Before construction begins, spell out the programming conventions you'll use.
225 They're at such a low level of detail that they're nearly impossible to retrofit into
226 software after it's written. Details of such conventions are provided throughout
the book.

227 4.3 Your Location on the Technology Wave

228 During my career I've seen the PC's star rise while the mainframes' star dipped
229 toward the horizon. I've seen GUI programs replace character-based programs.
230 And I've seen the Web ascend while Windows declines. I can only assume that
231 by the time you read this some new technology will be in ascendance, and web
232 programming as I know it today (2004) will be on its way out. These technology
233 cycles, or waves, imply different programming practices depending on where
234 you find yourself on the wave.

235 In mature technology environments—the end of the wave, such as web
236 programming in the mid 2000s—we benefit from a rich software development
237 infrastructure. Late-wave environments provide numerous programming
238 language choices, comprehensive error checking for code written in those
239 languages, powerful debugging tools, and automatic, reliable performance
240 optimization. The compilers are nearly bug free. The tools are well documented
241 in vendor literature, in third party books and articles, and in extensive web
242 resources. Tools are integrated, so you can do UI, database, reports, and business
243 logic from within a single environment. If you do run into problems, you can
244 readily find quirks of the tools described in FAQs. Many consultants and training
245 classes are also available.

246 In early-wave environments—web programming in the mid 1990s, for
247 example—the situation is the opposite. Few programming language choices are
248 available, and those languages tend to be buggy and poorly documented.
249 Programmers spend significant amounts of time simply trying to figure out how

250 the language works instead of writing new code. Programmers also spend
251 countless hours working around bugs in the language products, underlying
252 operating system, and other tools. Programming tools in early-wave
253 environments tend to be primitive. Debuggers might not exist at all, and
254 compiler optimizers are still only a gleam in some programmer's eye. Vendors
255 revise their compiler version often, and it seems that each new version breaks
256 significant parts of your code. Tools aren't integrated, and so you tend to work
257 with different tools for UI, database, reports, and business logic. The tools tend
258 not to be very compatible, and you can expend a significant amount of effort just
259 to keep existing functionality working against the onslaught of compiler and
260 library releases. Test automation is especially valuable because it helps you more
261 quickly detect defects arising from changes in the development environment. If
262 you run into trouble, reference literature exists on the web in some form, but it
263 isn't always reliable, and, if the available literature is any guide, every time you
264 encounter a problem it seems as though you're the first one to do so.

265 These comments might seem like a recommendation to avoid early-wave
266 programming, but that isn't their intent. Some of the most innovative
267 applications arise from early-wave programs, like Turbo Pascal, Lotus 123,
268 Microsoft Word, and the Mosaic browser. The point is that how you spend your
269 programming days will depend on where you are on the technology wave. If
270 you're in the late part of the wave, you can plan to spend most of your day
271 steadily writing new functionality. If you're in the early part of the wave, you
272 can assume that you'll spend a sizeable portion of your time trying to figure out
273 undocumented features of your programming language, debugging errors that
274 turn out to be defects in the library code, revising code so that it will work with a
275 new release of some vendor's library, and so on.

276 When you find yourself working in a primitive environment, realize that the
277 programming practices described in this book can help you even more than they
278 can in mature environments. As David Gries pointed out, your programming
279 tools don't have to determine how you think about programming (1981). Gries
280 makes a distinction between programming *in* a language vs. programming *into* a
281 language. Programmers who program "in" a language limit their thoughts to
282 constructs that the language directly support. If the language tools are primitive,
283 the programmer's thoughts will also be primitive.

284 Programmers who program "into" a language first decide what thoughts they
285 want to express, and then they determine how to express those thoughts using the
286 tools provided by their specific language.

287 In the early days of Visual Basic I was frustrated because I wanted to keep the
288 business logic, the UI, and the database separate in the product I was developing,
289 but there wasn't any built-in way to do that in VB. I knew that if I wasn't

careful, over time some of my VB “forms” would end up containing business logic, some forms would contain database code, and some would contain neither—I would end up never being able to remember which code was located in which place. I had just completed a C++ project that had done a poor job of separating those issues, and I didn’t want to experience déjà vu of those headaches in a different language.

Consequently, I adopted a design convention that the .frm file (the form file) was allowed only to retrieve data from the database and store data back into the database. It wasn’t allowed to communicate that data directly to other parts of the program. Each form supported an *IsFormCompleted()* routine, which was used by the calling routine to determine whether the form that had been activated had saved its data or not. *IsFormCompleted()* was the only public routine that forms were allowed to have. Forms also weren’t allowed to contain any business logic. All other code had to be contained in an associated .bas file, including validity checks for entries in the form.

VB did not encourage this kind of approach. It encouraged programmers to put as much code into the .frm file as possible, and it didn’t make it easy for the .frm file to call back into an associated .bas file.

This convention was pretty simple, but as I got deeper into my project, I found that it helped me avoid numerous cases in which I would have been writing convoluted code without the convention. I would have been loading forms but keeping them hidden so that I could call the data-validity checking routines inside them, or I would have been copying code from the forms into other locations, and then maintaining parallel code in multiple places. The *IsFormCompleted()* convention also kept things simple. Because every form worked exactly the same way, I never had to second-guess the semantics of *IsFormCompleted()*—it meant the same thing every time it was used.

VB didn’t support this convention directly, but the use of a simple programming convention—programming *into* the language—made up for VB’s lack of structure at that time and helped keep the project intellectually manageable.

Understanding the distinction between programming in a language and programming *into* one is critical to understanding this book. Most of the important programming principles depend not on specific languages but on the way you use them. If your language lacks constructs that you want to use or is prone to other kinds of problems, try to compensate for them. Invent your own coding conventions, standards, class libraries, and other augmentations.

326
327328
329
330
331332
333
334CC2E.COM/0496
335336
337
338
339
340
341
342
343344
345
346
347
348
349

CROSS-REFERENCE For more details on quality assurance, see Chapter 20, "The Software-Quality Landscape."

353
354
355
356
357
358

4.4 Selection of Major Construction Practices

Part of preparing for construction is deciding which of the many available good practices you'll emphasize. Some projects use pair programming and test-first development, while others use solo development and formal inspections. Either technique can work well depending on specific circumstances of the project.

The following checklist summarizes the specific practices you should consciously decide to include or exclude during construction. Details of the practices are contained throughout the book.

Checklist: Major Construction Practices

Coding

- Have you defined coding conventions for names, comments, and formatting?
- Have you defined specific coding practices that are implied by the architecture, such as how error conditions will be handled, how security will be addressed, and so on?
- Have you identified your location on the technology wave and adjusted your approach to match? If necessary, have you identified how you will program *into* the language rather than being limited by programming *in* it?

Teamwork

- Have you defined an integration procedure, that is, have you defined the specific steps a programmer must go through before checking code into the master sources?
- Will programmers program in pairs, or individually, or some combination of the two?

Quality Assurance

- Will programmers write test cases for their code before writing the code itself?
- Will programmers write unit tests for their code regardless of whether they write them first or last?
- Will programmers step through their code in the debugger before they check it in?
- Will programmers integration-test their code before they check it in?
- Will programmers review or inspect each others' code?

359 **CROSS-REFERENCE** For
more details on tools, see
360 Chapter 30, "Programming
Tools."
361

362
363
364
365

Tools

- Have you selected a revision control tool?
 - Have you selected a language and language version or compiler version?
 - Have you decided whether to allow use of non-standard language features?
 - Have you identified and acquired other tools you'll be using—editor, refactoring tool, debugger, test framework, syntax checker, and so on?
-

366

Key Points

- Every programming language has strengths and weaknesses. Be aware of the specific strengths and weaknesses of the language you're using.
- Establish programming conventions before you begin programming. It's nearly impossible to change code to match them later.
- More construction practices exist than you can use on any single project. Consciously choose the practices that are best suited to your project.
- Your position on the technology wave determines what approaches will be effective—or even possible. Identify where you are on the technology wave, and adjust your plans and expectations accordingly.

5

Design in Construction

Contents

- 5.1 Design Challenges
- 5.2 Key Design Concepts
- 5.3 Design Building Blocks: Heuristics
- 5.4 Design Practices
- 5.5 Comments on Popular Methodologies

Related Topics

Software architecture: Section 3.5

Characteristics of high-quality 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.

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

24 **CROSS-REFERENCE** For
25 details on the different levels
26 of formality required on large
and small projects, see
27 Chapter 27, "How Program
28 Size Affects Construction."

29 projects benefit from careful design just as larger projects do, and recognizing
30 design as an explicit activity maximizes the benefit you will receive from it.

31 Design is a huge topic, so only a few aspects of it are considered in this chapter.
32 A large part of good class or routine design is determined by the system
33 architecture, so be sure that the architecture prerequisite discussed in Section 3.5
34 has been satisfied. Even more design work is done at the level of individual
35 classes and routines, described in Chapters 6 and 7.

36 If you're already familiar with software design topics, you might want to read
37 the introduction in the next section, and hit the highlights in the sections about
38 design challenges in Section 5.1 and key heuristics in Section 5.3.

39 **5.1 Design Challenges**

40 **CROSS-REFERENCE** The
41 difference between heuristic
42 and deterministic processes is
43 described in Chapter 2,
44 "Metaphors for a Richer
45 Understanding of Software
Development."

The phrase "software design" means the conception, invention, or contrivance of a scheme for turning a specification for a computer program into an operational program. 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.

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

48 **Design is a Wicked Problem**

49 Horst Rittel and Melvin Webber defined a "wicked" problem as one that could
50 be clearly defined only by solving it, or by solving part of it (1973). This
51 paradox implies, essentially, that you have to "solve" the problem once in order
52 to clearly define it and then solve it again to create a solution that works. This
53 process is practically motherhood and apple pie in software development (Peters
54 and Tripp 1976).

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.

55
56
57
58
59
60 —David Parnas and Paul
61 Clements
62
63
64

F05xx01

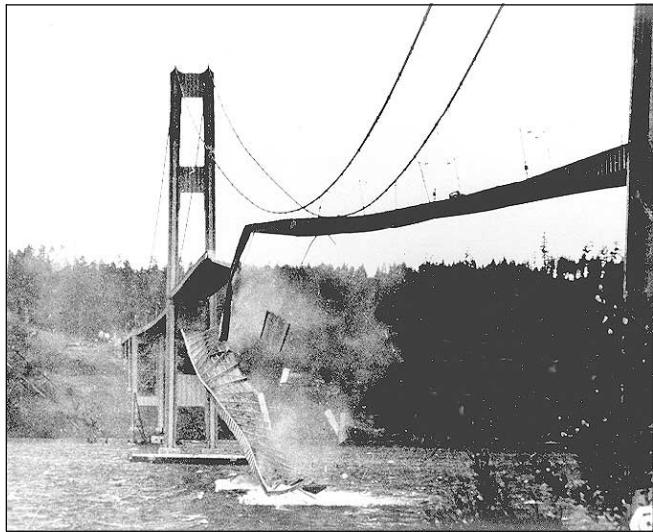
Figure 5-1

The Tacoma Narrows bridge—an example of a wicked problem.

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.

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.

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 hog tie 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.



78

79

80

81 **FURTHER READING** For a
82 fuller exploration of this
83 viewpoint, see “A Rational
84 Design Process: How and
85 Why to Fake It” (Parnas and
Clements 1986).

86

87 **CROSS-REFERENCE** For
88 a better answer to this
89 question, see “How Much
90 Design is Enough?” in
Section 5.4 later in this
91 chapter.

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

Design is a Sloppy Process

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.

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 that it would be to make the same mistakes, recognize them later, and have to correct full-blown code. Design is sloppy because a good solution is often only subtly different from a poor one.

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 Trade-Offs 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 is developed. 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 Non-Deterministic

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.

113

114 **KEY POINT**

115

116

117

118

119

120

121

122

123

124

125

126 **FURTHER READING** Software

127 isn't the only kind of
128 structure that changes over
129 time. For an interesting
insight into how physical
structures evolve, see *How
Buildings Learn* (Brand

130 1995).

131

132

133

134

135

136

137

Design is a Heuristic Process

Because design is non-deterministic, 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

A tidy way of summarizing these attributes of design is to say that design is “emergent” (Bain and Shalloway 2004). 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 itself.

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

To understand the importance of managing complexity, it’s useful to refer to Fred Brook’s landmark paper, “No Silver Bullets” (1987).

138

139 ***There are two ways of***
140 ***constructing a software***
141 ***design: One way is to***
142 ***make it so simple that***
143 ***there are obviously no***
144 ***deficiencies and the other***
145 ***is to make it so***

146 ***complicated that there are***
147 ***no obvious deficiencies.***

148 —C.A.R. Hoare

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

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, then it isn't really a car.

Accidental properties are the properties a thing just happens to have, that don't really bear on whether the thing is really that kind of thing. 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 *coincidental*, *discretionary*, *optional*, and *happenstance*.

Brooks observes that the major accidental difficulties in software were addressed long ago. 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 non-interactive 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 it is so challenging to determine 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.

176

177 *One symptom that you
178 have bogged down in
179 complexity overload is
180 when you find yourself
181 doggedly applying a
182 method that is clearly
183 irrelevant, at least to any
184 outside observer. It is like
the mechanically inept
185 person whose car breaks
186 down—so he puts water
187 in the battery and empties
the ashtrays.*

188 —P.J. Plauger

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206 **CROSS-REFERENCE** For
207 discussion on the way
208 complexity affects other
209 programming issues, see
210 “Software’s Primary
211 Technical Imperative:
212 Managing Complexity” in
213 Section 5.2 and Section 34.1,
“Conquer Complexity.”

Importance of Managing Complexity

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 really understands the impact that code changes in one area will have on other areas, progress grinds to a halt.

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 gave 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.

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.

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

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

220 **How to Attack Complexity**

221 There are three sources of overly costly, ineffective designs:

- 222 • A complex solution to a simple problem
223 • A simple, incorrect solution to a complex problem
224 • An inappropriate, complex solution to a complex problem

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

- 229 **KEY POINT**
230 • Minimize the amount of essential complexity that anyone's brain has to deal
 with at any one time.
231 • Keep accidental complexity from needlessly proliferating.

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

234 **Desirable Characteristics of a Design**

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

241 Here's a list of internal design characteristics:

242

243 **CROSS-REFERENCE** These characteristics are related to
244 general software-quality
245 attributes. For details on
246 general attributes, see Section
247 20.1, "Characteristics of
248 Software Quality."

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271 **HARD DATA**

272

273

274

275

276

277

Minimal complexity

The primary goal of design should be to minimize complexity for all the reasons described in the last section. 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.

Minimal connectedness

Minimal connectedness means designing so that you hold connections among different parts of a program to a minimum. Use the principles of strong cohesion, loose coupling, 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 from within a class (Card and Glass 1990; Basili, Briand, and Melo 1996).

278

279

280

281

282

283

284

285

286

287

288

289

290

291

292

293

294 **CROSS-REFERENCE** For
295 more on working with old
296 systems, see Section 24.6,
297 “Refactoring Strategies.”

298

299

300

301

302

303 **CROSS-REFERENCE** An
304 especially valuable kind of
305 standardization is the use of
306 design patterns, which are
307 discussed in “Look for
308 Common Design Patterns” in
309 Section 5.3.

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

Stratified design 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.

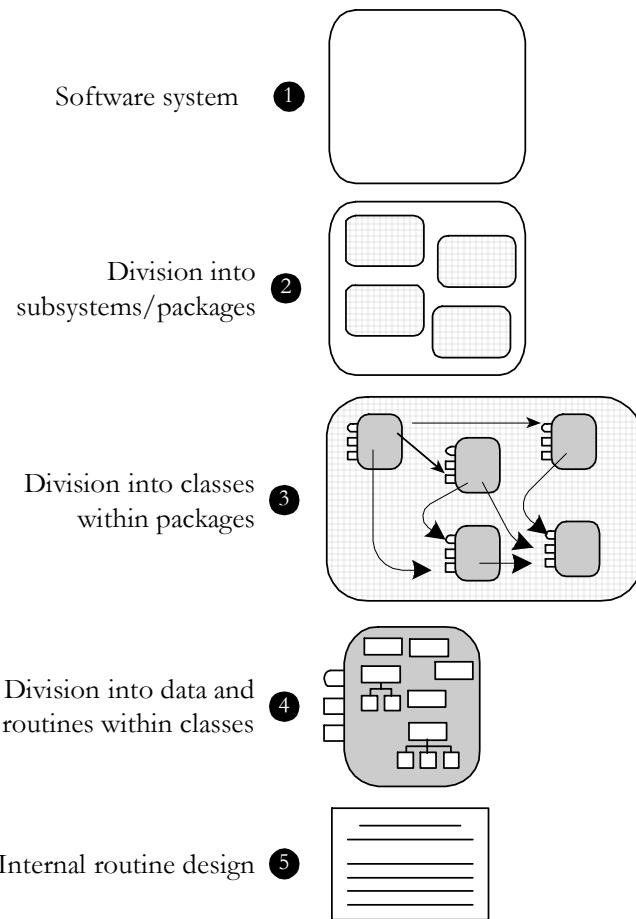
If you’re writing a modern system that has to use a lot of older, poorly designed code, write a layer of the new system that’s responsible for interfacing with the old code. Design the layer so that it hides the poor quality of the old code, presenting a consistent set of services to the newer layers. 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, you won’t need to modify any new code except the interface layer.

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.



310

311

312

313

314

315

316

317

318

319

320

321

322

323

324

325

326

F05xx02

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

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 logic, 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 subsystems. Division at this level is typically needed on any project that takes longer than a few

327 weeks. Within each subsystem, different methods of design might be used—
328 choosing the approach that best fits each part of the system. In Figure 5-2, design
329 at this level is shown in (2).

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

334 Suppose for example that you define a system with six subsystems, like this:

335 **Error! Objects cannot be created from editing field codes.**

336 **F05xx03**

337 **Figure 5-3**

338 *An example of a system with six subsystems.*

339 When there are no rules, the second law of thermodynamics will come into play
340 and the entropy of the system will increase. One way in which entropy increases
341 is that, without any restrictions on communications among subsystems,
342 communication will occur in an unrestricted way, like this:

343 **Error! Objects cannot be created from editing field codes.**

344 **F05xx04**

345 **Figure 5-4**

346 *An example of what happens with no restrictions on inter-subsystem
347 communications.*

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

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

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

362 With forethought, all of these issues can be addressed with little extra work.
363 Allow communication between subsystems only on a “need to know” basis—and
364 it had better be a *good* reason. If in doubt, it’s easier to restrict communication
365 early and relax it later than it is to relax it early and then try to tighten it up later
366 after you’ve coded several hundred inter-subsystem calls.

367 Figure 5-5 shows how a few communication guidelines could change the system
368 depicted in Figure 5-4:

369 **Error! Objects cannot be created from editing field codes.**

370 **F05xx05**

371 **Figure 5-5**

372 *With a few communication rules, you can simplify subsystem interactions
373 significantly.*

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

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

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

387 **Common Subsystems**

388 Some kinds of subsystems appear time and again in different systems. Here are
389 some of the usual suspects.

390 ***Business logic***

391 Business logic is the laws, regulations, policies, and procedures that you encode
392 into a computer system. If you’re writing a payroll system, you might encode
393 rules from the IRS about the number of allowable withholdings and the
394 estimated tax rate. Additional rules for a payroll system might come from a
395 union contract specifying overtime rates, vacation and holiday pay, and so on. If
396 you’re writing a program to quote auto insurance rates, rules might come from
397 state regulations on required liability coverages, actuarial rate tables, or
398 underwriting restrictions.

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

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

414 **System dependencies**
415 Package operating-system dependencies into a subsystem for the same reason
416 you package hardware dependencies. If you're developing a program for
417 Microsoft Windows, for example, why limit yourself to the Windows
418 environment? Isolate the Windows calls in a Windows-interface subsystem. If
419 you later want to move your program to a Macintosh or Linux, all you'll have to
420 change is the interface subsystem. This functionality can be too extensive to
421 implement the details on your own, but it's readily available in any of several
422 commercial code libraries.

423 **Level 3: Division into Classes**

424 **FURTHER READING** For a
425 good discussion of database
426 design, see *Agile Database
427 Techniques* (Ambler 2003).
428
429

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 meta data. 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.

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

435 The division of subsystems into classes is typically needed on any project that
436 takes longer than a few days. If the project is large, the division is clearly distinct
437 from the program partitioning of Level 2. If the project is very small, you might

438 move directly from the whole-system view of Level 1 to the classes view of
439 Level 3.

440

441 **Classes vs. Objects**

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

452

453 **CROSS-REFERENCE** For
454 details on characteristics of
455 high-quality classes, see
456 Chapter 6, "Working
457 Classes."

458

459

460

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.

461

462

463

This level of decomposition and design is often left up to the individual
programmer, and it is 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.

464

465 *In other words—and this*
466 *is the rock-solid principle*
467 *on which the whole of the*
468 *Corporation's*
469 *Galaxywide success is*
470 *founded—their*
471 *fundamental design flaws*
472 *are completely hidden by*
their superficial design
flaws.

—Douglas Adams

473

474

475

476

477

478

479

480

481

482

483

484

485

486

487

488

489 *Ask not first what the
490 system does; ask WHAT it
491 does to!*

—Bertrand Meyer

492

493 **CROSS-REFERENCE** For
more details on designing
494 using classes, see Chapter 6,
“Working Classes.”

496

497

498

499

500

501

502

503

504

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 non-deterministic, skillful application of an effective set of heuristics is the core activity in good software design. The following sections 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 sections describe each of the heuristics in terms of Software’s Primary Technical Imperative: Managing Complexity.

Find Real-World Objects

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.

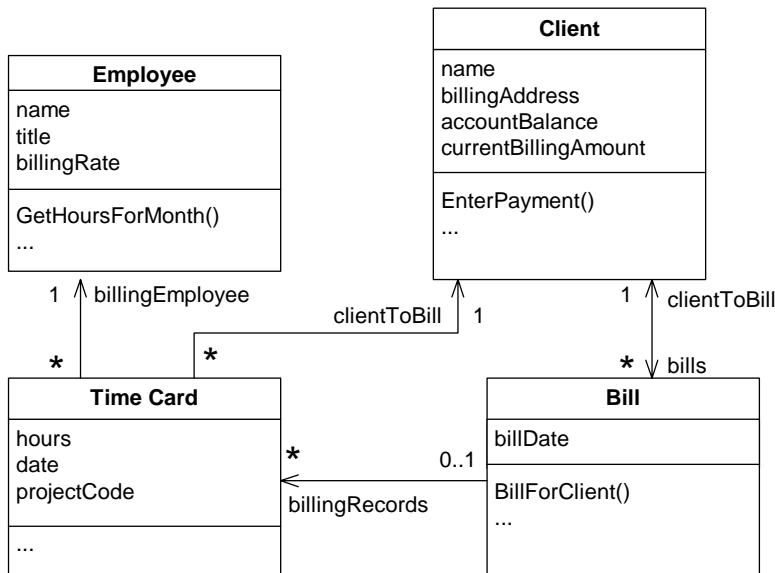
The steps in designing with objects are

- Identify the objects and their attributes (methods and data).
- Determine what can be done to each object.
- Determine what each object can 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, time cards, and bills. Figure 5-6 shows an object-oriented view of such a billing system.



F05xx06

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 can have its name or billing address changed, and so on.

Determine what each object can 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 time card can contain an employee and a client. A bill can contain one or

505

506

507

508

509

510

511

512

513

514

515

516

517

518

519

520

521

522

523

524

525

526

527

528

529

530
531
532

533 **CROSS-REFERENCE** For
534 details on classes and
535 information hiding, see “Hide
536 Secrets (Information
Hiding)” in Section 5.3.

537
538
539
540

541
542
543
544
545

546

547
548
549
550
551
552

553
554
555
556
557
558
559

560
561
562
563
564
565
566

more time cards. In addition, a bill can indicate that a client has been billed. A client can enter payments against a bill. A more complicated system would include additional interactions.

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 services.

Define each object’s interface

Define the formal, syntactic, programming-language-level interfaces to each object. This includes services offered by the class as well as inheritance relationships among classes.

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.

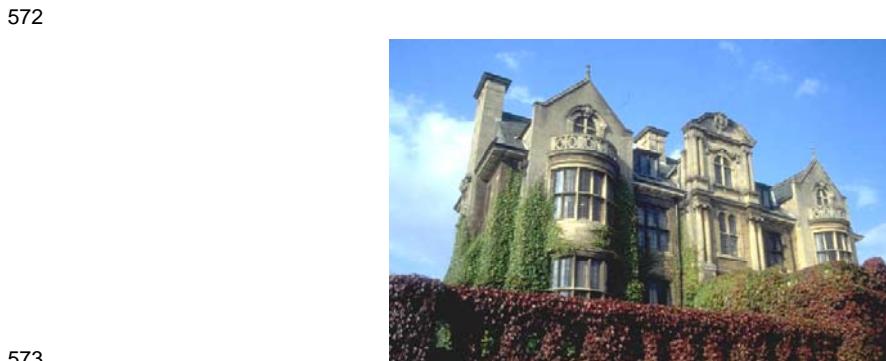
Form Consistent Abstractions

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. 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.

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



F05xx07

Figure 5-7

574 *Abstraction allows you to take a simpler view of a complex concept.*

575 **CROSS-REFERENCE** For
576 more details on abstraction in
577 class design, see "Good
578 Abstraction" in Section 6.2.
579
580
581
582
583
584

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

Encapsulate Implementation Details

586 Encapsulation picks up where abstraction leaves off. Abstraction says, "You're
587 allowed to look at an object at a high level of detail." Encapsulation says,
588 "Furthermore, you aren't allowed to look at an object at any other level of
589 detail."

590 To continue the housing-materials analogy: Encapsulation is a way of saying that
591 you can look at the outside of the house, but you can't get close enough to make
592 out the door's details. You are allowed to know that there's a door, and you're
593 allowed to know whether the door is open or closed, but you're not allowed to
594 know whether the door is made of wood, fiberglass, steel, or some other
595 material, and you're certainly not allowed to look at each individual wood fiber.

596 As Figure 5-8 suggests, encapsulation helps to manage complexity by forbidding
597 you to look at the complexity. The section titled “Good Encapsulation” in Section
598 6.2 provides more background on encapsulation as it applies to class design.



599

600

601

602

603

604

F05xx08

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!

605

Inherit When Inheritance Simplifies the Design

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

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

620 The benefit of inheritance is that it works synergistically with the notion of
621 abstraction. Abstraction deals with objects at different levels of detail. Recall the
622 door that was a collection of certain kinds of molecules at one level; a collection
623 of wood fibers at the next; and something that keeps burglars out of your house

624 at the next level. Wood has certain properties (for example, you can cut it with a
625 saw or glue it with wood glue), and two-by-fours or cedar shingles have the
626 general properties of wood as well as some specific properties of their own.

627 Inheritance simplifies programming because you write a general routine to
628 handle anything that depends on a door's general properties and then write
629 specific routines to handle specific operations on specific kinds of doors. Some
630 operations, such as *Open()* or *Close()*, might apply regardless of whether the
631 door is a solid door, interior door, exterior door, screen door, French door, or
632 sliding glass door. The ability of a language to support operations like *Open()* or
633 *Close()* without knowing until run time what kind of door you're dealing with is
634 called "polymorphism." Object-oriented languages such as C++, Java, and
635 Visual Basic support inheritance and polymorphism.

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

639 **Hide Secrets (Information Hiding)**

640 Information hiding is part of the foundation of both structured design and
641 object-oriented design. In structured design, the notion of "black boxes"
642 comes from information hiding. In object-oriented design, it gives rise to the
643 concepts of encapsulation and modularity, and it is associated with the
644 concept of abstraction.

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

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

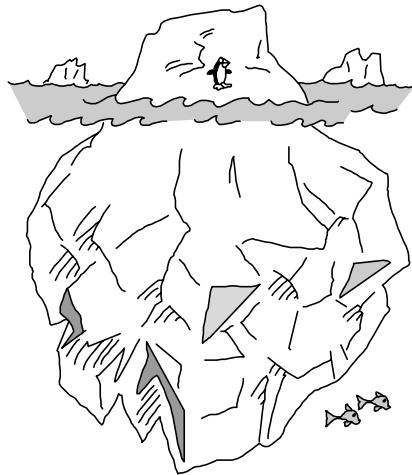
657 Information hiding is a particularly powerful heuristic for Software's Primary
658 Technical Imperative because, from its name on, it emphasizes *hiding*
659 *complexity*.

660 **Secrets and the Right to Privacy**

661 In information hiding, each class (or package or routine) is characterized by the
662 design or construction decisions that it hides from all other classes. The secret
663 might be an area that's likely to change, the format of a file, the way a data type
664 is implemented, or an area that needs to be walled off from the rest of the
665 program so that errors in that area cause as little damage as possible. The class's
666 job is to keep this information hidden and to protect its own right to privacy.
667 Minor changes to a system might affect several routines within a class, but they
668 should not ripple beyond the class interface.

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

675 The interface to a class should reveal as little as possible about its inner
676 workings. A class is a lot like an iceberg: Seven-eighths is under water, and you
677 can see only the one-eighth that's above the surface.



678

679

F05xx09

680

Figure 5-9

681

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

682

683

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

686 An Example of Information Hiding

687 Suppose you have a program in which each object is supposed to have a
688 unique ID stored in a member variable called *id*. One design approach would
689 be to use integers for the IDs and to store the highest ID assigned so far in a
690 global variable called *g_maxId*. As each new object is allocated, perhaps in
691 each object's constructor, you could simply use the statement

692 *id* = *++g_maxId*;

693 That would guarantee a unique *id*, and it would add the absolute minimum of
694 code in each place an object is created. What could go wrong with that?

695 A lot of things could go wrong. What if you want to reserve ranges of IDs for
696 special purposes? What if you want to be able to reuse the IDs of objects that
697 have been destroyed? What if you want to add an assertion that fires when
698 you allocate more IDs than the maximum number you've anticipated? If you
699 allocated IDs by spreading *id* = *++g_maxId* statements throughout your
700 program, you would have to change code associated with every one of those
701 statements.

702 The way that new IDs are created is a design decision that you should hide. If
703 you use the phrase *++g_maxId* throughout your program, you expose the way
704 a new ID is created, which is simply by incrementing *g_maxId*. If instead you
705 put the statement

706 *id* = *NewId()*;

707 throughout your program, you hide the information about how new IDs are
708 created. Inside the *NewId()* routine you might still have just one line of code,
709 *return (++g_maxId)* or its equivalent, but if you later decide to reserve
710 certain ranges of IDs for special purposes or to reuse old IDs, you could
711 make those changes within the *NewId()* routine itself—without touching
712 dozens or hundreds of *id* = *NewId()* statements. No matter how complicated
713 the revisions inside *NewId()* might become, they wouldn't affect any other
714 part of the program.

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

720 An additional secret to hide is the ID's type. In C++ you could use a simple
721 *typedef* to declare your IDs to be of *IdType*—a user-defined type that resolves
722 to *int*—rather than directly declaring them to be of type *int*. Alternatively, in
723 C++ and other languages you could create a simple *IdType* class. Once again,

724

725

726 KEY POINT

727

728

729

730

731

732

733

734

735

736

737

738

739 **FURTHER READING** Parts
740 of this section are adapted
741 from "Designing Software
for Ease of Extension and
Contraction" (Parnas 1979).

742

743

744

745

746

747

748

749

750

751

752

753 **CROSS-REFERENCE** For
754 more on accessing global
755 data through class interfaces,
756 see "Using Access Routines
Instead of Global Data" in
757 Section 13.3.

758

759

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

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.

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.

760
761
762
763

764
765

766
767
768
769
770
771

772
773
774
775
776
777
778

779
780
781
782
783

784
785 **CROSS-REFERENCE** Cod
786 e-level performance
787 optimizations are discussed
788 in Chapter 25, "Code-Tuning
789 Strategies" and Chapter 26,
790 "Code-Tuning Techniques."

791
792
793
794
795
796
797

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: (1) Routines operate on global data without knowing that other routines are operating on it; and (2) 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.

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.

798

799 **HARD DATA**

800

801

802

803

804

805

806

807

808

809

810

811

812

813

814

815

816

817

818

819

820

821

822

823

824

825

826

827

828

829

830

831

832

833

834

835

836

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 create an interface for an *Id* class; 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

837
838
839

840 **KEY POINT**

841

842

843 **FURTHER READING** The
844 approach described in this
845 section is adapted from
846 “Designing Software for Ease
847 of Extension and
Contraction” (Parnas 1979).

848

849

850

851

852

853

854

855

856

857

858

859

860

861

862

863 **CROSS-REFERENCE** One
864 of the most powerful
865 techniques for anticipating
change is to use table driven
866 methods. For details, see
867 Chapter 18, “Table-Driven
868 Methods.”

869

870

871

872

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.

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

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 class. 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:

Business logic

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

873 dependencies helps when you move the program to a new hardware
874 environment. It also helps initially when you're developing a program for
875 volatile hardware. You can write software that simulates interaction with specific
876 hardware, have the hardware-interface subsystem use the simulator as long as the
877 hardware is unstable or unavailable, and then unplug the hardware-interface
878 subsystem from the simulator and plug the subsystem into the hardware when
879 it's ready to use.

Input and output

880 At a slightly higher level of design than raw hardware interfaces, input/output is
881 a volatile area. If your application creates its own data files, the file format will
882 probably change as your application becomes more sophisticated. User-level
883 input and output formats will also change—the positioning of fields on the page,
884 the number of fields on each page, the sequence of fields, and so on. In general,
885 it's a good idea to examine all external interfaces for possible changes.
886

Nonstandard language features

887 Most language implementations contain handy, nonstandard extensions. Using
888 the extensions is a double-edged sword because they might not be available in a
889 different environment, whether the different environment is different hardware, a
890 different vendor's implementation of the language, or a new version of the
891 language from the same vendor.
892

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

Difficult design and construction areas

898 It's a good idea to hide difficult design and construction areas because they
899 might be done poorly and you might need to do them again. Compartmentalize
900 them and minimize the impact their bad design or construction might have on the
901 rest of the system.
902

Status variables

903 Status variables indicate the state of a program and tend to be changed more
904 frequently than most other data. In a typical scenario, you might originally define
905 an error-status variable as a boolean variable and decide later that it would be
906 better implemented as an enumerated type with the values *ErrorType_None*,
907 *ErrorType_Warning*, and *ErrorType_Fatal*.
908

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

- 911
912
913
914
- 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.
- 915
916
917
918
919
920

921
922
923
924
925

926

927 **CROSS-REFERENCE** This
928 section's approach to
929 anticipating change does not
930 involve designing ahead or
931 coding ahead. For a
932 discussion of those practices,
933 see "A program contains
934 code that seems like it might
be needed someday" in
Section 24.3.

935 **FURTHER READING** This
936 discussion draws on the
937 approach described in "On
the design and development
938 of program families" (Parnas
939 1976).

940
941
942

943

944
945
946
947
948

Data-size constraints

When you declare an array of size *15*, 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 *15*.

Anticipating Different Degrees of Change

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. 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 (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.

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

957 Try to create modules that depend little on other modules. Make them detached,
958 as business associates are, rather than attached, as Siamese twins are. A routine
959 like *sin()* is loosely coupled because everything it needs to know is passed in to it
960 with one value representing an angle in degrees. A routine such as *InitVars(var*
961 *1, var2, var3, ..., varN)* is more tightly coupled because, with all the variables it
962 must pass, the calling module practically knows what is happening inside
963 *InitVars()*. Two classes that depend on each other's use of the same global data
964 are even more tightly coupled.

965 **Coupling Criteria**

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

967 **Size**

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

974 **Visibility**

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

982 **Flexibility**

983 Flexibility refers to how easily you can change the connections between
984 modules. Ideally, you want something more like the USB connector on your
985 computer than like bare wire and a soldering gun. Flexibility is partly a product
986 of the other coupling characteristics, but it's a little different too. Suppose you
987 have a routine that looks up an employee's vacation benefit, given a hiring date

988 and a job classification. Name the routine *LookupVacationBenefit()*. Suppose in
989 another module you have an *employee* object that contains the hiring date and
990 the job classification, among other things, and that module passes the object to
991 *LookupVacationBenefit()*.

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

999 For the third module to use *LookupVacationBenefit()*, it has to know about the
1000 *Employee* class. It could dummy up an *employee* object with only two fields, but
1001 that would require internal knowledge of *LookupVacationBenefit()*, namely that
1002 those are the only fields it uses. Such a solution would be a kludge, and an ugly
1003 one. The second option would be to modify *LookupVacationBenefit()* so that it
1004 would take hiring date and job classification instead of *employee*. In either case,
1005 the original module turns out to be a lot less flexible than it seemed to be at first.

1006 The happy ending to the story is that an unfriendly module can make friends if
1007 it's willing to be flexible—in this case, by changing to take hiring date and job
1008 classification specifically instead of *employee*.

1009 In short, the more easily other modules can call a module, the more loosely
1010 coupled it is, and that's good because it's more flexible and maintainable. In
1011 creating a system structure, break up the program along the lines of minimal
1012 interconnectedness. If a program were a piece of wood, you would try to split it
1013 with the grain.

1014 **Kinds of Coupling**

1015 Here are the most common kinds of coupling you'll encounter.

1016 ***Simple-data-parameter coupling***

1017 Two modules are simple-data-parameter coupled if all the data passed between
1018 them are of primitive data types and all the data is passed through parameter
1019 lists. This kind of coupling is normal and acceptable.

1020 ***Simple-object coupling***

1021 A module is simple-object coupled to an object if it instantiates that object. This
1022 kind of coupling is fine.

1023
1024
1025
1026

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.

1027
1028
1029
1030

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:

1031
1032
1033
1034
1035

- *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.

1036
1037
1038
1039

- *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.

1040
1041
1042
1043
1044

- *Module1*'s interface states that its *Module1.Initialize()* routine should be called before its *Module1.Routine1()* is called. *Module2* knows that *Module1.Routine1()* calls *Module1.Initialize()* anyway, so it just instantiates *Module1* and calls *Module1.Routine1()* without calling *Module1.Initialize()* first.

1045
1046
1047

- *Module1* passes *Object* to *Module2*. Because *Module1* knows that *Module2* uses only three of *Object*'s seven methods, it only initializes *Object* only partially—with the specific data those three methods need.

1048
1049
1050
1051

- *Module1* passes *BaseObject* to *Module2*. Because *Module2* knows that *Module2* is really passing it *DerivedObject*, it casts *BaseObject* to *DerivedObject* and calls methods that are specific to *DerivedObject*.

1052
1053
1054
1055
1056

• *DerivedClass* modifies *BaseClass*'s protected member data directly. 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.

1057
1058
1059
1060

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—

1061 knowledge of its internal workings, modification to global data, uncertain
1062 functionality—the abstractive power is lost and the module’s ability to help
1063 manage complexity is reduced or eliminated.

1064 **KEY POINT**

1065

1066 CC2E.COM/0585

1067 Design patterns provide the cores of ready-made solutions that can be used to
1068 solve many of software’s most common problems. Some software problems
1069 require solutions that are derived from first principles. But most problems are
1070 similar to past problems, and those can be solved using similar solutions, or
1071 patterns. Common patterns include Adapter, Bridge, Decorator, Facade, Factory
1072 Method, Observor, Singleton, Strategy, and Template Method.

1073 Patterns provide several benefits that fully-custom design doesn’t:

Patterns reduce complexity by providing ready-made abstractions

1074 If you say, “Let’s use a Factory Method to create instances of derived classes,”
1075 other programmers on your project will understand that you are suggesting a
1076 fairly rich set of interrelationships and programming protocols, all of which are
1077 invoked when you refer to the design pattern of Factory Method.* You don’t
1078 have to spell out every line of code for other programmers to understand your
1079 proposal.

Patterns reduce errors by institutionalizing details of common solutions

1081 Software design problems contain nuances that emerge fully only after the
1082 problem has been solved once or twice (or three times, or four times, or ...).
1083 Because patterns represent standardized ways of solving common problems, they
1084 embody the wisdom accumulated from years of attempting to solve those
1085 problems, and they also embody the corrections to the false attempts that people
1086 have made in solving those problems.

1088 Using a design pattern is thus conceptually similar to using library code instead
1089 of writing your own. Sure, everybody has written a custom Quicksort a few
1090 times, but what are the odds that your custom version will be fully correct on the

* 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).

1091 first try? Similarly, numerous design problems are similar enough to past
1092 problems that you're better off using a prebuilt design solution than creating a
1093 novel solution.

1094 ***Patterns provide heuristic value by suggesting design alternatives***

1095 A designer who's familiar with common patterns can easily run through a list of
1096 patterns and ask, "Which of these patterns fits my design problem?" Cycling
1097 through a set of familiar alternatives is immeasurably easier than creating a
1098 custom design solution out of whole cloth. And the code arising from a familiar
1099 pattern will also be easier for readers of the code to understand than fully custom
1100 code would be.

1101 ***Patterns streamline communication by moving the design dialog to a***
1102 ***higher level***

1103 In addition to their complexity-management benefit, design patterns can
1104 accelerate design discussions by allowing designers to think and discuss at a
1105 larger level of granularity. If you say, "I can't decide whether I should use a
1106 Creator or a Factory Method in this situation," you've communicated a great
1107 deal with just a few words—as long as you and your listener are both familiar
1108 with those patterns. Imagine how much longer it would take you to dive into the
1109 details of the code for a Creator pattern and the code for a Factory Method
1110 pattern, and then compare and contrast the two approaches.

1111 If you're not already familiar with design patterns, Table 5-1 summarizes some
1112 of the most common patterns to stimulate your interest.

1113 **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 sync with each other by making a third object responsible for notifying the set of objects about changes to members 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.

1114
1115
1116
1117
1118 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.

1119
1120
1121 The only real potential trap with patterns is feature-it-is: using a pattern because
of a desire to try out a pattern rather than because the pattern is an appropriate
design solution.

1122 Overall, design patterns are a powerful tool for managing complexity. You can
1123 read more detailed descriptions in any of the good books that are listed at the end
1124 of this chapter.

1125 Other Heuristics

1126 The preceding sections describe the major software design heuristics. There are a
1127 few other heuristics that might not be useful quite as often but are still worth
1128 mentioning.

1129 Aim for Strong Cohesion

1130 Cohesion arose from structured design and is usually discussed in the same
1131 context as coupling. Cohesion refers to how closely all the routines in a class or
1132 all the code in a routine support a central purpose. Classes that contain strongly
1133 related functionality are described as having strong cohesion, and the heuristic
1134 goal is to make cohesion as strong as possible. Cohesion is a useful tool for
1135 managing complexity because the more code in a class supports a central
1136 purpose, the more easily your brain can remember everything the code does.

1137 Thinking about cohesion at the routine level has been a useful heuristic for
1138 decades and is still useful today. At the class level, the heuristic of cohesion has
1139 largely been subsumed by the broader heuristic of well-defined abstractions,
1140 which was discussed earlier in this chapter and in Chapter 6, "Working Classes."

1141 (Abstractions are useful at the routine level, too, but on a more even footing with
1142 cohesion at that level of detail.

1143 **Build Hierarchies**

1144 A hierarchy is a tiered information structure in which the most general or
1145 abstract representation of concepts are contained at the top of the hierarchy, with
1146 increasingly detailed, specialized representations at the hierarchy's lower levels.
1147 In software, hierarchies are found most commonly in class hierarchies, but as
1148 Level 4 in Figure 5-2 illustrated, programmers work with routine calling
1149 hierarchies as well.

1150 Hierarchies have been an important tool for managing complex sets of
1151 information for at least 2000 years. Aristotle used a hierarchy to organize the
1152 animal kingdom. Humans frequently use outlines to organize complex
1153 information (like this book). Researchers have found that people generally find
1154 hierarchies to be a natural way to organize complex information. When they
1155 draw a complex object such as a house, they draw it hierarchically. First they
1156 draw the outline of the house, then the windows and doors, and then more details
1157 They don't draw the house brick by brick, shingle by shingle, or nail by nail
1158 (Simon 1996).

1159 Hierarchies are a useful tool for achieving Software's Primary Technical
1160 Imperative because they allow you to focus on only the level of detail you're
1161 currently concerned with. The details don't go away completely; they're simply
1162 pushed to another level so that you can think about them when you want to
1163 rather than thinking about all the details all of the time.

1164 **Formalize Class Contracts**

1165 At a more detailed level, thinking of each class's interface as a contract with the
1166 rest of the program can yield good insights. Typically, the contract is something
1167 like "If you promise to provide data x, y, and z and you promise they'll have
1168 characteristics a, b, and c, I promise to perform operations 1, 2, and 3 within
1169 constraints 8, 9, and 10." The promises the clients of the class make to the class
1170 are typically called "preconditions," and the promises the object makes to its
1171 clients are called the "postconditions."

1172 Contracts are useful for managing complexity because, at least in theory, the
1173 object can safely ignore any non-contractual behavior. In practice, this issue is
1174 much more difficult. For more on contracts, see "Use assertions to document
1175 preconditions and postconditions" in Section 8.2.

1176
1177
1178
1179
1180

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.

1181
1182
1183
1184
1185
1186
1187

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 it minimizes dependencies on other subsystems? Designing for test tends to result in more formalized class interfaces, which is generally beneficial.

1188
1189
1190
1191
1192
1193
1194
1195
1196

Avoid Failure

Civil engineering professor Henry Petroski wrote an interesting book called *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.

1197
1198
1199

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.

1200
1201 **CROSS-REFERENCE** For
1202 more on binding time, see
1203 Section 10.6, "Binding
1204 Time."
1205
1206

Choose Binding Time Consciously

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, What if I bound these values earlier? or What if I bound these values later? What if I initialized this table right here in the code, or what if I read the value of this variable from the user at run time?

1207
1208
1209
1210
1211
1212

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.

1213
1214

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

1215 **When in doubt, use brute
force.**

1217 —**Butler Lampson**

1218
1219
1220
1221
1222

Consider Using Brute Force

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).

1223
1224
1225
1226
1227
1228

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 generally.

1229
1230
1231
1232
1233
1234
1235

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. If your routines are like black boxes, they're perfectly modular, perform well-defined functions, and have simple interfaces.

1236
1237
1238
1239

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 it's worth having in your back pocket.

1240

Summary of Design Heuristics

1241
1242
1243
1244
1245
1246

- Find Real-World Objects
- Form Consistent Abstractions
- Encapsulate Implementation Details
- Inherit When Possible
- Hide Secrets (Information Hiding)

- 1247 • Identify Areas Likely to Change
1248 • Keep Coupling Loose
1249 • Look for Common Design Patterns

1250 The following heuristics are sometimes useful too:

- 1251 • Aim for Strong Cohesion
1252 • Build Hierarchies
1253 • Formalize Class Contracts
1254 • Assign Responsibilities
1255 • Design for Test
1256 • Avoid Failure
1257 • Choose Binding Time Consciously
1258 • Make Central Points of Control
1259 • Consider Using Brute Force
1260 • Draw a Diagram
1261 • Keep Your Design Modular

1262 **Guidelines for Using Heuristics**

1263 *More alarming, the same*
1264 *programmer is quite*
1265 *capable of doing the same*
1266 *task himself in two or*
1267 *three ways, sometimes*
1268 *unconsciously, but quite*
1269 *often simply for a change,*
1270 *or to provide elegant*
1271 *variation ...*

1272 —A. R. Brown and W. A.

1273 Sampson

1274
1275
1276
1277
1278
1279
1280
1281

CC2E.COM/0592

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*

1282 *it? Can you use its result? Can you use its method? Should you introduce some*
1283 *auxiliary element in order to make its use possible?*

1284 *Can you restate the problem? Can you restate it still differently? Go back to*
1285 *definitions.*

1286 *If you cannot solve the proposed problem, try to solve some related problem first.*
1287 *Can you imagine a more accessible related problem? A more general problem? A*
1288 *more special problem? An analogous problem? Can you solve a part of the problem?*
1289 *Keep only a part of the condition, drop the other part; how far is the unknown then*
1290 *determined, how can it vary? Can you derive something useful from the data? Can*
1291 *you think of other data appropriate for determining the unknown? Can you change*
1292 *the unknown or the data, or both if necessary, so that the new unknown and the new*
1293 *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.

1300 Can you *check the result*? Can you check the argument? Can you derive the result
1301 differently? Can you see it at a glance?

1302 Can you use the result, or the method, from Question 1301?

Figure 5-10. How to Solve It.
G. Polya developed an approach to problem-solving in mathematics that's also

¹³⁰⁵ useful in solving problems in software design (Polya 1957).

¹⁸⁸⁸ One of the most effective guidelines is not to get stuck on a single approach.

One of the most effective guidelines is not to get stuck on one dimension of the design. UP has found this to be true in its own experience.

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).

1324

1325

1326

1327

1328

1329 ***The bad news is that, in
1330 our opinion, we will never
1331 find the philosopher's
1332 stone. We will never find
1333 a process that allows us to
design software in a***

1334 **KEY POINT**
perfectly rational way.

1335 ***The good news is that we
can fake it.***

1336 —David Parnas and Paul
1337 Clements

1338

1339

1340

1341

1342

1343

1344

1345

1346

1347

1348

1349

1350

1351

1352

1353

1354

1355

1356

1357

1358

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 effective design. 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.

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.

1359

Divide and Conquer

1360

1361

1362

1363

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!

1364

1365

1366

1367

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

1368

Top-Down and Bottom-Up Design Approaches

1369

1370

1371

1372

1373

1374

"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 non-specific design elements. As you develop the design, you increase the level of detail, identifying derived classes, collaborating classes, and other detailed design elements.

1375

1376

1377

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.

1378

1379

1380

1381

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.

1382

Argument for Top Down

1383

1384

1385

1386

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.

1387

1388

1389

1390

1391

1392

1393

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

1394 another way and see whether that works better. After several attempts, you'll
1395 have a good idea of what will work and why.

1396 How far do you decompose a program? Continue decomposing until it seems as
1397 if it would be easier to code the next level than to decompose it. Work until you
1398 become somewhat impatient at how obvious and easy the design seems. At that
1399 point, you're done. If it's not clear, work some more. If the solution is even
1400 slightly tricky for you now, it'll be a bear for anyone who works on it later.

1401 **Argument for Bottom Up**

1402 Sometimes the top-down approach is so abstract that it's hard to get started. If
1403 you need to work with something more tangible, try the bottom-up design
1404 approach. Ask yourself, "What do I know this system needs to do?"

1405 Undoubtedly, you can answer that question. You might identify a few low-level
1406 responsibilities that you can assign to concrete classes. For example, you might
1407 know that a system needs to format a particular report, compute data for that
1408 report, center its headings, display the report on the screen, print the report on a
1409 printer, and so on. After you identify several low-level responsibilities, you'll
1410 usually start to feel comfortable enough to look at the top again.

1411 Here are some things to keep in mind as you do bottom-up composition:

- 1412 • Ask yourself what you know the system needs to do.
- 1413 • Identify concrete objects and responsibilities from that question.
- 1414 • Identify common objects and group them using subsystem organization,
1415 packages, composition within objects, or inheritance, whichever is
1416 appropriate
- 1417 • Continue with the next level up, or go back to the top and try again to work
1418 down.

1419 **No Argument, Really**

1420 The key difference between top-down and bottom-up strategies is that one is a
1421 decomposition strategy and the other is a composition strategy. One starts from
1422 the general problem and breaks it into manageable pieces; the other starts with
1423 manageable pieces and builds up a general solution. Both approaches have
1424 strengths and weaknesses that you'll want to consider as you apply them to your
1425 design problems.

1426 The strength of top-down design is that it's easy. People are good at breaking
1427 something big into smaller components, and programmers are especially good at
1428 it.

1429 Another strength of top-down design is that you can defer construction details.
1430 Since systems are often perturbed by changes in construction details (for
1431 example, changes in a file structure or a report format), it's useful to know early
1432 on that those details should be hidden in classes at the bottom of the hierarchy.

1433 One strength of the bottom-up approach is that it typically results in early
1434 identification of needed utility functionality, which results in a compact, well-
1435 factored design. If similar systems have already been built, the bottom-up
1436 approach allows you to start the design of the new system by looking at pieces of
1437 the old system and asking, "What can I reuse?"

1438 A weakness of the bottom-up composition approach is that it's hard to use
1439 exclusively. Most people are better at taking one big concept and breaking it into
1440 smaller concepts than they are at taking small concepts and making one big one.
1441 It's like the old assemble-it-yourself problem: I thought I was done, so why does
1442 the box still have parts in it? Fortunately, you don't have to use the bottom-up
1443 composition approach exclusively.

1444 Another weakness of the bottom-up design strategy is that sometimes you find
1445 that you can't build a program from the pieces you've started with. You can't
1446 build an airplane from bricks, and you might have to work at the top before you
1447 know what kinds of pieces you need at the bottom.

1448 To summarize, top down tends to start simple, but sometimes low-level
1449 complexity ripples back to the top, and those ripples can make things more
1450 complex than they really needed to be. Bottom up tends to start complex, but
1451 identifying that complexity early on leads to better design of the higher-level
1452 classes—if the complexity doesn't torpedo the whole system first!

1453 In the final analysis, top-down and bottom-up design aren't competing
1454 strategies—they're mutually beneficial. Design is a heuristic process, which
1455 means that no solution is guaranteed to work every time; design contains
1456 elements of trial and error. Try a variety of approaches until you find one that
1457 works well.

1458 **Experimental Prototyping**

1459 CC2E.COM/0599
1460 Sometimes you can't really know whether a design will work until you better
1461 understand some implementation detail. You might not know if a particular
1462 database organization will work until you know whether it will meet your
1463 performance goals. You might not know whether a particular subsystem design
1464 will work until you select the specific GUI libraries you'll be working with.
1465 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.

1466 A general technique for addressing these questions at low cost is experimental
1467 prototyping. The word “prototyping” means lots of different things to different
1468 people (McConnell 1996). In this context, prototyping means writing the
1469 absolute minimum amount of throwaway code that’s needed to answer a specific
1470 design question.

1471 Prototyping works poorly when developers aren’t disciplined about writing the
1472 *absolute minimum* of code needed to answer a question. Suppose the design
1473 question is, “Can the database framework we’ve selected support the transaction
1474 volume we need?” You don’t need to write any production code to answer that
1475 question. You don’t even need to know the database specifics. You just need to
1476 know enough to approximate the problem space—number of tables, number of
1477 entries in the tables, and so on. You can then write very simple prototyping code
1478 that uses tables with names like *Table1*, *Table2*, and *Column1*, and *Column2*,
1479 populate the tables with junk data, and do your performance testing.

1480 Prototyping also works poorly when the design question is not *specific* enough.
1481 A design question like, “Will this database framework work?” does not provide
1482 enough direction for prototyping. A design question like, “Will this database
1483 framework support 1,000 transactions per second under assumptions X, Y, and
1484 Z” provides a more solid basis for prototyping.

1485 A final risk of prototyping arises when developers do not treat the code as
1486 *throwaway* code. I have found that it is not possible for people to write the
1487 absolute minimum amount of code to answer a question if they believe that the
1488 code will eventually end up in the production system. They end up implementing
1489 the system instead of prototyping. By adopting the attitude that once the question
1490 is answered the code will be thrown away, you can minimize this risk. A
1491 practical standard that can help is requiring that class names or package names
1492 for prototype code be prefixed with *prototype*. That at least makes a programmer
1493 think twice before trying to extend prototype code (Stephens 2003).

1494 Used with discipline, prototyping is the workhorse tool a designer has to combat
1495 design wickedness. Used without discipline, prototyping adds some wickedness
1496 of its own.

1497 Collaborative Design

1498 **CROSS-REFERENCE** For
1499 more details on collaborative
1500 development, see Chapter 21,
1501 “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.

- 1502 • You and your co-worker sit together in a conference room and draw design
1503 alternatives on a whiteboard.
- 1504 • You and your co-worker sit together at the keyboard and do detailed design
1505 in the programming language you're using.
- 1506 • You schedule a meeting to walk through your design ideas with one or more
1507 co-workers.
- 1508 • You schedule a formal inspection with all the structured described in
1509 Chapter TBD.
- 1510 • You don't work with anyone who can review your work, so you do some
1511 initial work, put it into a drawer, and come back to it a week later. You will
1512 have forgotten enough that you should be able to give yourself a fairly good
1513 review.

1514 If the goal is quality assurance, I tend to recommend the most structured review
1515 practice, formal inspections, for the reasons described in Chapter 21,
1516 “Collaborative Construction.” But if the goal is to foster creativity and to
1517 increase the number of design alternatives generated, not just to find errors, less
1518 structured approaches work better. After you've settled on a specific design,
1519 switching to a more formal inspection might be appropriate, depending on the
1520 nature of your project.

1521 How Much Design is Enough?

1522 Sometimes only the barest sketch of an architecture is mapped out before coding
1523 begins. Other times, teams create designs at such a level of detail that coding
1524 becomes a mostly mechanical exercise. How much design should you do before
1525 you begin coding?

1526 A related question is how formal to make the design. Do you need formal,
1527 polished design diagrams, or would digital snapshots of a few drawings on a
1528 whiteboard be enough?

1529 Deciding how much design to do before beginning full-scale coding and how
1530 much formality to use in documenting that design is hardly an exact science. The
1531 experience of the team, expected lifetime of the system, desired level of
1532 reliability, and size of project should all be considered. Table 5-2 summarizes
1533 how each of these factors influence the design approach.

1534

Table 5-2. Design Formality and Level of Detail Needed

Factor	Level of Detail in Design before Beginning 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

1535

1536

Two or more of these factors may 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.

1537

1538

1539

1540

1541

1542

1543

1544

1545

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

1546

1547

1548

1549

1550

1551

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 are those 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

1552 designs for at all. I rarely encounter projects that are suffering from having done
1553 too much design work.

1554 On the other hand, occasionally I have seen projects that are suffering from too
1555 much design *documentation*. Gresham's Law states that "programmed activity
1556 tends to drive out nonprogrammed activity" (Simon 1965). A premature rush to
1557 polish a design description is a good example of that law. I would rather see 80
1558 percent of the design effort go into creating and exploring numerous design
1559 alternatives and 20 percent go into creating less polished documentation than to
1560 have 20 percent go into creating mediocre design alternatives and 80 percent go
1561 into polishing documentation of designs that are not very good.

1562 **Capturing Your Design Work**

1563 CC2E.COM/0506 The traditional approach to capturing design work is to write up the designs in a
1564 formal design document. However, there are numerous alternative ways to
1565 capture designs that can work well on small projects, informal projects, or
1566 projects that are otherwise looking for a lightweight way to capture a design:

1567 ***Insert design documentation into the code itself***
1568 Document key design decisions in code comments, typically in the file or class
1569 header. When you couple this approach with a documentation extractor like
1570 JavaDoc, this assures that design documentation will readily available to a
1571 programmer working on a section of code, and it maximizes the chance that
1572 programmers will keep the design documentation reasonably up to date.

1573 ***Capture design discussions and decisions on a Wiki***
1574 Have your design discussions in writing, on a project wiki. This will capture
1575 your design discussions and decision automatically, albeit with the extra
1576 overhead of typing rather than talking. You can also use the Wiki to capture
1577 digital pictures to supplement the text discussion. This technique is especially
1578 useful if your development team is geographically distributed.

1579 ***Write email summaries***
1580 After a design discussion, adopt the practice of designating someone to write a
1581 summary of the discussion—especially what was decided—and send it to the
1582 project team. Archive a copy of the email in the project's public email folder.

1583 ***Use a digital camera***
1584 One common barrier to documenting designs is the tedium of creating design
1585 drawings in some popular drawing tools. But the documentation choices are not
1586 limited to the two options of "capturing the design in a nicely formatted, formal
1587 notation" vs. "no design documentation at all."

1588
1589
1590
1591
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 0.20
percent of the work required if you use a drawing tool.

1592
1593
1594
1595
1596
1597
Save design flipcharts
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 flipchart paper, you can simply archive the flipcharts 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.

1598
1599 CC2E.COM/0513
1600
1601
1602
1603
1604
1605
Use CRC 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).

1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
Create UML diagrams at appropriate levels of detail
One popular technique for diagramming designs is called UML (Unified Modeling Language), 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.

1617
1618
1619
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.

1620

1621 **People who preach
software design as a
disciplined activity spend
considerable energy
making us all feel guilty.**

1622 *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.*

1623 —P.J. Plauger

1638
1639
1640
1641
1642

CC2E.COM/0520

1643

1644
1645

1646
1647
1648
1649
1650
1651
1652
1653

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.

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 10 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, there are two amounts of design that 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

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 OO, this book introduces fundamental object-oriented concepts including objects, classes, interfaces, inheritance, polymorphism, overloading, abstract classes, aggregation and association, constructors/destructors, exceptions, and other topics.

1654 Riel, Arthur J. *Object-Oriented Design Heuristics*, Reading, Mass.: Addison
1655 Wesley, 1996. This book is easy to read and focuses on design at the class level.

1656 Plauger, P.J. *Programming on Purpose: Essays on Software Design*. Englewood
1657 Cliffs, N.J.: PTR Prentice Hall, 1993. I picked up as many tips about good
1658 software design from reading this book as from any other book I've read.
1659 Plauger is well-versed in a wide-variety of design approaches, he's pragmatic,
1660 and he's a great writer.

1661 Meyer, Bertrand. *Object-Oriented Software Construction, 2d Ed.* New York:
1662 Prentice Hall PTR, 1997. Meyer presents a forceful advocacy of hard-core
1663 object-oriented programming.

1664 Raymond, Eric S. *The Art of Unix Programming*, Boston, Mass.: Addison
1665 Wesley, 2004. This is a well-researched look at software design through Unix-
1666 colored glasses. Section 1.6 is an especially concise 12-page explanation of 17
1667 key Unix design principles.

1668 Larman, Craig, *Applying UML and Patterns: An Introduction to Object-Oriented*
1669 *Analysis and Design and the Unified Process*, 2d Ed., Englewood Cliffs, N.J.:
1670 Prentice Hall, 2001. This book is a popular introduction to object-oriented design
1671 in the context of the Unified Process. It also discusses object-oriented analysis.

1672 Software Design Theory

1673 Parnas, David L., and Paul C. Clements. "A Rational Design Process: How and
1674 Why to Fake It." *IEEE Transactions on Software Engineering* SE-12, no. 2
1675 (February 1986): 251–57. This classic article describes the gap between how
1676 programs are really designed and how you sometimes wish they were designed.
1677 The main point is that no one ever really goes through a rational, orderly design
1678 process but that aiming for it makes for better designs in the end.

1679 I'm not aware of any comprehensive treatment of information hiding. Most
1680 software-engineering textbooks discuss it briefly, frequently in the context of
1681 object-oriented techniques. The three Parnas papers listed below are the seminal
1682 presentations of the idea and are probably still the best resources on information
1683 hiding.

1684 Parnas, David L. "On the Criteria to Be Used in Decomposing Systems into
1685 Modules." *Communications of the ACM* 5, no. 12 (December 1972): 1053-58.

1686 Parnas, David L. "Designing Software for Ease of Extension and Contraction."
1687 *IEEE Transactions on Software Engineering* SE-5, no. 2 (March 1979): 128-38.

1688 Parnas, David L., Paul C. Clements, and D. M. Weiss. "The Modular Structure
1689 of Complex Systems." *IEEE Transactions on Software Engineering* SE-11, no. 3
1690 (March 1985): 259-66.

1691 **Design Patterns**

1692 Gamma, Erich, et al. *Design Patterns*, Reading, Mass.: Addison Wesley, 1995.
1693 This book by the "Gang of Four" is the seminal book on design patterns.

1694 Shalloway, Alan and James R. Trott. *Design Patterns Explained*, Boston, Mass.:
1695 Addison Wesley, 2002. This book contains an easy-to-read introduction to
1696 design patterns.

1697 **Design in General**

1698 Adams, James L. *Conceptual Blockbusting: A Guide to Better Ideas*, 4th ed.
1699 Cambridge, Mass.: Perseus Publishing, 2001. Although not specifically about
1700 software design, this book was written to teach design to engineering students at
1701 Stanford. Even if you never design anything, the book is a fascinating discussion
1702 of creative thought processes. It includes many exercises in the kinds of thinking
1703 required for effective design. It also contains a well-annotated bibliography on
1704 design and creative thinking. If you like problem solving, you'll like this book.

1705 Polya, G. *How to Solve It: A New Aspect of Mathematical Method*, 2d ed.
1706 Princeton, N.J.: Princeton University Press, 1957. This discussion of heuristics
1707 and problem solving focuses on mathematics but is applicable to software
1708 development. Polya's book was the first written about the use of heuristics in
1709 mathematical problem solving. It draws a clear distinction between the messy
1710 heuristics used to discover solutions and the tidier techniques used to present
1711 them once they've been discovered. It's not easy reading, but if you're interested
1712 in heuristics, you'll eventually read it whether you want to or not. Polya's book
1713 makes it clear that problem solving isn't a deterministic activity and that
1714 adherence to any single methodology is like walking with your feet in chains. At
1715 one time Microsoft gave this book to all its new programmers.

1716 Michalewicz, Zbigniew, and David B. Fogel, *How to Solve It: Modern*
1717 *Heuristics*, Berlin: Springer-Verlag, 2000. This is an updated treatment of
1718 Polya's book that's quite a bit easier to read and that also contains some non-
1719 mathematical examples.

1720 Simon, Herbert. *The Sciences of the Artificial*, 3d Ed. Cambridge, Mass.: MIT
1721 Press, 1996. This fascinating book draws a distinction between sciences that deal
1722 with the natural world (biology, geology, and so on) and sciences that deal with
1723 the artificial world created by humans (business, architecture, and computer
1724 science). It then discusses the characteristics of the sciences of the artificial,
1725 emphasizing the science of design. It has an academic tone and is well worth

- 1726 reading for anyone intent on a career in software development or any other
1727 “artificial” field.
- 1728 Glass, Robert L. *Software Creativity*. Englewood Cliffs, N.J.: Prentice Hall PTR,
1729 1995. Is software development controlled more by theory or by practice? Is it
1730 primarily creative or is it primarily deterministic? What intellectual qualities
1731 does a software developer need? This book contains an interesting discussion of
1732 the nature of software development with a special emphasis on design.
- 1733 Petroski, Henry. *Design Paradigms: Case Histories of Error and Judgment in*
1734 *Engineering*. Cambridge: Cambridge University Press, 1994. This book draws
1735 heavily from the field of civil engineering (especially bridge design) to explain
1736 its main argument that successful design depends at least as much upon learning
1737 from past failures as from past successes.

Standards

1739 *IEEE Std 1016-1998, Recommended Practice for Software Design Descriptions*.

1740 This document contains the IEEE-ANSI standard for software-design
1741 descriptions. It describes what should be included in a software-design
1742 document.

1743 *IEEE Std 1471-2000. Recommended Practice for Architectural Description of*
1744 *Software Intensive Systems*, Los Alamitos, CA: IEEE Computer Society Press.
1745 This document is the IEEE-ANSI guide for creating software architecture
1746 specifications.

CC2E.COM/0527

CHECKLIST: Design in Construction

Design Practices

- 1749 Have you iterated, selecting the best of several attempts rather than the first
1750 attempt?
- 1751 Have you tried decomposing the system in several different ways to see
1752 which way will work best?
- 1753 Have you approached the design problem both from the top down and from
1754 the bottom up?
- 1755 Have you prototyped risky or unfamiliar parts of the system, creating the
1756 absolute minimum amount of throwaway code needed to answer specific
1757 questions?
- 1758 Has your design been reviewed, formally or informally, by others?
- 1759 Have you driven the design to the point that its implementation seems
1760 obvious?

- 1761 Have you captured your design work using an appropriate technique such as
1762 a Wiki, email, flipcharts, digital camera, UML, CRC cards, or comments in
1763 the code itself?

1764 **Design Goals**

- 1765 Does the design adequately address issues that were identified and deferred
1766 at the architectural level?
1767 Is the design stratified into layers?
1768 Are you satisfied with the way the program has been decomposed into
1769 subsystems, packages, and classes?
1770 Are you satisfied with the way the classes have been decomposed into
1771 routines?
1772 Are classes designed for minimal interaction with each other?
1773 Are classes and subsystems designed so that you can use them in other
1774 systems?
1775 Will the program be easy to maintain?
1776 Is the design lean? Are all of its parts strictly necessary?
1777 Does the design use standard techniques and avoid exotic, hard-to-
1778 understand elements?
1779 Overall, does the design help minimize both accidental and essential
1780 complexity?

1781 **Key Points**

- 1782
 - 1783 • Software's Primary Technical Imperative is *managing complexity*. This is
1784 accomplished primarily through a design focus on simplicity.
 - 1785 • Simplicity is achieved in two general ways: minimizing the amount of
1786 essential complexity that anyone's brain has to deal with at any one time and
1787 keeping accidental complexity from proliferating needlessly.
 - 1788 • Design is heuristic. Dogmatic adherence to any single methodology hurts
1789 creativity and hurts your programs.
 - 1790 • Good design is iterative; the more design possibilities you try, the better
1791 your final design will be.
 - 1792 • Information hiding is a particularly valuable concept. Asking, "What should
1793 I hide?" settles many difficult design issues.
 - 1794 • Lots of useful, interesting information on design is available outside this
1795 book. The perspectives presented here are just the tip of the iceberg.

6

Working Classes

3

CC2E.COM/0665

Contents

- 4 6.1 Class Foundations: Abstract Data Types (ADTs)
- 5 6.2 Good Class Interfaces
- 6 6.3 Design and Implementation Issues
- 7 6.4 Reasons to Create a Class
- 8 6.5 Language-Specific Issues
- 9 6.6 Beyond Classes: Packages

Related Topics

10 Design in construction: Chapter 5

11 Software architecture: Section 3.5

12 Characteristics of high-quality routines: Chapter 7

13 The Pseudocode Programming Process: Chapter 9

14 Refactoring: Chapter 24

15 In the dawn of computing, programmers thought about programming in terms of statements. Throughout the 1970s and 1980s, programmers began thinking about programs in terms of routines. In the twenty-first century, programmers think about programming in terms of classes.

20 **KEY POINT**
21 A class is a collection of data and routines that share a cohesive, well-defined
22 responsibility. A class might also be a collection of routines that provides a
23 cohesive set of services even if no common data is involved. A key to being an
24 effective programmer is maximizing the portion of a program that you can safely
25 ignore while working on any one section of code. Classes are the primary tool
for accomplishing that objective.

26 This chapter contains a distillation of advice in creating high quality classes. If
27 you're still warming up to object-oriented concepts, this chapter might be too
28 advanced. Make sure you've read Chapter 5. Then start with Section 6.1,
29 "Abstract Data Types (ADTs)," and ease your way into the remaining sections.

30
31
32
33

If you're already familiar with class basics, you might skim Section 6.1 and then
dive into the discussion of good class interfaces in Section 6.2. The "Additional
Resources" section at the end of the chapter contains pointers to introductory
reading, advanced reading, and programming-language-specific resources.

34

6.1 Class Foundations: Abstract Data Types (ADTs)

35

36
37
38
39
40
41

An abstract data type is a collection of data and operations that work on that
data. The operations both describe the data to the rest of the program and allow
the rest of the program to change the data. The word "data" in "abstract data
type" is used loosely. An ADT might be a graphics window with all the
operations that affect it; a file and file operations; an insurance-rates table and
the operations on it; or something else.

42 **CROSS-REFERENCE** Thin
king about ADTs first and
classes second is an example
of programming *into* a
language vs. programming in
one. Section 4.3, "Your
47 Location on the Technology
Wave" and Section 34.4,
48 "Program Into Your
49 Language, Not In It."

50
51
52

Understanding ADTs is essential to understanding object-oriented programming.
Without understanding ADTs, programmers create classes that are "classes" in
name only—in reality, they are little more than convenient carrying cases for
loosely related collections of data and routines. With an understanding of ADTs,
programmers can create classes that are easier to implement initially and easier
to modify over time.

Traditionally, programming books wax mathematical when they arrive at the
topic of abstract data types. They tend to make statements like "One can think of
an abstract data type as a mathematical model with a collection of operations
defined on it." Such books make it seem as if you'd never actually use an
abstract data type except as a sleep aid.

Such dry explanations of abstract data types completely miss the point. Abstract
data types are exciting because you can use them to manipulate real-world
entities rather than low-level, implementation entities. Instead of inserting a node
into a linked list, you can add a cell to a spreadsheet, a new type of window to a
list of window types, or another passenger car to a train simulation. Tap into the
power of being able to work in the problem domain rather than at the low-level
implementation domain!

60

Example of the Need for an ADT

61 To get things started, here's an example of a case in which an ADT would be
62 useful. We'll get to the theoretical details after we have an example to talk about.

63 Suppose you're writing a program to control text output to the screen using a
64 variety of typefaces, point sizes, and font attributes (such as bold and italic). Part

65 of the program manipulates the text's fonts. If you use an ADT, you'll have a
66 group of font routines bundled with the data—the typeface names, point sizes,
67 and font attributes—they operate on. The collection of font routines and data is
68 an ADT.

69 If you're not using ADTs, you'll take an ad hoc approach to manipulating fonts.
70 For example, if you need to change to a 12-point font size, which happens to be
71 16 pixels high, you'll have code like this:

72 `currentFont.size = 16`

73 If you've built up a collection of library routines, the code might be slightly
74 more readable:

75 `currentFont.size = PointsToPixels(12)`

76 Or you could provide a more specific name for the attribute, something like

77 `currentFont.sizeInPixels = PointsToPixels(12)`

78 But what you can't do is have both `currentFont.sizeInPixels` and
79 `currentFont.sizeInPoints`, because, if both the data members are in play,
80 `currentFont` won't have any way to know which of the two it should use.

81 If you change sizes in several places in the program, you'll have similar lines
82 spread throughout your program.

83 If you need to set a font to bold, you might have code like this:

84 `currentFont.attribute = currentFont.attribute or 0x02`

85 If you're lucky, you'll have something cleaner than that, but the best you'll get
86 with an ad hoc approach is something like this:

87 `currentFont.attribute = currentFont.attribute or BOLD`

88 Or maybe something like this:

89 `currentFont.bold = True`

90 As with the font size, the limitation is that the client code is required to control
91 the data members directly, which limits how `currentFont` can be used.

92 If you program this way, you're likely to have similar lines in many places in
93 your program.

94 Benefits of Using ADTs

95 The problem isn't that the ad hoc approach is bad programming practice. It's that
96 you can replace the approach with a better programming practice that produces
97 these benefits:

98 You can hide implementation details

99
100
101
102
103
104
105
106
Hiding information about the font data type means that if the data type changes,
you can change it in one place without affecting the whole program. For
example, unless you hid the implementation details in an ADT, changing the
data type from the first representation of bold to the second would entail
changing your program in every place in which bold was set rather than in just
one place. Hiding the information also protects the rest of the program if you
decide to store data in external storage rather than in memory or to rewrite all the
font-manipulation routines in another language.

107 Changes don't affect the whole program

108
109
110
If fonts need to become richer and support more operations (such as switching to
small caps, superscripts, strikethrough, and so on), you can change the program
in one place. The change won't affect the rest of the program.

111 You can make the interface more informative

112
113
114
115
116
Code like `currentFont.size = 16` is ambiguous because 16 could be a size in
either pixels or points. The context doesn't tell you which is which. Collecting
all similar operations into an ADT allows you to define the entire interface in
terms of points, or in terms of pixels, or to clearly differentiate between the two,
which helps avoid confusing them.

117 It's easier to improve performance

118
119
If you need to improve font performance, you can recode a few well-defined
routines rather than wading through an entire program.

120 The program is more obviously correct

121
122
123
124
125
126
127
128
You can replace the more tedious task of verifying that statements like
`currentFont.attribute = currentFont.attribute or 0x02` are correct with the easier
task of verifying that calls to `currentFont.BoldOn()` are correct. With the first
statement, you can have the wrong structure name, the wrong field name, the
wrong logical operation (a logical *and* instead of *or*), or the wrong value for the
attribute (`0x20` instead of `0x02`). In the second case, the only thing that could
possibly be wrong with the call to `currentFont.BoldOn()` is that it's a call to the
wrong routine name, so it's easier to see whether it's correct.

129 The program becomes more self-documenting

130
131
132
You can improve statements like `currentFont.attribute or 0x02` by replacing
`0x02` with *BOLD* or whatever `0x02` represents, but that doesn't compare to the
readability of a routine call such as `currentFont.BoldOn()`.

133 **HARD DATA**
134
135
136
Woodfield, Dunsmore, and Shen conducted a study in which graduate and senior
undergraduate computer-science students answered questions about two
programs—one that was divided into eight routines along functional lines and
one that was divided into eight abstract-data-type routines (1981). Students using

137 the abstract-data-type program scored over 30 percent higher than students using
138 the functional version.

139 ***You don't have to pass data all over your program***
140 In the examples just presented, you have to change *currentFont* directly or pass
141 it to every routine that works with fonts. If you use an abstract data type, you
142 don't have to pass *currentFont* all over the program and you don't have to turn it
143 into global data either. The ADT has a structure that contains *currentFont*'s data.
144 The data is directly accessed only by routines that are part of the ADT. Routines
145 that aren't part of the ADT don't have to worry about the data.

146 ***You're able to work with real-world entities rather than with low-level***
147 ***implementation structures***

148 You can define operations dealing with fonts so that most of the program
149 operates solely in terms of fonts rather than in terms of array accesses, structure
150 definitions, and *True* and *False* booleans.

151 In this case, to define an abstract data type, you'd define a few routines to
152 control fonts—perhaps these:

```
153     currentFont.SetSizeInPoints( sizeInPoints )  
154     currentFont.SetSizeInPixels( sizeInPixels )  
155     currentFont.BoldOn()  
156     currentFont.BoldOff()  
157     currentFont.ItalicOn()  
158     currentFont.ItalicOff()  
159     currentFont.SetTypeFace( faceName )
```

160 **KEY POINT**
161 The code inside these routines would probably be short—it would probably be
162 similar to the code you saw in the ad hoc approach to the font problem earlier.
163 The difference is that you've isolated font operations in a set of routines. That
164 provides a better level of abstraction for the rest of your program to work with
fonts, and it gives you a layer of protection against changes in font operations.

165 More Examples of ADTs

166 Here are a few more examples of ADTs:

167 Suppose you're writing software that controls the cooling system for a nuclear
168 reactor. You can treat the cooling system as an abstract data type by defining the
169 following operations for it:

```
170     coolingSystem.Temperature()  
171     coolingSystem.SetCirculationRate( rate )  
172     coolingSystem.OpenValve( valveNumber )  
173     coolingSystem.CloseValve( valveNumber )
```

174 The specific environment would determine the code written to implement each
 175 of these operations. The rest of the program could deal with the cooling system
 176 through these functions and wouldn't have to worry about internal details of
 177 data-structure implementations, data-structure limitations, changes, and so on.

178 Here are more examples of abstract data types and likely operations on them:

<i>Cruise Control</i>	<i>Blender</i>	<i>Fuel Tank</i>
Set speed	Turn on	Fill tank
Get current settings	Turn off	Drain tank
Resume former speed	Set speed	Get tank capacity
Deactivate	Start "Insta-Pulverize"	Get tank status
	Stop "Insta-Pulverize"	
<i>Set of Help Screens</i>		<i>Stack</i>
Add help topic	<i>Menu</i>	Initialize stack
Remove help topic	Start new menu	Push item onto stack
Set current help topic	Delete menu	Pop item from stack
Display help screen	Add menu item	Read top of stack
Remove help display	Remove menu item	
Display help index	Activate menu item	<i>File</i>
Back up to previous screen	Deactivate menu item	Open file
	Display menu	Read file
<i>List</i>		<i>Write file</i>
Initialize list	Get menu choice	Set current file location
Insert item in list		Close file
Remove item from list	<i>Pointer</i>	
Read next item from list	Get pointer to new memory	<i>Elevator</i>
	Dispose of memory from existing pointer	Move up one floor
<i>Light</i>		Move down one floor
Turn on	Change amount of memory allocated	
		Move to specific floor

Turn off

Report current floor

Return to home floor

179 Yon can derive several guidelines from a study of these examples:

180 ***Build or use typical low-level data types as ADTs, not as low-level data***
181 ***types***

182 Most discussions of ADTs focus on representing typical low-level data types as
183 ADTs. As you can see from the examples, you can represent a stack, a list, and a
184 queue, as well as virtually any other typical data type, as an ADTs.

185 The question you need to ask is, What does this stack, list, or queue represent? If
186 a stack represents a set of employees, treat the ADT as employees rather than as
187 a stack. If a list represents a set of billing records, treat it as billing records rather
188 than a list. If a queue represents cells in a spreadsheet, treat it as a collection of
189 cells rather than a generic item in a queue. Treat yourself to the highest possible
190 level of abstraction.

191 ***Treat common objects such as files as ADTs***

192 Most languages include a few abstract data types that you're probably familiar
193 with but might not think of as ADTs. File operations are a good example. While
194 writing to disk, the operating system spares you the grief of positioning the
195 read/write head at a specific physical address, allocating a new disk sector when
196 you exhaust an old one, and checking for binary error codes. The operating
197 system provides a first level of abstraction and the ADTs for that level. High-
198 level languages provide a second level of abstraction and ADTs for that higher
199 level. A high-level language protects you from the messy details of generating
200 operating-system calls and manipulating data buffers. It allows you to treat a
201 chunk of disk space as a "file."

202 You can layer ADTs similarly. If you want to use an ADT at one level that offers
203 data-structure level operations (like pushing and popping a stack), that's fine.
204 You can create another level on top of that one that works at the level of the real-
205 world problem.

206 ***Treat even simple items as ADTs***

207 You don't have to have a formidable data type to justify using an abstract data
208 type. One of the ADTs in the example list is a light that supports only two
209 operations—turning it on and turning it off. You might think that it would be a
210 waste to isolate simple "on" and "off" operations in routines of their own, but
211 even simple operations can benefit from the use of ADTs. Putting the light and
212 its operations into an ADT makes the code more self-documenting and easier to
213 change, confines the potential consequences of changes to the *TurnLightOn()*

214 and *TurnLightOff()* routines, and reduces the amount of data you have to pass
215 around.

216 ***Refer to an ADT independently of the medium it's stored on***
217 Suppose you have an insurance-rates table that's so big that it's always stored on
218 disk. You might be tempted to refer to it as a "rate file" and create access
219 routines such as *rateFile.Read()*. When you refer to it as a file, however, you're
220 exposing more information about the data than you need to. If you ever change
221 the program so that the table is in memory instead of on disk, the code that refers
222 to it as a file will be incorrect, misleading, and confusing. Try to make the names
223 of classes and access routines independent of how the data is stored, and refer to
224 the abstract data type, like the insurance-rates table, instead. That would give
225 your class and access routine names like *rateTable.Read()* or simply
226 *rates.Read()*.

227 **Handling Multiple Instances of Data with ADTs in 228 Non-OO Environments**

229 Object-oriented languages provide automatic support for handling multiple
230 instances of an ADT. If you've worked exclusively in object-oriented
231 environments and have never had to handle the implementation details of
232 multiple instances yourself, count your blessings! (You can also move on to the
233 next section, "ADTs and Classes")

234 If you're working in a non-object oriented environment such as C, you will have
235 to build support for multiple instances manually. In general, that means
236 including services for the ADT to create and delete instances and designing the
237 ADT's other services so that they can work with multiple instances.

238 The font ADT originally offered these services:

```
239     currentFont.SetSize( sizeInPoints )  
240     currentFont.BoldOn()  
241     currentFont.BoldOff()  
242     currentFont.ItalicOn()  
243     currentFont.ItalicOff()  
244     currentFont.SetTypeFace( faceName )
```

245 In a non-OO environment, these functions would not be attached to a class, and
246 would look more like this:

```
247     SetCurrentFontSize( sizeInPoints )  
248     SetCurrentFontBoldOn()  
249     SetCurrentFontBoldOff()  
250     SetCurrentFontItalicOn()  
251     SetCurrentFontItalicOff()  
252     SetCurrentFontTypeFace( faceName )
```

253 If you want to work with more than one font at a time, you'll need to add
254 services to create and delete font instances—maybe these:

```
255 CreateFont( fontId )
256 DeleteFont( fontId )
257 SetCurrentFont( fontId )
```

258 The notion of a *fontId* has been added as a way to keep track of multiple fonts as
259 they're created and used. For other operations, you can choose from among three
260 ways to handle the ADT interface:

261 ***Option 1: Use implicit instances (with great care)***

262 Design a new service to call to make a specific font instance the current one—
263 something like *SetCurrentFont(fontId)*. Setting the current font makes all other
264 services use the current font when they're called. If you use this approach, you
265 don't need *fontId* as a parameter to the other services. For simple applications
266 this can streamline use of multiple instances. For complex applications, this
267 system-wide dependence on state means that you must keep track of the current
268 font instance throughout code that uses the *Font* functions. Complexity tends to
269 proliferate, and for applications of any size, there are better alternatives.

270 ***Option 2: Explicitly identify instances each time you use ADT services***

271 In this case, you don't have the notion of a "current font." You pass *fontId* to
272 each routine that manipulates fonts. The *Font* functions keep track of any
273 underlying data, and the client code needs to keep track only of the *fontId*. This
274 requires adding *fontId* as a parameter to each font routine.

275 ***Option 3: Explicitly provide the data used by the ADT services***

276 In this approach, you declare the data that the ADT uses within each routine that
277 uses an ADT service. In other words, you create a *Font* data type that you pass to
278 each of the ADT service routines. You must design the ADT service routines so
279 that they use the *Font* data that's passed to them each time they're called. The
280 client code doesn't need a font ID if you use this approach because it keeps track
281 of the font data itself. (Even though the data is available directly from the *Font*
282 data type, you should access it only with the ADT service routines. This is called
283 keeping the structure "closed.")

284 The advantage of this approach is that the ADT service routines don't have to
285 look up font information based on a font ID. The disadvantage is that it exposes
286 font data to the rest of the program, which increases the likelihood that client
287 code will make use of the ADT's implementation details that should have
288 remained hidden within the ADT.

289 Inside the abstract data type, you'll have a wealth of options for handling
290 multiple instances, but outside, this sums up the choices if you're working in a
291 non-object oriented language.

292

293

294

295

296

297

298

299

300

301

302

303

304

305

306

307

308

309

310

311

312 **CROSS-REFERENCE** Cod
e samples in this book are
313 formatted using a coding
314 convention that emphasizes
315 similarity of styles across
316 multiple languages. For
317 details on the convention
318 (and discussions about
319 multiple coding styles), see
320 “Mixed-Language
321 Programming
322 Considerations” in Section
11.4.

323

324

325

326

327

328

ADTs and Classes

Abstract data types form the foundation for the concept of classes. In languages that support classes, you can implement each abstract data type in its own class. Classes usually involve the additional concepts of inheritance and polymorphism. One way of thinking of a class is as an abstract data type plus inheritance and polymorphism.

6.2 Good Class Interfaces

The first and probably most important step in creating a high quality class is creating a good interface. This consists of creating a good abstraction for the interface to represent and ensuring the details remain hidden behind the abstraction.

Good Abstraction

As “Form Consistent Abstractions” in Section 5.3 discussed, abstraction is the ability to view a complex operation in a simplified form. A class interface provides an abstraction of the implementation that’s hidden behind the interface. The class’s interface should offer a group of routines that clearly belong together.

You might have a class that implements an employee. It would contain data describing the employee’s name, address, phone number, and so on. It would offer services to initialize and use an employee. Here’s how that might look.

C++ Example of a Class Interface that Presents a Good Abstraction

```
class Employee {
public:
    // public constructors and destructors
    Employee();
    Employee(
        FullName name,
        String address,
        String workPhone,
        String homePhone,
        TaxId taxIdNumber,
        JobClassification jobClass
    );
    virtual ~Employee();

    // public routines
    FullName Name();
}
```

```
329     String Address();
330     String WorkPhone();
331     String HomePhone();
332     TaxId TaxIdNumber();
333     JobClassification GetJobClassification();
334     ...
335 private:
336     ...
337 }
```

Internally, this class might have additional routines and data to support these services, but users of the class don't need to know anything about them. The class interface abstraction is great because every routine in the interface is working toward a consistent end.

A class that presents a poor abstraction would be one that contained a collection of miscellaneous functions. Here's an example:

CODING HORROR

```
344
345 class Program {
346 public:
347     ...
348     // public routines
349     void InitializeCommandStack();
350     void PushCommand( Command &command );
351     Command PopCommand();
352     void ShutdownCommandStack();
353     void InitializeReportFormatting();
354     void FormatReport( Report &report );
355     void PrintReport( Report &report );
356     void InitializeGlobalData();
357     void ShutdownGlobalData();
358     ...
359 private:
360     ...
361 }
```

Suppose that a class contains routines to work with a command stack, format reports, print reports, and initialize global data. It's hard to see any connection among the command stack and report routines or the global data. The class interface doesn't present a consistent abstraction, so the class has poor cohesion. The routines should be reorganized into more-focused classes, each of which provides a better abstraction in its interface.

If these routines were part of a "Program" class, they could be revised to present a consistent abstraction.

370

```

371 class Program {
372 public:
373     ...
374     // public routines
375     void InitializeProgram();
376     void ShutDownProgram();
377     ...
378 private:
379     ...
380 }
```

The cleanup of this interface assumes that some of these routines were moved to other, more appropriate classes and some were converted to private routines used by *InitializeProgram()* and *ShutDownProgram()*.

This evaluation of class abstraction is based on the class's collection of public routines, that is, its class interface. The routines inside the class don't necessarily present good individual abstractions just because the overall class does, but they need to be designed to present good abstractions, too. For guidelines on that, see Section 7.2, "Design at the Routine Level."

The pursuit of good, abstract interfaces gives rise to several guidelines for creating class interfaces.

Present a consistent level of abstraction in the class interface

A good way to think about a class is as the mechanism for implementing the abstract data types (ADTs) described in Section 6.1. Each class should implement one and only one ADT. If you find a class implementing more than one ADT, or if you can't determine what ADT the class implements, it's time to reorganize the class into one or more well-defined ADTs.

Here's an example of a class that presents an interface that's inconsistent because its level of abstraction is not uniform:

CODING HORROR

400
401
402
403
404 *The abstraction of these*
405 *routines is at the "employee"*
406 *level.*
407 *The abstraction of these*
408 *routines is at the "list" level.*
409

C++ Example of a Class Interface that Presents a Better Abstraction

```

371 class Program {
372 public:
373     ...
374     // public routines
375     void InitializeProgram();
376     void ShutDownProgram();
377     ...
378 private:
379     ...
380 }
```

The cleanup of this interface assumes that some of these routines were moved to other, more appropriate classes and some were converted to private routines used by *InitializeProgram()* and *ShutDownProgram()*.

This evaluation of class abstraction is based on the class's collection of public routines, that is, its class interface. The routines inside the class don't necessarily present good individual abstractions just because the overall class does, but they need to be designed to present good abstractions, too. For guidelines on that, see Section 7.2, "Design at the Routine Level."

The pursuit of good, abstract interfaces gives rise to several guidelines for creating class interfaces.

Present a consistent level of abstraction in the class interface

A good way to think about a class is as the mechanism for implementing the abstract data types (ADTs) described in Section 6.1. Each class should implement one and only one ADT. If you find a class implementing more than one ADT, or if you can't determine what ADT the class implements, it's time to reorganize the class into one or more well-defined ADTs.

Here's an example of a class that presents an interface that's inconsistent because its level of abstraction is not uniform:

C++ Example of a Class Interface with Mixed Levels of Abstraction

```

400 class EmployeeList: public ListContainer {
401 public:
402     ...
403     // public routines
404     void AddEmployee( Employee &employee );
405     void RemoveEmployee( Employee &employee );
406
407     Employee NextItemInList( Employee &employee );
408     Employee FirstItem( Employee &employee );
409     Employee LastItem( Employee &employee );
```

```
410 ...
411 private:
412 ...
413 }
414
415 This class is presenting two ADTs: an Employee and a ListContainer. This sort
416 of mixed abstraction commonly arises when a programmer uses a container class
417 or other library classes for implementation and doesn't hide the fact that a library
418 class is used. Ask yourself whether the fact that a container class is used should
419 be part of the abstraction. Usually that's an implementation detail that should be
hidden from the rest of the program, like this:
```

C++ Example of a Class Interface with Consistent Levels of Abstraction

```
420
421 class EmployeeList {
422 public:
423 ...
424 // public routines
425     The abstraction of all these
426         routines is now at the
427             "employee" level.
428 void AddEmployee( Employee &employee );
429 void RemoveEmployee( Employee &employee );
430 Employee NextEmployee( Employee &employee );
431 Employee FirstEmployee( Employee &employee );
432 Employee LastEmployee( Employee &employee );
433 ...
434 private:
435     ListContainer m_EmployeeList;
436 ...
437 }
```

Programmers might argue that inheriting from *ListContainer* is convenient because it supports polymorphism, allowing an external search or sort function that takes a *ListContainer* object.

That argument fails the main test for inheritance, which is, Is inheritance used only for “is a” relationships? To inherit from *ListContainer* would mean that *EmployeeList* “is a” *ListContainer*, which obviously isn’t true. If the abstraction of the *EmployeeList* object is that it can be searched or sorted, that should be incorporated as an explicit, consistent part of the class interface.

If you think of the class’s public routines as an air lock that keeps water from getting into a submarine, inconsistent public routines are leaky panels in the class. The leaky panels might not let water in as quickly as an open air lock, but if you give them enough time, they’ll still sink the boat. In practice, this is what happens when you mix levels of abstraction. As the program is modified, the mixed levels of abstraction make the program harder and harder to understand, and it gradually degrades until it becomes unmaintainable.

KEY POINT

450 ***Be sure you understand what abstraction the class is implementing***
451 Some classes are similar enough that you must be careful to understand which
452 abstraction the class interface should capture. I once worked on a program that
453 needed to allow information to be edited in a table format. We wanted to use a
454 simple grid control, but the grid controls that were available didn't allow us to
455 color the data-entry cells, so we decided to use a spreadsheet control that did
456 provide that capability.

457 The spreadsheet control was far more complicated than the grid control,
458 providing about 150 routines to the grid control's 15. Since our goal was to use a
459 grid control, not a spreadsheet control, we assigned a programmer to write a
460 wrapper class to hide the fact that we were using a spreadsheet control as a grid
461 control. The programmer grumbled quite a bit about unnecessary overhead and
462 bureaucracy, went away, and came back a couple days later with a wrapper class
463 that faithfully exposed all 150 routines of the spreadsheet control.

464 This was not what was needed. We wanted a grid-control interface that
465 encapsulate the fact that, behind the scenes, we were using a much more
466 complicated spreadsheet control. The programmer should have exposed just the
467 15 grid control routines plus a 16th routine that supported cell coloring. By
468 exposing all 150 routines, the programmer created the possibility that, if we ever
469 wanted to change the underlying implementation, we could find ourselves
470 supporting 150 public routines. The programmer failed to achieve the
471 encapsulation we were looking for, as well as creating a lot more work for
472 himself than necessary.

473 Depending on specific circumstances, the right abstraction might be either a
474 spreadsheet control or a grid control. When you have to choose between two
475 similar abstractions, make sure you choose the right one.

476 ***Provide services in pairs with their opposites***
477 Most operations have corresponding, equal, and opposite operations. If you have
478 an operation that turns a light on, you'll probably need one to turn it off. If you
479 have an operation to add an item to a list, you'll probably need one to delete an
480 item from the list. If you have an operation to activate a menu item, you'll
481 probably need one to deactivate an item. When you design a class, check each
482 public routine to determine whether you need its complement. Don't create an
483 opposite gratuitously, but do check to see whether you need one.

484 ***Move unrelated information to another class***
485 In some cases, you'll find that half a class's routines work with half the class's
486 data, and half the routines work with the other half of the data. In such a case,
487 you really have two classes masquerading as one. Break them up!

488

489 **CROSS-REFERENCE** For
490 more suggestions about how
491 to preserve code quality as
492 code is modified, See
Chapter 24, "Refactoring."

493 **CODING HORROR**

494

495

496

497

498

499

500

501

502

503

504

505

506

507

508

509

510

511

512

513

514

515

516

517

518

519

520

521

522

523

524

525

526

527

528

Beware of erosion of the interface's abstraction under modification

As a class is modified and extended, you often discover additional functionality that's needed, that doesn't quite fit with the original class interface, but that seems too hard to implement any other way. For example, in the *Employee* class, you might find that the class evolves to look like this:

C++ Example of a Class Interface that's Eroding Under Maintenance

```
class Employee {  
public:  
    ...  
    // public routines  
    FullName GetName();  
    Address GetAddress();  
    PhoneNumber GetWorkPhone();  
    ...  
    Boolean IsJobClassificationValid( JobClassification jobClass );  
    Boolean IsZipCodeValid( Address address );  
    Boolean IsPhoneNumberValid( PhoneNumber phoneNumber );  
  
    SqlDataReader GetQueryToCreateNewEmployee();  
    SqlDataReader GetQueryToModifyEmployee();  
    SqlDataReader GetQueryToRetrieveEmployee();  
    ...  
private:  
    ...  
}
```

What started out as a clean abstraction in an earlier code sample has evolved into a hodgepodge of functions that are only loosely related. There's no logical connection between employees and routines that check zip codes, phone numbers, or job classifications. The routines that expose SQL query details are at a much lower level of abstraction than the *Employee* class, and they break the *Employee* abstraction.

Don't add public members that are inconsistent with the interface abstraction

Each time you add a routine to a class interface, ask, "Is this routine consistent with the abstraction provided by the existing interface?" If not, find a different way to make the modification, and preserve the integrity of the abstraction.

Consider abstraction and cohesion together

The ideas of abstraction and cohesion are closely related—a class interface that presents a good abstraction usually has strong cohesion. Classes with strong cohesion tend to present good abstractions, although that relationship is not as strong.

529
530
531
532

I have found that focusing on the abstraction presented by the class interface
tends to provide more insight into class design than focusing on class cohesion.
If you see that a class has weak cohesion and aren't sure how to correct it, ask
yourself whether the class presents a consistent abstraction instead.

533

534 **CROSS-REFERENCE** For
535 more on encapsulation, see
536 "Encapsulate Implementation
537 Details" in Section 5.3.

538
539
540

541 *The single most
542 important factor that
543 distinguishes a well-
544 designed module from a
545 poorly designed one is the
546 degree to which the
547 module hides its internal
548 data and other
549 implementation details
from other modules.*

550 —Joshua Bloch

552

Good Encapsulation

As Section 5.3 discussed, encapsulation is a stronger concept than abstraction.
Abstraction helps to manage complexity by providing models that allow you to
ignore implementation details. Encapsulation is the enforcer that prevents you
from looking at the details even if you want to.

The two concepts are related because, without encapsulation, abstraction tends to
break down. In my experience either you have both abstraction and
encapsulation, or you have neither. There is no middle ground.

Minimize accessibility of classes and members

Minimizing accessibility is one of several rules that are designed to encourage
encapsulation. If you're wondering whether a specific routine should be public,
private, or protected, one school of thought is that you should favor the strictest
level of privacy that's workable (Meyers 1998, Bloch 2001). I think that's a fine
guideline, but I think the more important guideline is, "What best preserves the
integrity of the interface abstraction?" If exposing the routine is consistent with
the abstraction, it's probably fine to expose it. If you're not sure, hiding more is
generally better than hiding less.

Don't expose member data in public

Exposing member data is a violation of encapsulation and limits your control
over the abstraction. As Arthur Riel points out, a *Point* class that exposes

553 float x;
554 float y;
555 float z;

is violating encapsulation because client code is free to monkey around with
Point's data, and *Point* won't necessarily even know when its values have been
changed (Riel 1996). However, a *Point* class that exposes

559 float X();
560 float Y();
561 float Z();
562 void SetX(float x);
563 void SetY(float y);
564 void SetZ(float z);

is maintaining perfect encapsulation. You have no idea whether the underlying
implementation is in terms of *floats* *x*, *y*, and *z*, whether *Point* is storing those

567 items as *doubles* and converting them to *floats*, or whether Point is storing them
568 on the moon and retrieving them from a satellite in outer space.

569 ***Don't put private implementation details in a class's interface***

570 With true encapsulation, programmers would not be able to see implementation
571 details at all. They would be hidden both figuratively and literally.

572 In popular languages like C++, however, the structure of the language requires
573 programmers to disclose implementation details in the class interface. Here's an
574 example:

575 **C++ Example of Inadvertently Exposing a Class's Implementation
576 Details**

```
577 class Employee {  
578     public:  
579         ...  
580         Employee(  
581             FullName name,  
582             String address,  
583             String workPhone,  
584             String homePhone,  
585             TaxId taxIdNumber,  
586             JobClassification jobClass  
587         );  
588         ...  
589         FullName Name();  
590         String Address();  
591         ...  
592     private:  
593         Here are the exposed  
594         implementation details.  
595         String m_Name;  
596         String m_Address;  
597         int m_jobClass;  
598         ...  
599     }
```

600 Including *private* declarations in the class header file might seem like a small
601 transgression, but it encourages programmers to examine the implementation
602 details. In this case, the client code is intended to use the *Address* type for
603 addresses, but the header file exposes the implementation detail that addresses
604 are stored as *Strings*.

605 As the writer of a class in C++, there isn't much you can do about this without
606 going to great lengths that usually add more complexity than they're worth. As
the *reader* of a class, however, you can resist the urge to comb through the
607 *private* section of the class interface looking for implementation clues.

607
608
609
610
611612
613614
615
616
617
618619
620
621
622
623624
625
626
627
628
629
630
631
632633 *It ain't abstract if you
634 have to look at the
635 underlying
636 implementation to
637 understand what's going
on.*

638 —P.J. Plauger

640
641
642
643***Don't make assumptions about the class's users***

A class should be designed and implemented to adhere to the contract implied by the class interface. It shouldn't make any assumptions about how that interface will or won't be used, other than what's documented in the interface. Comments like this are an indication that a class is more aware of its users than it should be:

```
-- initialize x, y, and z to 1.0 because DerivedClass blows  
-- up if they're initialized to 0.0
```

Avoid friend classes

In a few circumstances such as the State pattern, friend classes can be used in a disciplined way that contributes to managing complexity (Gamma et al 1995). But, in general, friend classes violate encapsulation. They expand the amount of code you have to think about at any one time, increasing complexity.

Don't put a routine into the public interface just because it uses only public routines

The fact that a routine uses only public routines is not a very significant consideration. Instead, ask whether exposing the routine would be consistent with the abstraction presented by the interface.

Favor read-time convenience to write-time convenience

Code is read far more times than it's written, even during initial development. Favoring a technique that speeds write-time convenience at the expense of read-time convenience is a false economy. This is especially applicable to creation of class interfaces. Even if a routine doesn't quite fit the interface's abstraction, sometimes it's tempting to add a routine to an interface that would be convenient for the particular client of a class that you're working on at the time. But adding that routine is the first step down a slippery slope, and it's better not to take even the first step.

Be very, very wary of semantic violations of encapsulation

At one time I thought that when I learned how to avoid syntax errors I would be home free. I soon discovered that learning how to avoid syntax errors had merely bought me a ticket to a whole new theater of coding errors—most of which were more difficult to diagnose and correct than the syntax errors.

The difficulty of semantic encapsulation compared to syntactic encapsulation is similar. Syntactically, it's relatively easy to avoid poking your nose into the internal workings of another class just by declaring the class's internal routines and data *private*. Achieving semantic encapsulation is another matter entirely. Here are some examples of the ways that a user of a class can break encapsulation semantically:

- 644 • Not calling Class A's *Initialize()* routine because you know that Class A's
645 *PerformFirstOperation()* routine calls it automatically.
646 • Not calling the *database.Connect()* routine before you call
647 *employee.Retrieve(database)* because you know that the
648 *employee.Retrieve()* function will connect to the database if there isn't
649 already a connection.
650 • Not calling Class A's *Terminate()* routine because you know that Class A's
651 *PerformFinalOperation()* routine has already called it.
652 • Using a pointer or reference to *ObjectB* created by *ObjectA* even after
653 *ObjectA* has gone out of scope, because you know that *ObjectA* keeps
654 *ObjectB* in *static* storage, and *ObjectB* will still be valid.
655 • Using ClassB's *MAXIMUM_ELEMENTS* constant instead of using
656 *ClassA.MAXIMUM_ELEMENTS*, because you know that they're both equal
657 to the same value.

658 **KEY POINT**

659 The problem with each of these examples is that they make the client code
660 dependent not on the class's public interface, but on its private implementation.
661 Anytime you find yourself looking at a class's implementation to figure out how
662 to use the class, you're not programming to the interface; you're programming
663 *through* the interface *to* the implementation. If you're programming through the
664 interface, encapsulation is broken, and once encapsulation starts to break down,
abstraction won't be far behind.

665 If you can't figure out how to use a class based solely on its interface
666 documentation, the right response is *not* to pull up the source code and look at
667 the implementation. That's good initiative but bad judgment. The right response
668 is to contact the author of the class and say, "I can't figure out how to use this
669 class." The right response on the class-author's part is *not* to answer your
670 question face to face. The right response for the class author is to check out the
671 class-interface file, modify the class-interface documentation, check the file back
in, and then say, "See if you can understand how it works now." You want this
672 dialog to occur in the interface code itself so that it will be preserved for future
673 programmers. You don't want the dialog to occur solely in your own mind,
674 which will bake subtle semantic dependencies into the client code that uses the
675 class. And you don't want the dialog to occur interpersonally so that it benefits
676 only your code but no one else's.
677

678 ***Watch for coupling that's too tight***

679 "Coupling" refers to how tight the connection is between two classes. In general,
680 the looser the connection, the better. Several general guidelines flow from this
681 concept:

- 682 • Minimize accessibility of classes and members

- 683 • Avoid *friend* classes, because they're tightly coupled
684 • Avoid making data *protected* in a base class because it allows derived
685 classes to be more tightly coupled to the base class
686 • Avoid exposing member data in a class's public interface
687 • Be wary of semantic violations of encapsulation
688 • Observe the Law of Demeter (discussed later in this chapter)
- 689 Coupling goes hand in glove with abstraction and encapsulation. Tight coupling
690 occurs when an abstraction is leaky, or when encapsulation is broken. If a class
691 offers an incomplete set of services, other routines might find they need to read
692 or write its internal data directly. That opens up the class, making it a glass box
693 instead of a black box, and virtually eliminates the class's encapsulation.

6.3 Design and Implementation Issues

694 Defining good class interfaces goes a long way toward creating a high-quality
695 program. The internal class design and implementation are also important. This
696 section discusses issues related to containment, inheritance, member functions
697 and data, class coupling, constructors, and value-vs.-reference objects.

Containment (“has a” relationships)

700 **KEY POINT**
701 Containment is the simple idea that a class contains a primitive data element or
702 object. A lot more is written about inheritance than about containment, but that's
703 because inheritance is more tricky and error prone, not because it's better.
Containment is the work-horse technique in object-oriented programming.

Implement “has a” through containment

704 One way of thinking of containment is as a “has a” relationship. For example, an
705 employee “has a” name, “has a” phone number, “has a” tax ID, and so on. You
706 can usually accomplish this by making the name, phone number, or tax ID
707 member data of the *Employee* class.

Implement “has a” through private inheritance as a last resort

710 In some instances you might find that you can't achieve containment through
711 making one object a member of another. In that case, some experts suggest
712 privately inheriting from the contained object (Meyers 1998). The main reason
713 you would do that is to set up the containing class to access protected member
714 functions or data of the class that's contained. In practice, this approach creates
715 an overly cozy relationship with the ancestor class and violates encapsulation. It
716 tends to point to design errors that should be resolved some way other than
717 through private inheritance.

718 *Be critical of classes that contain more than about seven members*
719
720
721
722
723
724
725

The number “7±2” has been found to be a number of discrete items a person can remember while performing other tasks (Miller 1956). If a class contains more than about seven data members, consider whether the class should be decomposed into multiple smaller classes (Riel 1996). You might err more toward the high end of 7±2 if the data members are primitive data types like integers and strings; more toward the lower end of 7±2 if the data members are complex objects.

726 **Inheritance (“is a” relationships)**
727
728
729
730
731

Inheritance is the complex idea that one class is a specialization of another class. Inheritance is perhaps the most distinctive attribute of object-oriented programming, and it should be used sparingly and with great caution. A great many of the problems in modern programming arise from overly enthusiastic use of inheritance.

732
733
734
735

The purpose of inheritance is to create simpler code by defining a base class that specifies common elements of two or more derived classes. The common elements can be routine interfaces, implementations, data members, or data types.

736 When you decide to use inheritance, you have to make several decisions:

- 737
738
739
- For each member routine, will the routine be visible to derived classes? Will it have a default implementation? Will the default implementation be overridable?
 - For each data member (including variables, named constants, enumerations, and so on), will the data member be visible to derived classes?

740
741
742

The following subsections explain the ins and outs of making these decisions.

743
744 *The single most
745 important rule in object-
746 oriented programming
747 with C++ is this: public
748 inheritance means “isa.”*
749 *Commit this rule to
750 memory.*
751 *—Scott Meyers*

752 *Design and document for inheritance or prohibit it*

753 Inheritance adds complexity to a program, and, as such, it is a dangerous
754 technique. As Java guru Joshua Bloch says, “design and document for
755 inheritance, or prohibit it.” If a class isn’t designed to be inherited from, make its
756 members non-*virtual* in C++, *final* in Java, or non *overridable* in Visual Basic so
757 that you can’t inherit from it.

758 *Adhere to the Liskov Substitution Principle*

759 In one of object-oriented programming’s seminal papers, Barbara Liskov argued
760 that you shouldn’t inherit from a base class unless the derived class truly “is a”
761 more specific version of the base class (Liskov 1988). Andy Hunt and Dave
762 Thomas suggest a good litmus test for this: “Subclasses must be usable through
763 the base class interface without the need for the user to know the difference”
764 (Hunt and Thomas 2000).

765 In other words, all the routines defined in the base class should mean the same
766 thing when they’re used in each of the derived classes.

767 If you have a base class of *Account*, and derived classes of *CheckingAccount*,
768 *SavingsAccount*, and *AutoLoanAccount*, a programmer should be able to invoke
769 any of the routines derived from *Account* on any of *Account*’s subtypes without
770 caring about which subtype a specific account object is.

771 If a program has been written so that the Liskov Substitution Principle is true,
772 inheritance is a powerful tool for reducing complexity because a programmer can
773 focus on the generic attributes of an object without worrying about the details. If,
774 a programmer must be constantly thinking about semantic differences in subclass
775 implementations, then inheritance is increasing complexity rather than reducing
776 it. Suppose a programmer has to think, “If I call the *InterestRate()* routine on
777 *CheckingAccount* or *SavingsAccount*, it returns the interest the bank pays, but if I
778 call *InterestRate()* on *AutoLoanAccount* I have to change the sign because it
779 returns the interest the consumer pays to the bank.” According to Liskov, the
780 *InterestRate()* routine should not be inherited because its semantics aren’t the
781 same for all derived classes.

782 *Be sure to inherit only what you want to inherit*

783 A derived class can inherit member routine interfaces, implementations, or both.
784 Table 6-1 shows the variations of how routines can be implemented and
785 overridden.

786

Table 6-1. Variations on inherited routines

	Overridable	Not Overridable
Implementation: Default Provided	Overridable Routine	Non-Overridable Routine
Implementation: No default provided	Abstract Overridable Routine	Not used (doesn't make sense to leave a routine undefined and not allow it to be overridden)

787

As the table suggests, inherited routines come in three basic flavors:

788

789

790

791

792

793

794

795

796

797

798

799

- An *abstract overridable routine* means that the derived class inherits the routine's interface but not its implementation.
- An *overridable routine* means that the derived class inherits the routine's interface and a default implementation, and it is allowed to override the default implementation.
- A *non-overridable routine* means that the derived class inherits the routine's interface and its default implementation, and it is not allowed to override the routine's implementation.

When you choose to implement a new class through inheritance, think through the kind of inheritance you want for each member routine. Beware of inheriting implementation just because you're inheriting an interface, and beware of inheriting an interface just because you want to inherit an implementation.

800

801

802

803

804

805

806

807

Don't "override" a non-overridable member function

Both C++ and Java allow a programmer to override a non-overridable member routine—kind of. If a function is *private* in the base class, a derived class can create a function with the same name. To the programmer reading the code in the derived class, such a function can create confusion because it looks like it should be polymorphic, but it isn't; it just has the same name. Another way to state this guideline is, Don't reuse names of non-overridable base-class routines in derived classes.

808

809

810

811

812

813

Move common interfaces, data, and behavior as high as possible in the inheritance tree

The higher you move interfaces, data, and behavior, the more easily derived classes can use them. How high is too high? Let *abstraction* be your guide. If you find that moving a routine higher would break the higher object's abstraction, don't do it.

814

815

816

Be suspicious of classes of which there is only one instance

A single instance might indicate that the design confuses objects with classes.

Consider whether you could just create an object instead of a new class. Can the

817 variation of the derived class be represented in data rather than as a distinct
818 class?

819 ***Be suspicious of base classes of which there is only one derived class***
820 When I see a base class that has only one derived class, I suspect that some
821 programmer has been “designing ahead”—trying to anticipate future needs,
822 usually without fully understanding what those future needs are. The best way to
823 prepare for future work is not to design extra layers of base classes that “might
824 be needed someday,” it’s to make current work as clear, straightforward, and
825 simple as possible. That means not creating any more inheritance structure than
826 is absolutely necessary.

827 ***Be suspicious of classes that override a routine and do nothing inside the***
828 ***derived routine***

829 This typically indicates an error in the design of the base class. For instance,
830 suppose you have a class *Cat* and a routine *Scratch()* and suppose that you
831 eventually find out that some cats are declawed and can’t scratch. You might be
832 tempted to create a class derived from *Cat* named *ScratchlessCat* and override
833 the *Scratch()* routine to do nothing. There are several problems with this
834 approach:

- 835 • It violates the abstraction (interface contract) presented in the *Cat* class by
836 changing the semantics of its interface.
- 837 • This approach quickly gets out of control when you extend it to other
838 derived classes. What happens when you find a cat without a tail? Or a cat
839 that doesn’t catch mice? Or a cat that doesn’t drink milk? Eventually you’ll
840 end up with derived classes like *ScratchlessTaillessMicelessMilklessCat*.
- 841 • Over time, this approach gives rise to code that’s confusing to maintain
842 because the interfaces and behavior of the ancestor classes imply little or
843 nothing about the behavior of their descendants.

844 The place to fix this problem is not in the base class, but in the original *Cat* class.
845 Create a *Claws* class and contain that within the *Cats* class, or build a constructor
846 for the class that includes whether the cat scratches. The root problem was the
847 assumption that all cats scratch, so fix that problem at the source, rather than just
848 bandaging it at the destination.

849 ***Avoid deep inheritance trees***

850 Object oriented programming provides a large number of techniques for
851 managing complexity. But every powerful tool has its hazards, and some object-
852 oriented techniques have a tendency to increase complexity rather than reduce it.

853 In his excellent book *Object-Oriented Design Heuristics*, Arthur Riel suggests
854 limiting inheritance hierarchies to a maximum of six levels (1996). Riel bases his

855 recommendation on the “magic number 7 ± 2 ,” but I think that’s grossly
856 optimistic. In my experience most people have trouble juggling more than two or
857 three levels of inheritance in their brains at once. The “magic number 7 ± 2 ” is
858 probably better applied as a limit to the *total number of subclasses* of a base
859 class rather than the number of levels in an inheritance tree.

860 Deep inheritance trees have been found to be significantly associated with
861 increased fault rates (Basili, Briand, and Melo 1996). Anyone who has ever tried
862 to debug a complex inheritance hierarchy knows why.

863 Deep inheritance trees increase complexity, which is exactly the opposite of
864 what inheritance should be used to accomplish. Keep the primary technical
865 mission in mind. Make sure you’re using inheritance to *minimize complexity*.

866 ***Prefer inheritance to extensive type checking***

867 Frequently repeated *case* statements sometimes suggest that inheritance might be
868 a better design choice, although this is not always true. Here is a classic example
869 of code that cries out for a more object-oriented approach:

870 **C++ Example of a Case Statement That Probably Should be Replaced
871 by Inheritance**

```
872 switch ( shape.type ) {  
873     case Shape_Circle:  
874         shape.DrawCircle();  
875         break;  
876     case Shape_Square:  
877         shape.DrawSquare();  
878         break;  
879     ...  
880 }
```

881 In this example, the calls to *shape.DrawCircle()* and *shape.DrawSquare()* should
882 be replaced by a single routine named *shape.Draw()*, which can be called
883 regardless of whether the shape is a circle or a square.

884 On the other hand, sometimes *case* statements are used to separate truly different
885 kinds of objects or behavior. Here is an example of a *case* statement that is
886 appropriate in an object-oriented program:

887 **C++ Example of a Case Statement That Probably Should not be
888 Replaced by Inheritance**

```
889 switch ( ui.Command() ) {  
890     case Command_OpenFile:  
891         OpenFile();  
892         break;
```

```
893     case Command_Print:  
894         Print();  
895         break;  
896     case Command_Save:  
897         Save();  
898         break;  
899     case Command_Exit:  
900         ShutDown();  
901         break;  
902         ...  
903     }
```

In this case, it would be possible to create a base class with derived classes and a polymorphic *DoCommand()* routine for each command. But the meaning of *DoCommand()* would be so diluted as to be meaningless, and the *case* statement is the more understandable solution.

Avoid using a base class's protected data in a derived class (or make that data private instead of protected in the first place)

As Joshua Bloch says, “Inheritance breaks encapsulation” (2001). When you inherit from an object, you obtain privileged access to that object’s protected routines and data. If the derived class really needs access to the base class’s attributes, provide protected accessor functions instead.

Multiple Inheritance

Inheritance is a power tool. It’s like using a chainsaw to cut down a tree instead of a manual cross-cut saw. It can be incredibly useful when used with care, but it’s dangerous in the hands of someone who doesn’t observe proper precautions.

If inheritance is a chain saw, multiple inheritance is a 1950s-era chain saw with no blade guard, not automatic shut off, and a finicky engine. There are times when such a tool is indispensable, mostly, you’re better off leaving the tool in the garage where it can’t do any damage.

Although some experts recommend broad use of multiple inheritance (Meyer 1997), in my experience multiple inheritance is useful primarily for defining “mixins,” simple classes that are used to add a set of properties to an object. Mixins are called mixins because they allow properties to be “mixed in” to derived classes. Mixins might be classes like *Displayable*, *Persistent*, *Serializable*, or *Sortable*. Mixins are nearly always abstract and aren’t meant to be instantiated independently of other objects.

Mixins require the use of multiple inheritance, but they aren’t subject to the classic diamond-inheritance problem associated with multiple inheritance as long as all mixins are truly independent of each other. They also make the design more comprehensible by “chunking” attributes together. A programmer will

933
934
935

936
937
938
939
940

941

942 | **CROSS-REFERENCE** For
943 more on complexity, see
944 "Software's Primary
945 Technical Imperative:
946 Managing Complexity" in
947 Section 5.2

948
949

950
951

952
953

954
955

956

957 | **CROSS-REFERENCE** For
958 more discussion of routines
959 in general, see Chapter 7,
960 "High-Quality Routines."

961
962
963
964
965

have an easier time understanding that an object uses the mixins *Displayable* and *Persistant* than understanding that an object uses the 11 more specific routines that would otherwise be needed to implement those two properties.

Java and Visual Basic recognize the value of mixins by allowing multiple inheritance of interfaces but only single class inheritance. C++ supports multiple inheritance of both interface and implementation. Programmers should use multiple inheritance only after carefully considering the alternatives and weighing the impact on system complexity and comprehensibility.

Why Are There So Many Rules for Inheritance?

This section has presented numerous rules for staying out of trouble with inheritance. The underlying message of all these rules is that, *inheritance tends to work against the primary technical imperative you have as a programmer, which is to manage complexity*. For the sake of controlling complexity you should maintain a heavy bias against inheritance. Here's a summary of when to use inheritance and when to use containment:

- If multiple classes share common data but not behavior, then create a common object that those classes can contain.
- If multiple classes share common behavior but not data, then derive them from a common base class that defines the common routines.
- If multiple classes share common data and behavior, then inherit from a common base class that defines the common data and routines.
- Inherit when you want the base class to control your interface; contain when you want to control your interface.

Member Functions and Data

Here are a few guidelines for implementing member functions and member data effectively.

Keep the number of routines in a class as small as possible

A study of C++ programs found that higher numbers of routines per class were associated with higher fault rates (Basili, Briand, and Melo 1996). However, other competing factors were found to be more significant, including deep inheritance trees, large number of routines called by a routine, and strong coupling between classes. Evaluate the tradeoff between minimizing the number of routines and these other factors.

966
967
968
969
970
971
972
973
974
975

Disallow implicitly generated member functions and operators you don't want

Sometimes you'll find that you want to disallow certain functions—perhaps you want to disallow assignment, or you don't want to allow an object to be constructed. You might think that, since the compiler generates operators automatically, you're stuck allowing access. But in such cases you can disallow those uses by declaring the constructor, assignment operator, or other function or operator *private*, which will prevent clients from accessing it. (Making the constructor private is a standard technique for defining a singleton class, which is discussed later in this chapter.)

976
977
978
979
980

Minimize direct routine calls to other classes

One study found that the number of faults in a class was statistically correlated with the total number of routines that were called from within a class (Basili, Briand, and Melo 1996). The same study found that the more classes a class used, the higher its fault rate tended to be.

981
982 **FURTHER READING** Good
983 accounts of the Law of
984 Demeter can be found in
985 *Pragmatic Programmer*
986 (Hunt and Thomas 2000),
987 *Applying UML and Patterns*
988 (Larman 2001), and
989 *Fundamentals of Object-
990 Oriented Design in UML*
(Page-Jones 2000).

Minimize indirect routine calls to other classes

Direct connections are hazardous enough. Indirect connections—such as *account.ContactPerson().DaytimeContactInfo().PhoneNumber()*—tend to be even more hazardous. Researchers have formulated a rule called the “Law of Demeter” (Lieberherr and Holland 1989) which essentially states that Object A can call any of its own routines. If Object A instantiates an Object B, it can call any of Object B’s routines. But it should avoid calling routines on objects provided by Object B. In the *account* example above, that means *account.ContactPerson()* is OK, but *account.ContactPerson().DaytimeContactInfo()* is not.

991
992
993
994

This is a simplified explanation, and, depending on how classes are arranged, it might be acceptable to see an expression like *account.ContactPerson().DaytimeContactInfo()*. See the additional resources at the end of this chapter for more details.

995
996
997

In general, minimize the extent to which a class collaborates with other classes

Try to minimize all of the following:

998
999
1000

- Number of kinds of objects instantiated
- Number of different direct routine calls on instantiated objects
- Number of routine calls on objects returned by other instantiated objects

Constructors

1001
1002 Here are some guidelines that apply specifically to constructors. Guidelines for
1003 constructors are pretty similar across languages (C++, Java, and Visual Basic,
1004 anyway). Destructors vary more, and so you should check out the materials listed
1005 in the “Additional Resources” section at the end of the chapter for more
1006 information on destructors.

Initialize all member data in all constructors, if possible

1007 Initializing all data members in all constructors is an inexpensive defensive
1008 programming practice.
1009

Initialize data members in the order in which they’re declared

1010 Depending on your compiler, you can experience some squirrely errors by
1011 trying to initialize data members in a different order than the order in which
1012 they’re declared. Using the same order in both places also provides consistency
1013 that makes the code easier to read.
1014

Enforce the singleton property by using a private constructor

1015 If you want to define a class that allows only one object to be instantiated, you
1016 **FURTHER READING** The
1017 code to do this in C++ would
1018 be similar. For details, see
1019 *More Effective C++*, Item 26
(Meyers 1998).

Java Example of Enforcing a Singleton With a Private Constructor

1020
1021
1022 *Here is the private*
1023 *constructor.*
1024
1025
1026
1027
1028
1029 *Here is the public routine that*
1030 *provides access to the single*
1031 *instance.*
1032
1033
1034
1035 *Here is the single instance.*
1036
1037
1038
1039
1040

```
public class MaxId {  
    // constructors and destructors  
    private MaxId() {  
        ...  
    }  
    ...  
  
    // public routines  
    public static MaxId GetInstance() {  
        return m_instance;  
    }  
    ...  
  
    // private members  
    private static final MaxId m_instance = new MaxId();  
    ...  
}
```

The private constructor is called only when the static object *m_instance* is initialized. In this approach, if you want to reference the *MaxId* singleton, you would simply refer to *MaxId.GetInstance()*.

1041 ***Enforce the singleton property by using all static member data and***
1042 ***reference counting***
1043 An alternative means of enforcing the singleton property is to declare all the
1044 class's data static. You can determine whether the class is being used by
1045 incrementing a reference counter in the object's constructor and decrementing it
1046 in the destructor (C++) or *Terminate* routine (Java and Visual Basic).

1047 The reference-counting approach comes with some systemic pitfalls. If the
1048 reference is copied, then the class data member won't necessarily be
1049 incremented, which can lead to an error in the reference count. If this approach is
1050 used, the project team should standardize on conventions to use reference-
1051 counted objects consistently.

1052 ***Prefer deep copies to shallow copies until proven otherwise***
1053 One of the major decisions you'll make about complex objects is whether to
1054 implement deep copies or shallow copies of the object. A deep copy of an object
1055 is a member-wise copy of the object's member data. A shallow copy typically
1056 just points to or refers to a single reference copy.

1057 Deep copies are simpler to code and maintain than shallow copies. In addition to
1058 the code either kind of object would contain, shallow copies add code to count
1059 references, ensure safe object copies, safe comparisons, safe deletes, and so on.
1060 This code tends to be error prone, and it should be avoided unless there's a
1061 compelling reason to create it.

1062 The motivation for creating shallow copies is typically to improve performance.
1063 Although creating multiple copies of large objects might be aesthetically
1064 offensive, it rarely causes any measurable performance impact. A small number
1065 of objects might cause performance issues, but programmers are notoriously
1066 poor at guessing which code really causes problems. (For details, see Chapter
1067 25.) Because it's a poor tradeoff to add complexity for dubious performance
1068 gains, a good approach to deep vs. shallow copies is to prefer deep copies until
1069 proven otherwise.

1070 If you find that you do need to use a shallow-copy approach, Scott Meyers'
1071 *More Effective C++*, Item 29 (1996) contains an excellent discussion of the
1072 issues in C++. Martin Fowler's *Refactoring* (1999) describes the specific steps
1073 needed to convert from shallow copies to deep copies and from deep copies to
1074 shallow copies. (Fowler calls them reference objects and value objects.)

1075 **CROSS-REFERENCE** The
1076 reasons to create a class
1077 overlap with the reasons to
create routines. For details,
1078 see Section 7.1, “Valid
1079 Reasons to Create a
Routine.”

1080 **CROSS-REFERENCE** For
1081 more on identifying real-
1082 world objects, see “Find
1083 Real-World Objects” in
1084 Section 5.3.

1085
1086
1087
1088
1089

1090 On programming projects, the abstractions are not ready made the way *Shape* is,
1091 so we have to work harder to come up with clean abstractions. The process of
1092 distilling abstract concepts from real-world entities is non-deterministic, and
1093 different designers will abstract out different generalities. If we didn’t know
1094 about geometric shapes like circles, squares and triangles, for example, we might
1095 come up with more unusual shapes like squash shape, rutabaga shape, and
1096 Pontiac Aztek shape. Coming up with appropriate abstract objects is one of the
1097 major challenges in object-oriented design.

1098 **KEY POINT**

1099
1100
1101
1102
1103
1104
1105
1106

Reduce complexity

The single most important reason to create a class is to reduce a program’s complexity. Create a class to hide information so that you won’t need to think about it. Sure, you’ll need to think about it when you write the class. But after it’s written, you should be able to forget the details and use the class without any knowledge of its internal workings. Other reasons to create classes—minimizing code size, improving maintainability, and improving correctness—are also good reasons, but without the abstractive power of classes, complex programs would be impossible to manage intellectually.

1107
1108
1109
1110
1111
1112

Isolate complexity

Complexity in all forms—complicated algorithms, large data sets, intricate communications protocols, and so on—is prone to errors. If an error does occur, it will be easier to find if it isn’t spread through the code but is localized within a class. Changes arising from fixing the error won’t affect other code because only one class will have to be fixed—other code won’t be touched. If you find a

1113 better, simpler, or more reliable algorithm, it will be easier to replace the old
1114 algorithm if it has been isolated into a class. During development, it will be
1115 easier to try several designs and keep the one that works best.

1116 ***Hide implementation details***

1117 The desire to hide implementation details is a wonderful reason to create a class
1118 whether the details are as complicated as a convoluted database access or as
1119 mundane as whether a specific data member is stored as a number or a string.

1120 ***Limit effects of changes***

1121 Isolate areas that are likely to change so that the effects of changes are limited to
1122 the scope of a single class or, at most, a few classes. Design so that areas that are
1123 most likely to change are the easiest to change. Areas likely to change include
1124 hardware dependencies, input/output, complex data types, and business rules.
1125 The subsection titled “*Hide Secrets (Information Hiding)*” in Section 5.3
1126 described several common sources of change. Several of the most common are
1127 summarized in this section.

1128 ***Hide global data***

1129 **CROSS-REFERENCE** For
1130 a discussion of problems
1131 associated with using global
1132 data, see Section 13.3,
1133 “*Global Data*.”
1134
1135

If you need to use global data, you can hide its implementation details behind a class interface. Working with global data through access routines provides several benefits compared to working with global data directly. You can change the structure of the data without changing your program. You can monitor accesses to the data. The discipline of using access routines also encourages you to think about whether the data is really global; it often becomes apparent that the “global data” is really just class data.

1136 ***Streamline parameter passing***

1137 If you’re passing a parameter among several routines, that might indicate a need
1138 to factor those routines into a class that share the parameter as class data.
1139 Streamlining parameter passing isn’t a goal, per se, but passing lots of data
1140 around suggests that a different class organization might work better.

1141 ***Make central points of control***

1142 **CROSS-REFERENCE** For
1143 details on information hiding,
1144 see “*Hide Secrets*
1145 (*Information Hiding*)” in
1146 Section 5.3.
1147

It’s a good idea to control each task in one place. Control assumes many forms. Knowledge of the number of entries in a table is one form. Control of devices—files, database connections, printers, and so on—is another. Using one class to read from and write to a database is a form of centralized control. If the database needs to be converted to a flat file or to in-memory data, the changes will affect only the one class.

1148 The idea of centralized control is similar to information hiding, but it has unique
1149 heuristic power that makes it worth adding to your programming toolbox.

1150
1151
1152
1153
1154
1155

Facilitate reusable code

Code put into well-factored classes can be reused in other programs more easily than the same code embedded in one larger class. Even if a section of code is called from only one place in the program and is understandable as part of a larger class, it makes sense to put it into its own class if that piece of code might be used in another program.

1156 **HARD DATA**

1157
1158
1159
1160
1161
1162
1163
1164

NASA's Software Engineering Laboratory studied ten projects that pursued reuse aggressively (McGarry, Waligora, and McDermott 1989). In both the object-oriented and the functionally oriented approaches, the initial projects weren't able to take much of their code from previous projects because previous projects hadn't established a sufficient code base. Subsequently, the projects that used functional design were able to take about 35 percent of their code from previous projects. Projects that used an object-oriented approach were able to take more than 70 percent of their code from previous projects. If you can avoid writing 70 percent of your code by planning ahead, do it!

1165 **CROSS-REFERENCE** For
1166 more on implementing the
minimum amount of
1167 functionality required, see "A
1168 program contains code that
1169 seems like it might be needed
1170 someday" in Section 24.3.

Notably, the core of NASA's approach to creating reusable classes does not involve "designing for reuse." NASA identifies reuse candidates at the ends of their projects. They then perform the work needed to make the classes reusable as a special project at the end of the main project or as the first step in a new project. This approach helps prevent "gold-plating"—creation of functionality that isn't required and that adds complexity unnecessarily.

Plan for a family of programs

If you expect a program to be modified, it's a good idea to isolate the parts that you expect to change by putting them into their own classes. You can then modify the classes without affecting the rest of the program, or you can put in completely new classes instead. Thinking through not just what one program will look like, but what the whole family of programs might look like is a powerful heuristic for anticipating entire categories of changes (Parnas 1976).

Several years ago I managed a team that wrote a series of programs used by our clients to sell insurance. We had to tailor each program to the specific client's insurance rates, quote-report format, and so on. But many parts of the programs were similar: the classes that input information about potential customers, that stored information in a customer database, that looked up rates, that computed total rates for a group, and so on. The team factored the program so that each part that varied from client to client was in its own class. The initial programming might have taken three months or so, but when we got a new client, we merely wrote a handful of new classes for the new client and dropped them into the rest of the code. A few days' work, and voila! Custom software!

1188
1189
1190
1191
1192

Package related operations

In cases in which you can't hide information, share data, or plan for flexibility, you can still package sets of operations into sensible groups such as trig functions, statistical functions, string-manipulation routines, bit-manipulation routines, graphics routines, and so on.

1193
1194
1195
1196
1197
1198

To accomplish a specific refactoring

Many of the specific refactorings described in Chapter 24 result in new classes—including converting one class to two, hiding a delegate, removing a middle man, and introducing an extension class. These new classes could be motivated by a desire to better accomplish any of the objectives described throughout this section.

1199

1200
1201

Classes to Avoid

While classes in general are good, you can run into a few gotchas. Here are some classes to avoid.

1202
1203
1204
1205
1206
1207

Avoid creating god classes

Avoid creating omniscient classes that are all-knowing and all-powerful. If a class spends its time retrieving data from other classes using *Get()* and *Set()* routines (that is, digging into their business and telling them what to do), ask whether that functionality might better be organized into those other classes rather than into the god class (Riel 1996).

1208
1209 **CROSS-REFERENCE** This
1210 kind of class is usually called
a structure. For more on
1211 structures, see Section 13.1,
1212 “Structures.”
1213
1214

Eliminate irrelevant classes

If a class consists only of data but no behavior, ask yourself whether it's really a class and consider demoting it to become an attribute of another class.

Avoid classes named after verbs

A class that has only behavior but no data is generally not really a class. Consider turning a class like *DatabaseInitialization()* or *StringBuilder()* into a routine on some other class.

1215

1216

Summary of Reasons to Create a Class

Here's a summary list of the valid reasons to create a class:

1217
1218
1219
1220
1221

- Model real-world objects
- Model abstract objects
- Reduce complexity
- Isolate complexity
- Hide implementation details

- 1222 • Limit effects of changes
1223 • Hide global data
1224 • Streamline parameter passing
1225 • Make central points of control
1226 • Facilitate reusable code
1227 • Plan for a family of programs
1228 • Package related operations
1229 • To accomplish a specific refactoring

1230 6.5 Language-Specific Issues

1231 Approaches to classes in different programming languages vary in interesting
1232 ways. Consider how you override a member routine to achieve polymorphism in
1233 a derived class. In Java, all routines are overridable by default, and a routine
1234 must be declared *final* to prevent a derived class from overriding it. In C++,
1235 routines are not overridable by default. A routine must be declared *virtual* in the
1236 base class to be overridable. In Visual Basic, a routine must be declared
1237 *overridable* in the base class, and the derived class should use the *overrides*
1238 keyword.

1239 Here are some of the class-related areas that vary significantly depending on the
1240 language:

- 1241 • Behavior of overridden constructors and destructors in an inheritance tree
1242 • Behavior of constructors and destructors under exception-handling
1243 conditions
1244 • Importance of default constructors (constructors with no arguments)
1245 • Time at which a destructor or finalizer is called
1246 • Wisdom of overriding the language's built-in operators, including
1247 assignment and equality
1248 • How memory is handled as objects are created and destroyed, or as they are
1249 declared and go out of scope

1250 Detailed discussions of these issues are beyond the scope of this book, but the
1251 “Additional Resources” section at the end of this chapter points to good
1252 language-specific resources.

1253

6.6 Beyond Classes: Packages

1254 **CROSS-REFERENCE** For
1255 more on the distinction
1256 between classes and
1257 packages, see “Levels of
Design” in Section 5.2.
1258
1259

Classes are currently the best way for programmers to achieve modularity. But modularity is a big topic, and it extends beyond classes. Over the past several decades, software development has advanced in large part by increasing the granularity of the aggregations that we have to work with. The first aggregation we had was the statement, which at the time seemed like a big step up from machine instructions. Then came subroutines, and later came classes.

1260
1261
1262
1263

It’s evident that we could better support the goals of abstraction and encapsulation if we had good tools for aggregating groups of objects. Ada supported the notion of packages more than a decade ago, and Java supports packages today.

1264
1265

C++’s and C#’s namespaces are a good step in the right direction, though creating packages with them is a little bit like writing web pages directly in html.

1266
1267
1268

If you’re programming in a language that doesn’t support packages directly, you can create your own poor-programmer’s version of a package and enforce it through programming standards that include

1269
1270
1271
1272
1273
1274

- naming conventions that differentiate which classes are public and which are for the package’s private use
- naming conventions, code-organization conventions (project structure), or both that identify which package each class belongs to
- Rules that define which packages are allowed to use which other packages, including whether the usage can be inheritance, containment, or both

1275
1276
1277

These workaround are good examples of the distinction between programming *in* a language vs. programming *into* a language. For more on this distinction, see Section 34.4, “Program Into Your Language, Not In It.”

1278 CC2E.COM/0672
1279
1280

CROSS-REFERENCE This is a checklist of considerations about the quality of the class. For a list of the steps used to build a class, see the checklist “The Pseudocode Programming Process” in Chapter 9, page 000.

1281

CHECKLIST: Class Quality

1282
1283
1284

Abstract Data Types

- Have you thought of the classes in your program as Abstract Data Types and evaluated their interfaces from that point of view?

1285

Abstraction

- Does the class have a central purpose?
- Is the class well named, and does its name describe its central purpose?
- Does the class's interface present a consistent abstraction?
- Does the class's interface make obvious how you should use the class?
- Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?
- Are the class's services complete enough that other classes don't have to meddle with its internal data?
- Has unrelated information been moved out of the class?
- Have you thought about subdividing the class into component classes, and have you subdivided it as much as you can?
- Are you preserving the integrity of the class's interface as you modify the class?

1299

Encapsulation

- Does the class minimize accessibility to its members?
- Does the class avoid exposing member data?
- Does the class hide its implementation details from other classes as much as the programming language permits?
- Does the class avoid making assumptions about its users, including its derived classes?
- Is the class independent of other classes? Is it loosely coupled?

1307

Inheritance

- Is inheritance used only to model "is a" relationships?
- Does the class documentation describe the inheritance strategy?
- Do derived classes adhere to the Liskov Substitution Principle?
- Do derived classes avoid "overriding" non overridable routines?
- Are common interfaces, data, and behavior as high as possible in the inheritance tree?
- Are inheritance trees fairly shallow?
- Are all data members in the base class private rather than protected?

1316

Other Implementation Issues

- Does the class contain about seven data members or fewer?
- Does the class minimize direct and indirect routine calls to other classes?

- 1319 Does the class collaborate with other classes only to the extent absolutely
1320 necessary?
1321 Is all member data initialized in the constructor?
1322 Is the class designed to be used as deep copies rather than shallow copies
1323 unless there's a measured reason to create shallow copies?

1324 **Language-Specific Issues**

- 1325 Have you investigated the language-specific issues for classes in your
1326 specific programming language?

1327 CC2E.COM/0679

1328 **Additional Resources**

1329 **Classes in General**

1330 Meyer, Bertrand. *Object-Oriented Software Construction*, 2d Ed. New York:
1331 Prentice Hall PTR, 1997. This book contains an in-depth discussion of Abstract
1332 Data Types and explains how they form the basis for classes. Chapters 14-16
1333 discuss inheritance in depth. Meyer provides a strong argument in favor of
1334 multiple inheritance in Chapter 15.

1335 Riel, Arthur J. *Object-Oriented Design Heuristics*, Reading, Mass.: Addison
1336 Wesley, 1996. This book contains numerous suggestions for improving program
1337 design, mostly at the class level. I avoided the book for several years because it
1338 appeared to be too big (talk about people in glass houses!). However, the body of
1339 the book is only about 200 pages long. Riel's writing is accessible and enjoyable.
1340 The content is focused and practical.

1341 **C++**

1342 CC2E.COM/0686
1343 Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and
Designs*, 2d Ed, Reading, Mass.: Addison Wesley, 1998.

1344 Meyers, Scott, 1996, *More Effective C++: 35 New Ways to Improve Your
Programs and Designs*, Reading, Mass.: Addison Wesley, 1996. Both of
1345 Meyers' books are canonical references for C++ programmers. The books are
1346 entertaining and help to instill a language-lawyer's appreciation for the nuances
1347 of C++.

1349 **Java**

1350 CC2E.COM/0693
1351 Bloch, Joshua. *Effective Java Programming Language Guide*, Boston, Mass.:
1352 Addison Wesley, 2001. Bloch's book provides much good Java-specific advice
as well as introducing more general, good object-oriented practices.

1353

Visual Basic

1354 CC2E.COM/0600

The following books are good references on classes in Visual Basic:

1355

Foxall, James. *Practical Standards for Microsoft Visual Basic .NET*, Redmond, WA: Microsoft Press, 2003.

1356

Cornell, Gary and Jonathan Morrison. *Programming VB .NET: A Guide for Experienced Programmers*, Berkeley, Calif.: Apress, 2002.

1357

Barwell, Fred, et al. *Professional VB.NET, 2d Ed.*, Wrox, 2002.

1358

1359

Key Points

1360

- Class interfaces should provide a consistent abstraction. Many problems arise from violating this single principle.
- A class interface should hide something—a system interface, a design decision, or an implementation detail.
- Containment is usually preferable to inheritance unless you’re modeling an “is a” relationship.
- Inheritance is a useful tool, but it adds complexity, which is counter to the Primary Technical Imperative of minimizing complexity.
- Classes are your primary tool for managing complexity. Give their design as much attention as needed to accomplish that objective.

7

High-Quality Routines

Contents

- 7.1 Valid Reasons to Create a Routine
- 7.2 Design at the Routine Level
- 7.3 Good Routine Names
- 7.4 How Long Can a Routine Be?
- 7.5 How to Use Routine Parameters
- 7.6 Special Considerations in the Use of Functions
- 7.7 Macro Routines and Inline Routines

Related Topics

Steps in routine construction: Section 9.3

Characteristics of high-quality classes: Chapter 6

General design techniques: Chapter 5

Software architecture: Section 3.5

CHAPTER 6 DESCRIBED DETAILS of creating classes. This chapter zooms in on routines, on the characteristics that make the difference between a good routine and a bad one. If you'd rather read about high-level design issues before wading into the nitty-gritty details of individual routines, be sure to read Chapter 5, "High-Level Design in Construction" first and come back to this chapter later. If you're more interested in reading about steps to create routines (and classes), Chapter 9, "The Pseudocode Programming Process" might be a better place to start.

Before jumping into the details of high-quality routines, it will be useful to nail down two basic terms. What is a "routine?" A routine is an individual method or procedure invocable for a single purpose. Examples include a function in C++, a method in Java, a function or sub procedure in Visual Basic. For some uses, macros in C and C++ can also be thought of as routines. You can apply many of the techniques for creating a high-quality routine to these variants.

30
31
32

What is a *high-quality* routine? That's a harder question. Perhaps the easiest answer is to show what a high-quality routine is not. Here's an example of a low-quality routine:

CODING HORROR33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec, double
    & estimRevenue, double ytdRevenue, int screenX, int screenY, COLOR_TYPE &
    newColor, COLOR_TYPE & prevColor, StatusType & status, int expenseType )
{
    int i;
    for ( i = 0; i < 100; i++ ) {
        inputRec.revenue[i] = 0;
        inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
    }
    UpdateCorpDatabase( empRec );
    estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
    newColor = prevColor;
    status = SUCCESS;
    if ( expenseType == 1 ) {
        for ( i = 0; i < 12; i++ )
            profit[i] = revenue[i] - expense.type1[i];
    }
    else if ( expenseType == 2 ) {
        profit[i] = revenue[i] - expense.type2[i];
    }
    else if ( expenseType == 3 )
        profit[i] = revenue[i] - expense.type3[i];
}
```

57
58
59

What's wrong with this routine? Here's a hint: You should be able to find at least 10 different problems with it. Once you've come up with your own list, look at the list below:

60
61
62
63
64
65
66
67
68
69

- The routine has a bad name. *HandleStuff()* tells you nothing about what the routine does.
- The routine isn't documented. (The subject of documentation extends beyond the boundaries of individual routines and is discussed in Chapter 19, "Self-Documenting Code.")
- The routine has a bad layout. The physical organization of the code on the page gives few hints about its logical organization. Layout strategies are used haphazardly, with different styles in different parts of the routine. Compare the styles where *expenseType == 2* and *expenseType == 3*. (Layout is discussed in Chapter 18, "Layout and Style.")

- 70 • The routine's input variable, *inputRec*, is changed. If it's an input variable,
71 its value should not be modified. If the value of the variable is supposed to
72 be modified, the variable should not be called *inputRec*.
- 73 • The routine reads and writes global variables. It reads from *corpExpense* and
74 writes to *profit*. It should communicate with other routines more directly
75 than by reading and writing global variables.
- 76 • The routine doesn't have a single purpose. It initializes some variables,
77 writes to a database, does some calculations—none of which seem to be
78 related to each other in any way. A routine should have a single, clearly
79 defined purpose.
- 80 • The routine doesn't defend itself against bad data. If *crntQtr* equals 0, then
81 the expression *ytdRevenue * 4.0 / (double) crntQtr* causes a divide-by-zero
82 error.
- 83 • The routine uses several magic numbers: 100, 4.0, 12, 2, and 3. Magic
84 numbers are discussed in Section 11.1, "Numbers in General."
- 85 • The routine uses only two fields of the *CORP_DATA* type of parameter. If
86 only two fields are used, the specific fields rather than the whole structured
87 variable should probably be passed in.
- 88 • Some of the routine's parameters are unused. *screenX* and *screenY* are not
89 referenced within the routine.
- 90 • One of the routine's parameters is mislabeled. *prevColor* is labeled as a
91 reference parameter (&) even though it isn't assigned a value within the
92 routine.
- 93 • The routine has too many parameters. The upper limit for an understandable
94 number of parameters is about 7. This routine has 11. The parameters are
95 laid out in such an unreadable way that most people wouldn't try to examine
96 them closely or even count them.
- 97 • The routine's parameters are poorly ordered and are not documented.
98 (Parameter ordering is discussed in this chapter. Documentation is discussed
99 in Chapter 20.)

100 **CROSS-REFERENCE** The
101 class is also a good contender
102 for the single greatest
103 invention in computer
104 science. For details on how to
use classes effectively, See
Chapter 6, "Working
105 Classes."

106

107

Aside from the computer itself, the routine is the single greatest invention in computer science. The routine makes programs easier to read and easier to understand than any other feature of any programming language. It's a crime to abuse this senior statesman of computer science with code like that shown in the example above.

The routine is also the greatest technique ever invented for saving space and improving performance. Imagine how much larger your code would be if you had to repeat the code for every call to a routine instead of branching to the

108 routine. Imagine how hard it would be to make performance improvements in
109 the same code used in a dozen places instead of making them all in one routine.
110 The routine makes modern programming possible.

111 “OK,” you say, “I already know that routines are great, and I program with them
112 all the time. This discussion seems kind of remedial, so what do you want me to
113 do about it?”

114 I want you to understand that there are many valid reasons to create a routine and
115 that there are right ways and wrong ways to go about it. As an undergraduate
116 computer-science student, I thought that the main reason to create a routine was
117 to avoid duplicate code. The introductory textbook I used said that routines were
118 good because the avoidance of duplication made a program easier to develop,
119 debug, document, and maintain. Period. Aside from syntactic details about how
120 to use parameters and local variables, that was the total extent of the textbook’s
121 description of the theory and practice of routines. It was not a good or complete
122 explanation. The following sections contain a much better explanation.

123 7.1 Valid Reasons to Create a Routine

124 Here’s a list of valid reasons to create a routine. The reasons overlap somewhat,
125 and they’re not intended to make an orthogonal set.

126 KEY POINT

127 The single most important reason to create a routine is to reduce a program’s
128 complexity. Create a routine to hide information so that you won’t need to think
129 about it. Sure, you’ll need to think about it when you write the routine. But after
130 it’s written, you should be able to forget the details and use the routine without
131 any knowledge of its internal workings. Other reasons to create routines—
132 minimizing code size, improving maintainability, and improving correctness—
133 are also good reasons, but without the abstractive power of routines, complex
134 programs would be impossible to manage intellectually.

135 One indication that a routine needs to be broken out of another routine is deep
136 nesting of an inner loop or a conditional. Reduce the containing routine’s
137 complexity by pulling the nested part out and putting it into its own routine.

138 Make a section of code readable

139 Putting a section of code into a well-named routine is one of the best ways to
140 document its purpose. Instead of reading a series of statements like

```
141     if ( node <> NULL ) then
142         while ( node.next <> NULL ) do
143             node = node.next
```

```
144           leafName = node.name
145       end while
146   else
147       leafName = ""
148   end if
149 you can read a statement like
```

```
150     leafName = GetLeafName( node )
151
152 The new routine is so short that nearly all it needs for documentation is a good
153 name. Using a routine call instead of six lines of code makes the routine that
originally contained the code less complex and documents it automatically.
```

Avoid duplicate code

Undoubtedly the most popular reason for creating a routine is to avoid duplicate code. Indeed, creation of similar code in two routines implies an error in decomposition. Pull the duplicate code from both routines, put a generic version of the common code into its own routine, and then let both call the part that was put into the new routine. With code in one place, you save the space that would have been used by duplicated code. Modifications will be easier because you'll need to modify the code in only one location. The code will be more reliable because you'll have to check only one place to ensure that the code is right. Modifications will be more reliable because you'll avoid making successive and slightly different modifications under the mistaken assumption that you've made identical ones.

Hide sequences

It's a good idea to hide the order in which events happen to be processed. For example, if the program typically gets data from the user and then gets auxiliary data from a file, neither the routine that gets the user data nor the routine that gets the file data should depend on the other routine's being performed first. If you commonly have two lines of code that read the top of a stack and decrement a *stackTop* variable, put them into a *PopStack()* routine. Design the system so that either could be performed first, and then create a routine to hide the information about which happens to be performed first.

Hide pointer operations

Pointer operations tend to be hard to read and error prone. By isolating them in routines (or a class, if appropriate), you can concentrate on the intent of the operation rather than the mechanics of pointer manipulation. Also, if the operations are done in only one place, you can be more certain that the code is correct. If you find a better data type than pointers, you can change the program without traumatizing the routines that would have used the pointers.

182
183
184
185
186***Improve portability***

Use of routines isolates nonportable capabilities, explicitly identifying and isolating future portability work. Nonportable capabilities include nonstandard language features, hardware dependencies, operating-system dependencies, and so on.

187
188
189
190
191***Simplify complicated boolean tests***

Understanding complicated boolean tests in detail is rarely necessary for understanding program flow. Putting such a test into a function makes the code more readable because (1) the details of the test are out of the way and (2) a descriptive function name summarizes the purpose of the test.

192
193
194

Giving the test a function of its own emphasizes its significance. It encourages extra effort to make the details of the test readable inside its function. The result is that both the main flow of the code and the test itself become clearer.

195
196
197
198
199
200***Improve performance***

You can optimize the code in one place instead of several places. Having code in one place means that a single optimization benefits all the routines that use that routine, whether they use it directly or indirectly. Having code in one place makes it practical to recode the routine with a more efficient algorithm or in a faster, more efficient language such as assembler.

201 **CROSS-REFERENCE** For
202 details on information hiding,
203 see “Hide Secrets
204 (Information Hiding)” in
205 Section 5.3.***To ensure all routines are small?***

No. With so many good reasons for putting code into a routine, this one is unnecessary. In fact, some jobs are performed better in a single large routine. (The best length for a routine is discussed in Section 7.4, “How Long Can a Routine Be?”)

206
207**KEY POINT**209
210
211

Operations That Seem Too Simple to Put Into Routines

One of the strongest mental blocks to creating effective routines is a reluctance to create a simple routine for a simple purpose. Constructing a whole routine to contain two or three lines of code might seem like overkill. But experience shows how helpful a good small routine can be.

212
213

Small routines offer several advantages. One is that they improve readability. I once had the following single line of code in about a dozen places in a program:

214
215
216
217

Pseudocode Example of a Calculation

```
Points = deviceUnits * ( POINTS_PER_INCH / DeviceUnitsPerInch() )
```

This is not the most complicated line of code you’ll ever read. Most people would eventually figure out that it converts a measurement in device units to a

218 measurement in points. They would see that each of the dozen lines did the same
219 thing. It could have been clearer, however, so I created a well-named routine to
220 do the conversion in one place:

221 Pseudocode Example of a Calculation Converted to a Function

```
222 DeviceUnitsToPoints( deviceUnits Integer ): Integer;  
223 begin  
224     DeviceUnitsToPoints = deviceUnits *  
225         ( POINTS_PER_INCH / DeviceUnitsPerInch() )  
226 end function
```

227 When the routine was substituted for the inline code, the dozen lines of code all
228 looked more or less like this one:

229 Pseudocode Example of a Function Call to a Calculation Function

```
230 points = DeviceUnitsToPoints( deviceUnits )  
231 which was more readable—even approaching self-documenting.
```

232 This example hints at another reason to put small operations into functions:
233 Small operations tend to turn into larger operations. I didn't know it when I
234 wrote the routine, but under certain conditions and when certain devices were
235 active, *DeviceUnitsPerInch()* returned 0. That meant I had to account for division
236 by zero, which took three more lines of code:

237 Pseudocode Example of a Calculation that Expands Under Maintenance

```
238 DeviceUnitsToPoints( deviceUnits: Integer ): Integer;  
239     if ( DeviceUnitsPerInch() <> 0 )  
240         DeviceUnitsToPoints = deviceUnits *  
241             ( POINTS_PER_INCH / DeviceUnitsPerInch() )  
242     else  
243         DeviceUnitsToPoints = 0  
244     end if  
245 end function
```

246 If that original line of code had still been in a dozen places, the test would have
247 been repeated a dozen times, for a total of 36 new lines of code. A simple routine
248 reduced the 36 new lines to 3.

249 Summary of Reasons to Create a Routine

250 Here's a summary list of the valid reasons for creating a routine:

- 251 • Reduce complexity
- 252 • Make a section of code readable
- 253 • Avoid duplicate code

- 254 • Hide sequences
255 • Hide pointer operations
256 • Improve portability
257 • Simplify complicated boolean tests
258 • Improve performance

259 In addition, many of the reasons to create a class are also good reasons to create
260 a routine:

- 261 • Isolate complexity
262 • Hide implementation details
263 • Limit effects of changes
264 • Hide global data
265 • Make central points of control
266 • Facilitate reusable code
267 • To accomplish a specific refactoring

268 7.2 Design at the Routine Level

269 The concept of cohesion has been largely superceded by the concept of
270 abstraction at the class level, but cohesion is still alive and well as the workhorse
271 design heuristic at the individual-routine level.

272 **CROSS-REFERENCE** For
273 a discussion of cohesion in
274 general, see “Aim for Strong
Cohesion” in Section 5.3.
275
276
277
278

For routines, cohesion refers to how closely the operations in a routine are related. Some programmers prefer the term “strength”: How strongly related are the operations in a routine? A function like *Cosine()* is perfectly cohesive because the whole routine is dedicated to performing one function. A function like *CosineAndTan()* has lower cohesion because it tries to do more than one thing. The goal is to have each routine do one thing well and not do anything else.

279 The idea of cohesion was introduced in a paper by Wayne Stevens, Glenford
280 Myers, and Larry Constantine (1974). Other, more modern concepts including
281 abstraction and encapsulation tend to yield more insight at the class level, but
282 cohesion is still a workhorse concept for the design of routines.

283 **HARD DATA**
284
285

The payoff is higher reliability. One study of 450 routines found that 50 percent of the highly cohesive routines were fault free, whereas only 18 percent of routines with low cohesion were fault free (Card, Church, and Agresti 1986).

286 Another study of a different 450 routines (which is just an unusual coincidence)
287 found that routines with the highest coupling-to-cohesion ratios had 7 times as
288 many errors as those with the lowest coupling-to-cohesion ratios and were 20
289 times as costly to fix (Selby and Basili 1991).

290 Discussions about cohesion typically refer to several levels of cohesion.
291 Understanding the concepts is more important than remembering specific terms.
292 Use the concepts as aids in thinking about how to make routines as cohesive as
293 possible.

294 *Functional cohesion* is the strongest and best kind of cohesion, occurring when a
295 routine performs one and only one operation. Examples of highly cohesive
296 routines include *sin()*, *GetCustomerName()*, *EraseFile()*,
297 *CalculateLoanPayment()*, and *AgeFromBirthday()*. Of course, this evaluation of
298 their cohesion assumes that the routines do what their names say they do—if
299 they do anything else, they are less cohesive and poorly named.

300 Several other kinds of cohesion are normally considered to be less than ideal:

301 *Sequential cohesion* exists when a routine contains operations that must be
302 performed in a specific order, that share data from step to step, and that don't
303 make up a complete function when done together.

304 An example of sequential cohesion is a routine that calculates an employee's age
305 and time to retirement, given a birth date. If the routine calculates the age and
306 then uses that result to calculate the employee's time to retirement, it has
307 sequential cohesion. If the routine calculates the age and then calculates the time
308 to retirement in a completely separate computation that happens to use the same
309 birth-date data, it has only communicational cohesion.

310 How would you make the routine functionally cohesive? You'd create separate
311 routines to compute an employee's age given a birth date, and time to retirement
312 given a birth date. The time-to-retirement routine could call the age routine.
313 They'd both have functional cohesion. Other routines could call either routine or
314 both routines.

315 *Communicational cohesion* occurs when operations in a routine make use of the
316 same data and aren't related in any other way. If a routine prints a summary
317 report and then reinitializes the summary data passed into it, the routine has
318 communicational cohesion; the two operations are related only by the fact that
319 they use the same data.

320 To give this routine better cohesion, the summary data should be reinitialized
321 close to where it's created, which shouldn't be in the report-printing routine.

322 Split the operations into individual routines. The first prints the report. The
323 second reinitializes the data, close to the code that creates or modifies the data.
324 Call both routines from the higher-level routine that originally called the
325 communicationally cohesive routine.

326 *Temporal cohesion* occurs when operations are combined into a routine because
327 they are all done at the same time. Typical examples would be *Startup()*,
328 *CompleteNewEmployee()*, and *Shutdown()*. Some programmers consider
329 temporal cohesion to be unacceptable because it's sometimes associated with
330 bad programming practices such as having a hodgepodge of code in a *Startup()*
331 routine.

332 To avoid this problem, think of temporal routines as organizers of other events.
333 The *Startup()* routine, for example, might read a configuration file, initialize a
334 scratch file, set up a memory manager, and show an initial screen. To make it
335 most effective, have the temporally cohesive routine call other routines to
336 perform specific activities rather than performing the operations directly itself.
337 That way, it will be clear that the point of the routine is to orchestrate activities
338 rather than to do them directly.

339 This example raises the issue of choosing a name that describes the routine at the
340 right level of abstraction. You could decide to name the routine
341 *ReadConfigFileInitScratchFileEtc()*, which would imply that the routine had
342 only coincidental cohesion. If you name it *Startup()*, however, it would be clear
343 that it had a single purpose and clear that it had functional cohesion.

344 The remaining kinds of cohesion are generally unacceptable. They result in code
345 that's poorly organized, hard to debug, and hard to modify. If a routine has bad
346 cohesion, it's better to put effort into a rewrite to have better cohesion than
347 investing in a pinpoint diagnosis of the problem. Knowing what to avoid can be
348 useful, however, so here are the unacceptable kinds of cohesion:

349 *Procedural cohesion* occurs when operations in a routine are done in a specified
350 order. An example is a routine that gets an employee name, then an address, and
351 then a phone number. The order of these operations is important only because it
352 matches the order in which the user is asked for the data on the input screen.
353 Another routine gets the rest of the employee data. The routine has procedural
354 cohesion because it puts a set of operations in a specified order and the
355 operations don't need to be combined for any other reason.

356 To achieve better cohesion, put the separate operations into their own routines.
357 Make sure that the calling routine has a single, complete job:
358 *GetEmployeeData()* rather than *GetFirstPartOfEmployeeData()*. You'll probably
359 need to modify the routines that get the rest of the data too. It's common to

360 modify two or more original routines before you achieve functional cohesion in
361 any of them.

362 *Logical cohesion* occurs when several operations are stuffed into the same
363 routine and one of the operations is selected by a control flag that's passed in.
364 It's called logical cohesion because the control flow or "logic" of the routine is
365 the only thing that ties the operations together—they're all in a big *if* statement
366 or *case* statement together. It isn't because the operations are logically related in
367 any other sense. Considering that the defining attribute of logical cohesion is that
368 the operations are unrelated, a better name might *illogical cohesion*.

369 One example would be an *InputAll()* routine that input customer names,
370 employee time-card information, or inventory data depending on a flag passed to
371 the routine. Other examples would be *ComputeAll()*, *EditAll()*, *PrintAll()*, and
372 *SaveAll()*. The main problem with such routines is that you shouldn't need to
373 pass in a flag to control another routine's processing. Instead of having a routine
374 that does one of three distinct operations, depending on a flag passed to it, it's
375 cleaner to have three routines, each of which does one distinct operation. If the
376 operations use some of the same code or share data, the code should be moved
377 into a lower-level routine and the routines should be packaged into a class.

378 **CROSS-REFERENCE** While
379 the routine might have
380 better cohesion, a higher-
381 level design issue is whether
382 the system should be using a
383 *case* statement instead of
384 polymorphism. For more on
385 this issue, see "Replace
386 conditionals with
387 polymorphism (especially
388 repeated *case* statements)" in
389 Section 24.4.

390 None of these terms are magical or sacred. Learn the ideas rather than the
391 terminology. It's nearly always possible to write routines with functional
392 cohesion, so focus your attention on functional cohesion for maximum benefit.

393

394 **CROSS-REFERENCE** For
395 details on naming variables,
see Chapter 11, "The Power
396 of Variable Names."

397

398

399

400

401

402

403

404 **CROSS-REFERENCE** For
405 details on creating good
406 variable names, see Chapter
407 11, "The Power of Variable
Names."

408

409

410

411 KEY POINT

412

413

414

415

416

417

418

419

420

421

422

423

424

425

7.3 Good Routine Names

A good name for a routine clearly describes everything the routine does. Here are guidelines for creating effective routine names.

Describe everything the routine does

In the routine's name, describe all the outputs and side effects. If a routine computes report totals and opens an output file, *ComputeReportTotals()* is not an adequate name for the routine. *ComputeReportTotalsAndOpenOutputFile()* is an adequate name but is too long and silly. If you have routines with side effects, you'll have many long, silly names. The cure is not to use less-descriptive routine names; the cure is to program so that you cause things to happen directly rather than with side effects.

Avoid meaningless or wishy-washy verbs

Some verbs are elastic, stretched to cover just about any meaning. Routine names like *HandleCalculation()*, *PerformServices()*, *ProcessInput()*, and *DealWithOutput()* don't tell you what the routines do. At the most, these names tell you that the routines have something to do with calculations, services, input, and output. The exception would be when the verb "handle" was used in the specific technical sense of handling an event.

Sometimes the only problem with a routine is that its name is wishy-washy; the routine itself might actually be well designed. If *HandleOutput()* is replaced with *FormatAndPrintOutput()*, you have a pretty good idea of what the routine does.

In other cases, the verb is vague because the operations performed by the routine are vague. The routine suffers from a weakness of purpose, and the weak name is a symptom. If that's the case, the best solution is to restructure the routine and any related routines so that they all have stronger purposes and stronger names that accurately describe them.

Make names of routines as long as necessary

Research shows that the optimum average length for a variable name is 9 to 15 characters. Routines tend to be more complicated than variables, and good names for them tend to be longer. Michael Rees of the University of Southampton thinks that an average of 20 to 35 characters is a good nominal length (Rees 1982). An average length of 15 to 20 characters is probably more realistic, but clear names that happened to be longer would be fine.

426 **CROSS-REFERENCE** For
427 the distinction between
428 procedures and functions, see
429 Section 7.6, “Special
430 Considerations in the Use of
Functions” later in this
chapter.

431

432

433

434

435

436

437

438

439

440

441

442

443

444

445

446

447

448

449

450

451 **CROSS-REFERENCE** For
a similar list of opposites in
452 variable names, see
“Common Opposites in
453 Variable Names” in Section
454 11.1.

455

456

457

458

459

460

461

To name a function, use a description of the return value

A function returns a value, and the function should be named for the value it returns. For example, *cos()*, *customerId.Next()*, *printer.IsReady()*, and *pen.CurrentColor()* are all good function names that indicate precisely what the functions return.

To name a procedure, use a strong verb followed by an object

A procedure with functional cohesion usually performs an operation on an object. The name should reflect what the procedure does, and an operation on an object implies a verb-plus-object name. *PrintDocument()*, *CalcMonthlyRevenues()*, *CheckOrderInfo()*, and *RepaginateDocument()* are samples of good procedure names.

In object-oriented languages, you don’t need to include the name of the object in the procedure name because the object itself is included in the call. You invoke routines with statements like *document.Print()*, *orderInfo.Check()*, and *monthlyRevenues.Calc()*. Names like *document.PrintDocument()* are redundant and can become inaccurate when they’re carried through to derived classes. If *Check* is a class derived from *Document*, *check.Print()* seems clearly to be printing a check, whereas *check.PrintDocument()* sounds like it might be printing a checkbook register or monthly statement—but it doesn’t sound like it’s printing a check.

Use opposites precisely

Using naming conventions for opposites helps consistency, which helps readability. Opposite-pairs like *first/last* are commonly understood. Opposite-pairs like *FileOpen()* and *_lclose()* (from the Windows 3.1 software developer’s kit) are not symmetrical and are confusing. Here are some common opposites:

- add/remove
- begin/end
- create/destroy
- first/last
- get/put
- get/set
- increment/decrement
- insert/delete
- lock/unlock
- min/max
- next/previous

- 462 ● old/new
- 463 ● open/close
- 464 ● show/hide
- 465 ● source/target
- 466 ● start/stop
- 467 ● up/down

468 *Establish conventions for common operations*

469 In some systems, it's important to distinguish among different kinds of
470 operations. A naming convention is often the easiest and most reliable way of
471 indicating these distinctions.

472 The code on one of my projects assigned each object a unique identifier. We
473 neglected to establish a convention for naming the routines that would return the
474 object identifier, so we had routine names like these:

```
475       employee.id.Get()  
476       dependent.GetId()  
477       supervisor()  
478       candidate.id()
```

479 The *Employee* class exposed its *id* object, which in turn exposed its *Get()*
480 routine. The *Dependent* class exposed a *GetId()* routine. The *Supervisor* class
481 made the *id* its default return value. The *Candidate* class made use of the fact
482 that the *id* object's default return value was the *id*, and exposed the *id* object. By
483 the middle of the project, no one could remember which of these routines was
484 supposed to be used on which object, but by that time too much code had been
485 written to go back and make everything consistent. Consequently, every person
486 on the team had to devote an unnecessary amount of gray matter to remembering
487 the inconsequential detail of which syntax was used on which class to retrieve
488 the *id*. A naming convention for retrieving *ids* would have eliminated this
489 annoyance.

490 **7.4 How Long Can a Routine Be?**

491 On their way to America, the Pilgrims argued about the best maximum length for
492 a routine. After arguing about it for the entire trip, they arrived at Plymouth Rock
493 and started to draft the Mayflower Compact. They still hadn't settled the
494 maximum-length question, and since they couldn't disembark until they'd signed
495 the compact, they gave up and didn't include it. The result has been an
496 interminable debate ever since about how long a routine can be.

497 The theoretical best maximum length is often described as one or two pages of
498 program listing, 66 to 132 lines. In this spirit, IBM once limited routines to 50
499 lines, and TRW limited them to two pages (McCabe 1976). Modern programs
500 tend to have volumes of extremely short routines mixed in with a few longer
501 routines. Long routines are far from extinct, however. In the Spring of 2003, I
502 visited two client sites within a month. Programmers at one site were wrestling
503 with a routine that was about 4,000 lines of code long, and programmers at the
504 other site were trying to tame a routine that was more than 12,000 lines long!

505 A mountain of research on routine length has accumulated over the years, some
506 of which is applicable to modern programs, and some of which isn't:

- 507 | **HARD DATA**
-
- 508 • A study by Basili and Perricone found that routine size was inversely
509 correlated with errors; as the size of routines increased (up to 200 lines of
510 code), the number of errors per line of code decreased (Basili and Perricone
1984).
- 511 • Another study found that routine size was not correlated with errors, even
512 though structural complexity and amount of data were correlated with errors
513 (Shen et al. 1985).
- 514 • A 1986 study found that small routines (32 lines of code or fewer) were not
515 correlated with lower cost or fault rate (Card, Church, and Agresti 1986;
516 Card and Glass 1990). The evidence suggested that larger routines (65 lines
517 of code or more) were cheaper to develop per line of code.
- 518 • An empirical study of 450 routines found that small routines (those with
519 fewer than 143 source statements, including comments) had 23 percent more
520 errors per line of code than larger routines but were 2.4 times less expensive
521 to fix than larger routines (Selby and Basili 1991).
- 522 • Another study found that code needed to be changed least when routines
523 averaged 100 to 150 lines of code (Lind and Vairavan 1989).
- 524 • A study at IBM found that the most error-prone routines were those that
525 were larger than 500 lines of code. Beyond 500 lines, the error rate tended to
526 be proportional to the size of the routine (Jones 1986a).

527 Where does all this leave the question of routine length in object-oriented
528 programs? A large percentage of routines in object-oriented programs will be
529 accessor routines, which will be very short. From time to time, a complex
530 algorithm will lead to a longer routine, and in those circumstances, the routine
531 should be allowed to grow organically up to 100-200 lines. (A line is a
532 noncomment, nonblank line of source code.) Decades of evidence say that
533 routines of such length are no more error prone than shorter routines. Let issues
534 such as depth of nesting, number of variables, and other complexity-related

535 considerations dictate the length of the routine rather than imposing a length
536 restriction per se.

537 If you want to write routines longer than about 200 lines, be careful. None of the
538 studies that reported decreased cost, decreased error rates, or both with larger
539 routines distinguished among sizes larger than 200 lines, and you're bound to
540 run into an upper limit of understandability as you pass 200 lines of code.

541 7.5 How to Use Routine Parameters

542 **HARD DATA**
543
544 Interfaces between routines are some of the most error-prone areas of a program.
545 One often-cited study by Basili and Perricone (1984) found that 39 percent of all
errors were internal interface errors—errors in communication between routines.
Here are a few guidelines for minimizing interface problems:

546 *Put parameters in input-modify-output order*

547 **CROSS-REFERENCE** For
548 details on documenting
549 routine parameters, see
“Commenting Routines” in
550 Section 32.5. For details on
551 formatting parameters, see
Section 31.7, “Laying Out
552 Routines.”

Instead of ordering parameters randomly or alphabetically, list the parameters
that are input-only first, input-and-output second, and output-only third. This
ordering implies the sequence of operations happening within the routine—
inputting data, changing it, and sending back a result. Here are examples of
parameter lists in Ada:

Ada Example of Parameters in Input-Modify-Output Order

```
procedure InvertMatrix(
    originalMatrix: in Matrix;
    resultMatrix: out Matrix
);
...
procedure ChangeSentenceCase(
    desiredCase: in StringCase;
    sentence: in out Sentence
);
...
procedure PrintPageNumber(
    pageNumber: in Integer;
    status: out StatusType
);
```

This ordering convention conflicts with the C-library convention of putting the
modified parameter first. The input-modify-output convention makes more sense
to me, but if you consistently order parameters in some way, you still do the
readers of your code a service.

573 **Create your own in and out keywords**
574 Other modern languages don't support the *in* and *out* keywords like Ada does. In
575 those languages, you might still be able to use the preprocessor to create your
576 own *in* and *out* keywords. Here's how that could be done in C++:

577 **C++ Example of Defining Your Own In and Out Keywords**

```
578 #define IN
579 #define OUT

580

581 void InvertMatrix(
582     IN Matrix originalMatrix,
583     OUT Matrix *resultMatrix
584 );
585 ...
586

587 void ChangeSentenceCase(
588     IN StringCase desiredCase,
589     IN OUT Sentence *sentenceToEdit
590 );
591 ...
592

593 void PrintPageNumber(
594     IN int pageNumber,
595     OUT StatusType &status
596 );
```

In this case, the *IN* and *OUT* macro-keywords are used for documentation purposes. To make the value of a parameter changeable by the called routine, the parameter still needs to be passed as a pointer or as a reference parameter.

600 **If several routines use similar parameters, put the similar parameters in a 601 consistent order**

602 The order of routine parameters can be a mnemonic, and inconsistent order can
603 make parameters hard to remember. For example, in C, the *fprintf()* routine is the
604 same as the *printf()* routine except that it adds a file as the first argument. A
605 similar routine, *fputs()*, is the same as *puts()* except that it adds a file as the last
606 argument. This is an aggravating, pointless difference that makes the parameters
607 of these routines harder to remember than they need to be.

608 On the other hand, the routine *strncpy()* in C takes the arguments target string,
609 source string, and maximum number of bytes, in that order, and the routine
610 *memcpy()* takes the same arguments in the same order. The similarity between
611 the two routines helps in remembering the parameters in either routine.

612
613
614

In Microsoft Windows programming, most of the Windows routines take a
“handle” as their first parameter. The convention is easy to remember and makes
each routine’s argument list easier to remember.

615
616 **HARD DATA**

617
618
619
620

If you pass a parameter to a routine, use it. If you aren’t using it, remove the
parameter from the routine interface. Unused parameters are correlated with an
increased error rate. In one study, 46 percent of routines with no unused
variables had no errors. Only 17 to 29 percent of routines with more than one
unreferenced variable had no errors (Card, Church, and Agresti 1986).

621
622
623
624
625
626
627
628

This rule to remove unused parameters has two exceptions. First, if you’re using
function pointers in C++, you’ll have several routines with identical parameter
lists. Some of the routines might not use all the parameters. That’s OK. Second,
if you’re compiling part of your program conditionally, you might compile out
parts of a routine that use a certain parameter. Be nervous about this practice, but
if you’re convinced it works, that’s OK too. In general, if you have a good
reason not to use a parameter, go ahead and leave it in place. If you don’t have a
good reason, make the effort to clean up the code.

629
630
631
632

Put status or error variables last

By convention, status variables and variables that indicate an error has occurred
go last in the parameter list. They are incidental to the main purpose of the
routine, and they are output-only parameters, so it’s a sensible convention.

633
634
635
636
637

Don’t use routine parameters as working variables

It’s dangerous to use the parameters passed to a routine as working variables.
Use local variables instead. For example, in the Java fragment below, the
variable *InputVal* is improperly used to store intermediate results of a
computation.

638 **Java Example of Improper Use of Input Parameters**

639
640
641
642
643 *At this point, inputVal no*
644 *longer contains the value that*
645 *was input.*

```
int Sample( int inputVal ) {  
    inputVal = inputVal * CurrentMultiplier( inputVal );  
    inputVal = inputVal + CurrentAdder( inputVal );  
    ...  
    return inputVal;  
}
```

646
647
648
649
650

inputVal in this code fragment is misleading because by the time execution
reaches the last line, *inputVal* no longer contains the input value; it contains a
computed value based in part on the input value, and it is therefore misnamed. If
you later need to modify the routine to use the original input value in some other
place, you’ll probably use *inputVal* and assume that it contains the original input
value when it actually doesn’t.

651 How do you solve the problem? Can you solve it by renaming *inputVal*?
652 Probably not. You could name it something like *workingVal*, but that's an
653 incomplete solution because the name fails to indicate that the variable's original
654 value comes from outside the routine. You could name it something ridiculous
655 like *InputValThatBecomesWorkingVal* or give up completely and name it *X* or
656 *Val*, but all these approaches are weak.

657 A better approach is to avoid current and future problems by using working
658 variables explicitly. The following code fragment demonstrates the technique:

659 Java Example of Good Use of Input Parameters

```
660 int Sample( int inputVal ) {  
661     int workingVal = inputVal;  
662     workingVal = workingVal * CurrentMultiplier( workingVal );  
663     workingVal = workingVal + CurrentAdder( workingVal );  
664     ...  
665     If you need to use the original  
666     value of inputVal here or  
667     somewhere else, it's still  
668     available.  
669     ...  
670     return workingVal;  
671 }
```

672 Introducing the new variable *workingVal* clarifies the role of *inputVal* and
673 eliminates the chance of erroneously using *inputVal* at the wrong time. (Don't
674 take this reasoning as a justification for literally naming a variable *workingVal*.
675 In general, *workingVal* is a terrible name for a variable, and the name is used in
676 this example only to make the variable's role clear.)

677 Assigning the input value to a working variable emphasizes where the value
678 comes from. It eliminates the possibility that a variable from the parameter list
679 will be modified accidentally. In C++, this practice can be enforced by the
680 compiler using the keyword *const*. If you designate a parameter as *const*, you're
681 not allowed to modify its value within a routine.

682 **CROSS-REFERENCE** For
683 details on interface
684 assumptions, see the
685 introduction to Chapter 8,
686 "Defensive Programming."
687 For details on documentation,
688 see Chapter 32, "Self-
689 Documenting Code."

689 *Document interface assumptions about parameters*

690 If you assume the data being passed to your routine has certain characteristics,
691 document the assumptions as you make them. It's not a waste of effort to
692 document your assumptions both in the routine itself and in the place where the
693 routine is called. Don't wait until you've written the routine to go back and write
694 the comments—you won't remember all your assumptions. Even better than
695 commenting your assumptions, use assertions to put them into code.

696 What kinds of interface assumptions about parameters should you document?

- 697 • Whether parameters are input-only, modified, or output-only
- 698 • Units of numeric parameters (inches, feet, meters, and so on)

- 689
- 690
- 691
- Meanings of status codes and error values if enumerated types aren't used
 - Ranges of expected values
 - Specific values that should never appear

HARD DATA

692

693

694

695

696

697

Seven is a magic number for people's comprehension. Psychological research has found that people generally cannot keep track of more than about seven chunks of information at once (Miller 1956). This discovery has been applied to an enormous number of disciplines, and it seems safe to conjecture that most people can't keep track of more than about seven routine parameters at once.

698

699

700

701

702

In practice, how much you can limit the number of parameters depends on how your language handles complex data types. If you program in a modern language that supports structured data, you can pass a composite data type containing 13 fields and think of it as one mental "chunk" of data. If you program in a more primitive language, you might need to pass all 13 fields individually.

703 **CROSS-REFERENCE** For details on how to think about interfaces, see "Good Abstraction" in Section 6.2.

704

705

706

707

If you find yourself consistently passing more than a few arguments, the coupling among your routines is too tight. Design the routine or group of routines to reduce the coupling. If you are passing the same data to many different routines, group the routines into a class and treat the frequently used data as class data.

708

709

710

711

712

Consider an input, modify, and output naming convention for parameters

If you find that it's important to distinguish among input, modify, and output parameters, establish a naming convention that identifies them. You could prefix them with *i_*, *m_*, and *o_*. If you're feeling verbose, you could prefix them with *Input_*, *Modify_*, and *Output_*.

713

714

715

716

717

718

Pass the variables or objects that the routine needs to maintain its interface abstraction

There are two competing schools of thought about how to pass parameters from an object to a routine. Suppose you have an object that exposes data through 10 access routines, and the called routine needs 3 of those data elements to do its job.

719

720

721

722

723

724

Proponents of the first school of thought argue that only the 3 specific elements needed by the routine should be passed. They argue that that will keep the connections between routines to a minimum, reduce coupling, and make them easier to understand, easier to reuse, and so on. They say that passing the whole object to a routine violates the principle of encapsulation by potentially exposing all 10 access routines to the routine that's called.

725 Proponents of the second school argue that the whole object should be passed.
726 They argue that the interface can remain more stable if the called routine has the
727 flexibility to use additional members of the object without changing the routine's
728 interface. They argue that passing 3 specific elements violates encapsulation by
729 exposing which specific data elements the routine is using.

730 I think both these rules are simplistic and miss the most important consideration,
731 which is, *what abstraction is presented by the routine's interface?*

- 732
- 733 • If the abstraction is that the routine expects you to have 3 specific data
734 elements, and it is only a coincidence that those 3 elements happen to be
735 provided by the same object, then you should pass the 3 specific data
elements individually.
 - 736 • If the abstraction is that you will always have that particular object in hand
737 and the routine will do something or other with that object, then you truly do
738 break the abstraction when you expose the three specific data elements.

739 If you're passing the whole object and you find yourself creating the object,
740 populating it with the 3 elements needed by the called routine, and then pulling
741 those elements out of the object after the routine is called, that's an indication
742 that you should be passing the 3 specific elements rather than the whole object.
743 (Generally code that "sets up" for a call to a routine or "takes down" after a call
744 to a routine is an indication that the routine is not well designed.)

745 If you find yourself frequently changing the parameter list to the routine, with
746 the parameters coming from the same object each time, that's an indication that
747 you should be passing the whole object rather than specific elements.

748 **Used named parameters**

749 In some languages, you can explicitly associate formal parameters with actual
750 parameters. This makes parameter usage more self-documenting and helps avoid
751 errors from mismatching parameters. Here's an example in Visual Basic:

752 **Visual Basic Example of Explicitly Identifying Parameters**

753

754 *Here's where the formal*
755 *parameters are declared.*

```
756 Private Function Distance3d( _
757     ByVal xDistance As Coordinate, _
758     ByVal yDistance As Coordinate, _
759     ByVal zDistance As Coordinate _
760 )
761     ...
762     End Function
763     ...
764 Private Function Velocity( _
765     ByVal latitude as Coordinate, _
```

```
763     ByVal longitude as Coordinate, _  
764     ByVal elevation as Coordinate _  
765 )  
766 ...  
767     Here's where the actual  
768 parameters are mapped to the  
769 formal parameters.  
770 End Function
```

This technique is especially useful when you have longer-than-average lists of identically typed arguments, which increases the chances that you can insert a parameter mismatch without the compiler detecting it. Explicitly associating parameters may be overkill in many environments, but in safety-critical or other high-reliability environments the extra assurance that parameters match up the way you expect can be worthwhile.

Don't assume anything about the parameter-passing mechanism

Some hard-core nanosecond scrapers worry about the overhead associated with passing parameters and bypass the high-level language's parameter-passing mechanism. This is dangerous and makes code nonportable. Parameters are commonly passed on a system stack, but that's hardly the only parameter-passing mechanism that languages use. Even with stack-based mechanisms, the parameters themselves can be passed in different orders and each parameter's bytes can be ordered differently. If you fiddle with parameters directly, you virtually guarantee that your program won't run on a different machine.

Make sure actual parameters match formal parameters

Formal parameters, also known as dummy parameters, are the variables declared in a routine definition. Actual parameters are the variables or constants used in the actual routine calls.

A common mistake is to put the wrong type of variable in a routine call—for example, using an integer when a floating point is needed. (This is a problem only in weakly typed languages like C when you're not using full compiler warnings. Strongly typed languages such as C++ and Java don't have this problem.) When arguments are input only, this is seldom a problem; usually the compiler converts the actual type to the formal type before passing it to the routine. If it is a problem, usually your compiler gives you a warning. But in some cases, particularly when the argument is used for both input and output, you can get stung by passing the wrong type of argument.

Develop the habit of checking types of arguments in parameter lists and heeding compiler warnings about mismatched parameter types.

801 **7.6 Special Considerations in the Use of**

802 **Functions**

803 Modern languages such as C++, Java, and Visual Basic support both functions
804 and procedures. A function is a routine that returns a value; a procedure is a
805 routine that does not. This distinction is as much a semantic distinction as a
806 syntactic one. In C++, all routines are typically called “functions,” however, a
807 function with a *void* return type is semantically a procedure and should be
808 treated as such.

809 **When to Use a Function and When to Use a**

810 **Procedure**

811 Purists argue that a function should return only one value, just as a mathematical
812 function does. This means that a function would take only input parameters and
813 return its only value through the function itself. The function would always be
814 named for the value it returned, as *sin()*, *CustomerID()*, and *ScreenHeight()* are.
815 A procedure, on the other hand, could take input, modify, and output
816 parameters—as many of each as it wanted to.

817 A common programming practice is to have a function that operates as a
818 procedure and returns a status value. Logically, it works as a procedure, but
819 because it returns a value, it’s officially a function. For example, you might have
820 a routine called *FormatOutput()* used with a *report* object in statements like this
821 one:

822 if (report.FormatOutput(formattedReport) = Success) then ...
823 In this example, *report.FormatOutput()* operates as a procedure in that it has an
824 output parameter, *formattedReport*, but it is technically a function because the
825 routine itself returns a value. Is this a valid way to use a function? In defense of
826 this approach, you could maintain that the function return value has nothing to
827 do with the main purpose of the routine, formatting output, or with the routine
828 name, *report.FormatOutput()*; in that sense it operates more as a procedure does
829 even if it is technically a function. The use of the return value to indicate the
830 success or failure of the procedure is not confusing if the technique is used
831 consistently.

832 The alternative is to create a procedure that has a status variable as an explicit
833 parameter, which promotes code like this fragment:

```
834           report.FormatOutput( formattedReport, outputStatus )  
835           if ( outputStatus = Success ) then ...
```

836
837
838
839
840
I prefer the second style of coding, not because I'm hard-nosed about the
difference between functions and procedures but because it makes a clear
separation between the routine call and the test of the status value. To combine
the call and the test into one line of code increases the density of the statement
and correspondingly its complexity. The following use of a function is fine too:

841 outputStatus = report.FormatOutput(formattedReport)
842 if (outputStatus = Success) then ...

843 **KEY POINT**
844 In short, use a function if the primary purpose of the routine is to return the value
indicated by the function name. Otherwise, use a procedure.

845 Setting the Function's Return Value

846 Using a function creates the risk that the function will return an incorrect return
847 value. This usually happens when the function has several possible paths and one
848 of the paths doesn't set a return value.

849 *Check all possible return paths*

850 When creating a function, mentally execute each path to be sure that the function
851 returns a value under all possible circumstances. It's good practice to initialize
852 the return value at the beginning of the function to a default value—which
853 provides a safety net in the event of that the correct return value is not set.

854 *Don't return references or pointers to local data*

855 As soon as the routine ends and the local data goes out of scope, the reference or
856 pointer to the local data will be invalid. If an object needs to return information
857 about its internal data, it should save the information as class member data. It
858 should then provide accessor functions that return the values of the member data
859 items rather than references or pointers to local data.

860 7.7 Macro Routines and Inline Routines

861 **CROSS-REFERENCE** Even if your language doesn't
862 have a macro preprocessor,
863 you can build your own. For
864 details, see Section 30.5,

865 "Building Your Own
866 Programming Tools."

867
868

869

870

871

872

873

874

875

C++ Example of a Macro That Doesn't Expand Properly

```
#define Cube( a ) a*a*a
```

This macro has a problem. If you pass it nonatomic values for *a*, it won't do the multiplication properly. If you use the expression *Cube(x+1)*, it expands to *x+1 * x + 1 * x + 1*, which, because of the precedence of the multiplication and addition operators, is not what you want. A better but still not perfect version of the macro looks like this:

876

877

878

879

880

C++ Example of a Macro That Still Doesn't Expand Properly

```
#define Cube( a ) (a)*(a)*(a)
```

This is close, but still no cigar. If you use *Cube()* in an expression that has operators with higher precedence than multiplication, the *(a)*(a)*(a)* will be torn apart. To prevent that, enclose the whole expression in parentheses:

881

882

C++ Example of a Macro That Works

```
#define Cube( a ) ((a)*(a)*(a))
```

883

884

885

Surround multiple-statement macros with curly braces

A macro can have multiple statements, which is a problem if you treat it as if it were a single statement. Here's an example of a macro that's headed for trouble:

886

C++ Example of a Macro with Multiple Statements That Doesn't Work

```
#define LookupEntry( key, index ) \
    index = (key - 10) / 5; \
    index = min( index, MAX_INDEX ); \
    index = max( index, MIN_INDEX ); \
    ... \
    for ( entryCount = 0; entryCount < numEntries; entryCount++ ) \
        LookupEntry( entryCount, tableIndex[ entryCount ] );
```

This macro is headed for trouble because it doesn't work as a regular function would. As it's shown, the only part of the macro that's executed in the *for* loop is the first line of the macro:

898

899

```
    index = (key - 10) / 5;
```

To avoid this problem, surround the macro with curly braces, as shown here:

900

C++ Example of a Macro with Multiple Statements That Works

```
#define LookupEntry( key, index ) { \
    index = (key - 10) / 5; \
    index = min( index, MAX_INDEX ); \
    index = max( index, MIN_INDEX ); \
}
```

906
907
908
The practice of using macros as substitutes for function calls is generally
considered risky and hard to understand—bad programming practice—so use
this technique only if your specific circumstances require it.

909
910
**Name macros that expand to code like routines so that they can be replaced
by routines if necessary**

911
912
913
914
The C++-language convention for naming macros is to use all capital letters. If
the macro can be replaced by a routine, however, name it using the naming
convention for routines instead. That way you can replace macros with routines
and vice versa without changing anything but the routine involved.

915
916
917
918
Following this recommendation entails some risk. If you commonly use `++` and
`--` as side effects (as part of other statements), you'll get burned when you use
macros that you think are routines. Considering the other problems with side
effects, this is just one more reason to avoid using side effects.

919
Limitations on the Use of Macro Routines

920
Modern languages like C++ provide numerous alternatives to the use of macros:

- 921
922
923
924
925
926
927
KEY POINT
As Bjarne Stroustrup, designer of C++ points out, “Almost every macro
demonstrates a flaw in the programming language, in the program, or in the
programmer.... When you use macros, you should expect inferior service from
tools such as debuggers, cross-reference tools, and profilers” (Stroustrup 1997).
Macros are useful for supporting conditional compilation (see Section 8.6), but
careful programmers generally use a macro as an alternative to a routine only as
a last resort.
-

934
Inline Routines

935
936
937
938
C++ supports an *inline* keyword. An *inline* routine allows the programmer to
treat the code as a routine at code-writing time. But the compiler will convert
each instance of the routine into inline code at compile time. The theory is that
inline can help produce highly efficient code that avoids routine-call overhead.

939 Use *inline routines sparingly*

940 Inline routines violate encapsulation because C++ requires the programmer to
941 put the code for the implementation of the inline routine in the header file, which
942 exposes it to every programmer who uses the header file.

943 Inline routines require a routine's full code to be generated every time the routine
944 is invoked, which for an inline routine of any size will increase code size. That
945 can create problems of its own.

946 The bottom line on inlining for performance reasons is the same as the bottom
947 line on any other coding technique that's motivated by performance—profile the
948 code and measure the improvement. If the anticipated performance gain doesn't
949 justify the bother of profiling the code to verify the improvement, it doesn't
950 justify the erosion in code quality either.

CHECKLIST: High-Quality Routines

Big-Picture Issues

- Is the reason for creating the routine sufficient?
- Have all parts of the routine that would benefit from being put into routines
of their own been put into routines of their own?
- Is the routine's name a strong, clear verb-plus-object name for a procedure
or a description of the return value for a function?
- Does the routine's name describe everything the routine does?
- Have you established naming conventions for common operations?
- Does the routine have strong, functional cohesion—doing one and only one
thing and doing it well?
- Do the routines have loose coupling—are the routine's connections to other
routines small, intimate, visible, and flexible?
- Is the length of the routine determined naturally by its function and logic,
rather than by an artificial coding standard?

Parameter-Passing Issues

- Does the routine's parameter list, taken as a whole, present a consistent
interface abstraction?
- Are the routine's parameters in a sensible order, including matching the
order of parameters in similar routines?
- Are interface assumptions documented?
- Does the routine have seven or fewer parameters?
- Is each input parameter used?

6625.COM/0702
951 **CROSS-REFERENCE** This
is a checklist of
considerations about the
952 quality of the routine. For a
list of the steps used to build
953 a routine, see the checklist
954 "The Pseudocode
955 Programming Process" in
Chapter 9, page 000.
956

957

958

959

960

961

962

963

964

965

966

967

968

969

970

971

972

973

- 974 Is each output parameter used?
975 Does the routine avoid using input parameters as working variables?
976 If the routine is a function, does it return a valid value under all possible
977 circumstances?
-

979 **Key Points**

- 980
 - The most important reason to create a routine is to improve the intellectual
981 manageability of a program, and you can create a routine for many other
982 good reasons. Saving space is a minor reason; improved readability,
983 reliability, and modifiability are better reasons.

984 - Sometimes the operation that most benefits from being put into a routine of
985 its own is a simple one.

986 - The name of a routine is an indication of its quality. If the name is bad and
987 it's accurate, the routine might be poorly designed. If the name is bad and
988 it's inaccurate, it's not telling you what the program does. Either way, a bad
989 name means that the program needs to be changed.

990 - Functions should be used only when the primary purpose of the function is
991 to return the specific value described by the function's name.

992 - Careful programmers use macro routines and inline routines with care, and
993 only as a last resort.

8

Defensive Programming

Contents

- 8.1 Protecting Your Program From Invalid Inputs
- 8.2 Assertions
- 8.3 Error Handling Techniques
- 8.4 Exceptions
- 8.5 Barricade Your Program to Contain the Damage Caused by Errors
- 8.6 Debugging Aids
- 8.7 Determining How Much Defensive Programming to Leave in Production Code
- 8.8 Being Defensive About Defensive Programming

Related Topics

Information hiding: "Hide Secrets (Information Hiding)" in Section 5.3.

Design for change: "Identify Areas Likely to Change" in Section 5.3.

Software architecture: Section 3.5

High-level design: Chapter 5

Debugging: Chapter 23

KEY POINT

DEFENSIVE PROGRAMMING DOESN'T MEAN being defensive about your programming—"It does so work!" The idea is based on defensive driving. In defensive driving, you adopt the mind-set that you're never sure what the other drivers are going to do. That way, you make sure that if they do something dangerous you won't be hurt. You take responsibility for protecting yourself even when it might be the other driver's fault. In defensive programming, the main idea is that if a routine is passed bad data, it won't be hurt, even if the bad data is another routine's fault. More generally, it's the recognition that programs will have problems and modifications, and that a smart programmer will develop code accordingly.

29 This chapter describes how to protect yourself from the cold, cruel world of in-
30 valid data, events that can “never” happen, and other programmers’ mistakes. If
31 you’re an experienced programmer, you might skip the next section on handling
32 input data and begin with Section 8.2, which reviews the use of assertions.

33

34 8.1 Protecting Your Program From Invalid 35 Inputs

36 In school you might have heard the expression, “Garbage in, garbage out.” That
37 expression is essentially software development’s version of *caveat emptor*: let
the user beware.

38 **KEY POINT**

39 For production software, garbage in, garbage out isn’t good enough. A good
40 program never puts out garbage, regardless of what it takes in. A good program
41 uses “garbage in, nothing out”; “garbage in, error message out”; or “no garbage
42 allowed in” instead. By today’s standards, “garbage in, garbage out” is the mark
of a sloppy, nonsecure program.

43 There are three general ways to handle garbage in.

44 ***Check the values of all data from external sources***

45 When getting data from a file, a user, the network, or some other external inter-
46 face, check to be sure that the data falls within the allowable range. Make sure
47 that numeric values are within tolerances and that strings are short enough to
48 handle. If a string is intended to represent a restricted range of values (such as a
49 financial transaction ID or something similar), be sure that the string is valid for
50 its intended purpose; otherwise reject it. If you’re working on a secure applica-
51 tion, be especially leery of data that might attack your system: attempted buffer
52 overflows, injected SQL commands, injected html or XML code, integer over-
53 flows, and so on.

54 ***Check the values of all routine input parameters***

55 Checking the values of routine input parameters is essentially the same as check-
56 ing data that comes from an external source, except that the data comes from
57 another routine instead of from an external interface.

58 ***Decide how to handle bad inputs***

59 Once you’ve detected an invalid parameter, what do you do with it? Depending
60 on the situation, you might choose any of a dozen different approaches, which
61 are described in detail later in this chapter.

62 Defensive programming is useful as an adjunct to the other techniques for qual-
63 ity improvement described in this book. The best form of defensive coding is not

64
65
66
67
68
inserting errors in the first place. Using iterative design, writing pseudocode before code, and having low-level design inspections are all activities that help to prevent inserting defects. They should thus be given a higher priority than defensive programming. Fortunately, you can use defensive programming in combination with the other techniques.

69
70
71
72
As Figure 8-1 suggests, protecting yourself from seemingly small problems can make more of a difference than you might think. The rest of this chapter describes specific options for checking data from external sources, checking input parameters, and handling bad inputs.



73
74
F08xx01

75
Figure 8-1

76
77
78
79
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.

80

8.2 Assertions

81
82
83
84
85
An assertion is code that's used during development—usually a routine or macro—that allows a program to check itself as it runs. When an assertion is true, that means everything is operating as expected. When it's false, that means it has detected an unexpected error in the code. For example, if the system assumes that a customer-information file will never have more than 50,000 re-

86
87
88
89
cords, the program might contain an assertion that the number of records is less
than or equal to 50,000. As long as the number of records is less than or equal to
50,000, the assertion will be silent. If it encounters more than 50,000 records,
however, it will loudly “assert” that there is an error in the program.

90 **KEY POINT**

91
92
93
Assertions are especially useful in large, complicated programs and in high-
reliability programs. They enable programmers to more quickly flush out mis-
matched interface assumptions, errors that creep in when code is modified, and
so on.

94
95
96
97
An assertion usually takes two arguments: a boolean expression that describes
the assumption that’s supposed to be true and a message to display if it isn’t.
Here’s what a Java assertion would look like if the variable *denominator* were
expected to be nonzero:

98 Java Example of an Assertion

```
99 assert denominator != 0 : "denominator is unexpectedly equal to 0.;"
```

100 This assertion asserts that *denominator* is not equal to 0. The first argument,
101 *denominator != 0*, is a boolean expression that evaluates to *True* or *False*. The
102 second argument is a message to print if the first argument is *False*—that is, if
103 the assertion is false.

104 Use assertions to document assumptions made in the code and to flush out unex-
105 pected conditions. Assertions can be used to check assumptions like these:

- 106 • That an input parameter’s value falls within its expected range (or an output
107 parameter’s value does)
- 108 • That a file or stream is open (or closed) when a routine begins executing (or
109 when it ends executing)
- 110 • That a file or stream is at the beginning (or end) when a routine begins exe-
111 cuting (or when it ends executing)
- 112 • That a file or stream is open for read-only, write-only, or both read and write
- 113 • That the value of an input-only variable is not changed by a routine
- 114 • That a pointer is non-NULL
- 115 • That an array or other container passed into a routine can contain at least *X*
116 number of data elements
- 117 • That a table has been initialized to contain real values
- 118 • That a container is empty (or full) when a routine begins executing (or when
119 it finishes)

- 120 • That the results from a highly optimized, complicated routine match the re-
121 sults from a slower but clearly written routine
122 • Etc.

123 Of course, these are just the basics, and your own routines will contain many
124 more specific assumptions that you can document using assertions.

125 Normally, you don't want users to see assertion messages in production code;
126 assertions are primarily for use during development and maintenance. Assertions
127 are normally compiled into the code at development time and compiled out of
128 the code for production. During development, assertions flush out contradictory
129 assumptions, unexpected conditions, bad values passed to routines, and so on.
130 During production, they are compiled out of the code so that the assertions don't
131 degrade system performance.

132 Building Your Own Assertion Mechanism

133 **CROSS-REFERENCE** Building your own assertion routine is a good example of
134 programming "into" a language rather than just pro-
135 gramming "in" a language.
136 For more details on this dis-
137 tinction, see Section 34.4,
138 "Program Into Your Lan-
139 guage, Not In It."

140 C++ Example of an Assertion Macro

```
141 #define ASSERT( condition, message ) {  
142     if ( !(condition) ) {  
143         fprintf( stderr, "Assertion %s failed: %s\n",  
144             #condition, message );  
145         exit( EXIT_FAILURE );  
146     }  
147 }
```

Once you've written an assertion routine like this, you can call it with statements like the first one above.

148 Guidelines for Using Assertions

149 Here are some guidelines for using assertions:

150 *Use error handling code for conditions you expect to occur; use assertions 151 for conditions that should never occur*

152 Assertions check for conditions that should *never* occur. Error handling code
153 checks for off-nominal circumstances that might not occur very often, but that
154 have been anticipated by the programmer who wrote the code and that need to be
155 handled by the production code. Error-handling typically checks for bad input
data; assertions check for bugs in the code.

156
157
158
159
160

If error handling code is used to address an anomalous condition, the error handling will enable the program to respond to the error gracefully. If an assertion is fired for an anomalous condition, the corrective action is not merely to handle an error gracefully—the corrective action is to change the program's source code, recompile, and release a new version of the software.

161
162
163

A good way to think of assertions is as executable documentation—you can't rely on them to make the code work, but they can document assumptions more actively than program-language comments can.

164
165
166
167

Avoid putting executable code in assertions
Putting code into an assertion raises the possibility that the compiler will eliminate the code when you turn off the assertions. Suppose you have an assertion like this:

168 **CROSS-REFERENCE** You
could view this as one of
169 many problems associated
170 with putting multiple state-
171 ments on one line. For more
172 examples, see "Using Only
One Statement per Line" in
173 Section 31.5.

174
175
176

177
178 **FURTHER READING** For
much more on preconditions
and postconditions, see *Ob-
ject-Oriented Software Con-
struction* (Meyer 1997).

182
183
184

185
186
187

188
189
190
191

Visual Basic Example of a Dangerous Use of an Assertion

```
Debug.Assert( PerformAction() ) ' Couldn't perform action
```

The problem with this code is that, if you don't compile the assertions, you don't compile the code that performs the action. Put executable statements on their own lines, assign the results to status variables, and test the status variables instead. Here's an example of a safe use of an assertion:

Visual Basic Example of a Safe Use of an Assertion

```
actionPerformed = PerformAction()  
Debug.Assert( actionPerformed ) ' Couldn't perform action
```

Use assertions to document preconditions and postconditions

Preconditions and postconditions are part of an approach to program design and development known as "design by contract" (Meyer 1997). When preconditions and postconditions are used, each routine or class forms a contract with the rest of the program.

Preconditions are the properties that the client code of a routine or class promises will be true before it calls the routine or instantiates the object. Preconditions are the client code's obligations to the code it calls.

Postconditions are the properties that the routine or class promises will be true when it concludes executing. Postconditions are the routine or class's obligations to the code that uses it.

Assertions are a useful tool for documenting preconditions and postconditions. Comments could be used to document preconditions and postconditions, but, unlike comments, assertions can check dynamically whether the preconditions and postconditions are true.

192 In the example below, assertions are used to document the preconditions and
193 postcondition of the *Velocity* routine.

194 Visual Basic Example of Using Assertions to Document Preconditions 195 and Postconditions

```
196 Private Function Velocity ( _  
197     ByVal latitude As Single, _  
198     ByVal longitude As Single, _  
199     ByVal elevation As Single _  
200     ) As Single  
201  
202     ' Preconditions  
203     Debug.Assert ( -90 <= latitude And latitude <= 90 )  
204     Debug.Assert ( 0 <= longitude And longitude < 360 )  
205     Debug.Assert ( -500 <= elevation And elevation <= 75000 )  
206  
207     ...  
208  
209     ' Postconditions  
210     Debug.Assert ( 0 <= returnVelocity And returnVelocity <= 600 )  
211  
212     ' return value  
213     Velocity = returnVelocity  
214 End Function
```

215 If the variables *latitude*, *longitude*, and *elevation* were coming from an external
216 source, invalid values should be checked and handled by error handling code
217 rather than assertions. If the variables are coming from a trusted, internal source,
218 however, and the routine's design is based on the assumption that these values
219 will be within their valid ranges, then assertions are appropriate.

220
221 **CROSS-REFERENCE** For
222 more on robustness, see "Robustness vs. Correctness" in
223 Section 8.2, later in this chapter.

224
225
226
227
228
229
230
231
232
233

234 With test teams working across different geographic regions and subject to business pressures that result in test coverage that varies with each release, you can't
235 count on comprehensive regression testing, either.
236

237 In such circumstances, both assertions and error handling code might be used to
238 address the same error. In the source code for Microsoft Word, for example,
239 conditions that should always be true are asserted, but such errors are also han-
240 dled by error-handling code in case the assertion fails. For extremely large, com-
241 plex, long-lived applications like Word, assertions are valuable because they
242 help to flush out as many development-time errors as possible. But the applica-
243 tion is so complex (million of lines of code) and has gone through so many gen-
244 erations of modification that it isn't realistic to assume that every conceivable
245 error will be detected and corrected before the software ships, and so errors must
246 be handled in the production version of the system as well.

247 Here is an example of how that might work in the *Velocity* example.

248 **Visual Basic Example of Using Assertions to Document Preconditions**
249 **and Postconditions**

250 *Here is the assertion code.*

```
Private Function Velocity ( _  
    ByRef latitude As Single, _  
    ByRef longitude As Single, _  
    ByRef elevation As Single _  
) As Single  
  
    ' Preconditions  
    Debug.Assert ( -90 <= latitude And latitude <= 90 )  
    Debug.Assert ( 0 <= longitude And longitude < 360 )  
    Debug.Assert ( -500 <= elevation And elevation <= 75000 )  
    ...  
  
    ' Sanitize input data. Values should be within the ranges asserted above,  
    ' but If a value is not within its valid range, it will be changed to the  
    ' closest legal value  
    If ( latitude < -90 ) Then  
        latitude = -90  
    ElseIf ( latitude > 90 ) Then  
        latitude = 90  
    End If  
    If ( longitude < 0 ) Then  
        longitude = 0  
    ElseIf ( longitude > 360 ) Then  
        ...
```

255
256
257 *Here is the code that handles*
258 *bad input data at runtime.*
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273

274

8.3 Error Handling Techniques

275

276

277

278

279

280

Assertions are used to handle errors that should never occur in the code. How do you handle errors that you do expect to occur? Depending on the specific circumstances, you might want to return a neutral value, substitute the next piece of valid data, return the same answer as the previous time, substitute the closest legal value, log a warning message to a file, return an error code, call an error processing routine or object, display an error message, or shutdown.

281

Here are some more details on these options.

282

283

284

285

286

287

Return a neutral value

Sometimes the best response to bad data is to continue operating and simply return a value that's known to be harmless. A numeric computation might return 0. A string operation might return an empty string, or a pointer operation might return an empty pointer. A drawing routine that gets a bad input value for color might use the default background or foreground color.

288

289

290

291

292

293

294

Substitute the next piece of valid data

When processing a stream of data, some circumstances call for simply returning the next valid data. If you're reading records from a database and encounter a corrupted record, you might simply continue reading until you find a valid record. If you're taking readings from a thermometer 100 times per second and you don't get a valid reading one time, you might simply wait another 1/100th of a second and take the next reading.

295

296

297

298

299

300

Return the same answer as the previous time

If the thermometer-reading software doesn't get a reading one time, it might simply return the same value as last time. Depending on the application, temperatures might not be very likely to change much in 1/100th of a second. In a video game, if you detect a request to paint part of the screen an invalid color, you might simply return the same color used previously.

301

302

303

304

305

306

307

308

309

310

311

Substitute the closest legal value

In some cases, you might choose to return the closest legal value, as in the *Velocity* example earlier in this chapter. This is often a reasonable approach when taking readings from a calibrated instrument. The thermometer might be calibrated between 0 and 100 degrees Celsius, for example. If you detect a reading less than 0, you can substitute 0 which is the closest legal value. If you detect a value greater than 100, you can substitute 100. For a string operation, if a string length is reported to be less than 0, you could substitute 0. My car uses this approach to error handling whenever I back up. Since my speedometer doesn't show negative speeds, when I back up it simply shows a speed of 0—the closest legal value.

312 *Log a warning message to a file*

313 When bad data is detected, you might choose to log a warning message to a file
314 and then continue on. This approach can be used in conjunction with other tech-
315 niques like substituting the closest legal value or substituting the next piece of
316 valid data.

317 *Return an error code*

318 You could decide that only certain parts of a system will handle errors; other
319 parts will not handle errors locally; they will simply report that an error has been
320 detected and trust that some other routine higher up in the calling hierarchy will
321 handle the error. The specific mechanism for notifying the rest of the system that
322 an error has occurred could be any of the following:

- 323 • Set the value of a status variable
324 • Return status as the function's return value
325 • Throw an exception using the language's built-in exception mechanism

326 In this case, the specific error-reporting mechanism is less important than the
327 decision about which parts of the system will handle errors directly and which
328 will just report that they've occurred. If security is an issue, be sure that calling
329 routines always check return codes.

330 *Call an error processing routine/object*

331 Another approach is to centralize error handling in a global error handling rou-
332 tine or error handling object. The advantage of this approach is that error proc-
333 essing responsibility can be centralized, which can make debugging easier. The
334 tradeoff is that the whole program will know about this central capability and
335 will be coupled to it. If you ever want to reuse any of the code from the system
336 in another system, you'll have to drag the error handling machinery along with
337 the code you reuse.

338 This approach has an important security implication. If your code has encoun-
339 tered a buffer-overrun, it's possible that an attacker has compromised the address
340 of the handler routine or object. Thus, once a buffer overrun has occurred while
341 an application is running, it is no longer safe to use this approach.

342 *Display an error message wherever the error is encountered*

343 This approach minimizes error-handling overhead, however it does have the po-
344 tential to spread user interface messages through the entire application, which
345 can create challenges when you need to create a consistent user interface, try to
346 clearly separate the UI from the rest of the system, or try to localize the software
347 into a different language. Also, beware of telling a potential attacker of the sys-
348 tem too much. Attackers sometimes use error messages to discover how to attack
349 a system.

350 *Handle the error in whatever way works best locally*

351 Some designs call for handling all errors locally—the decision of which specific
352 error-handling method to use is left up to the programmer designing and imple-
353 menting the part of the system that encounters the error.

354 This approach provides individual developers with great flexibility, but it creates
355 a significant risk that the overall performance of the system will not satisfy its
356 requirements for correctness or robustness (more on this later). Depending on
357 how developers end up handling specific errors, this approach also has the poten-
358 tial to spread user interface code throughout the system, which exposes the pro-
359 gram to all the problems associated with displaying error messages.

360 *Shutdown*

361 Some systems shut down whenever they detect an error. This approach is useful
362 in safety critical applications. For example, if the software that controls radiation
363 equipment for treating cancer patients receives bad input data for the radiation
364 dosage, what is its best error-handling response? Should it use the same value as
365 last time? Should it use the closest legal value? Should it use a neutral value? In
366 this case, shutting down is the best option. We'd much prefer to reboot the ma-
367 chine than to run the risk of delivering the wrong dosage.

368 A similar approach can be used to improve security of Microsoft Windows. By
369 default, Windows continues to operate even when its security log is full. But you
370 can configure Windows to halt the server if the security log becomes full, which
371 can be appropriate in a security-critical environment.

372 Robustness vs. Correctness

373 Here's a brain teaser:

374 *Suppose an application displays graphic information on
375 a screen. An error condition results in a few pixels in the
376 lower right quadrant displaying in the wrong color. On next
377 update, the screen will refresh, and the pixels will be the right
378 color again. What is the best error processing approach?*

379 What do you think is the best approach? Is it to use the same value as last time?
380 Or perhaps to use the closest legal value? Suppose this error occurs inside a fast-
381 paced video game, and the next time the screen is refreshed the pixels will be
382 repainted to be the right color (which will occur within less than one second)? In
383 that case, choose an approach like using the same color as last time or using the
384 default background color.

385 Now suppose that the application is not a video game, but software that displays
386 X-rays. Would using the same color as last time be a good approach, or using the

387 default background color? Developers of that application would not want to run
388 the risk of having bad data on an X-ray, and so displaying an error message or
389 shutting down would be better ways to handle that kind of error.

390 The style of error processing that is most appropriate depends on the kind of
391 software the error occurs in and generally favors more correctness or more ro-
392 bustness. Developers tend to use these terms informally, but, strictly speaking,
393 these terms are at opposite ends of the scale from each other. *Correctness* means
394 never returning an inaccurate result; no result is better than an inaccurate result.
395 *Robustness* means always trying to do something that will allow the software to
396 keep operating, even if that leads to results that are inaccurate sometimes.

397 Safety critical applications tend to favor correctness to robustness. It is better to
398 return no result than to return a wrong result. The radiation machine is a good
399 example of this principle.

400 Consumer applications tend to favor robustness to correctness. Any result what-
401 soever is usually better than the software shutting down. The word processor I'm
402 using occasionally displays a fraction of a line of text at the bottom of the screen.
403 If it detects that condition do I want the word processor to shut down? No. I
404 know that the next time I hit *page up* or *page down*, the screen will refresh, and
405 the display will be back to normal.

406 **High-Level Design Implications of Error Process- 407 ing**

408 **KEY POINT**
409 With so many options, you need to be careful to handle invalid parameters in
410 consistent ways throughout the program. The way in which errors are handled
411 affects the software's ability to meet requirements related to correctness, robust-
412 ness, and other non-functional attributes. Deciding on a general approach to bad
413 parameters is an architectural or high-level design decision and should be ad-
dressed at one of those levels.

414 Once you decide on the approach, make sure you follow it consistently. If you
415 decide to have high-level code handle errors and low-level code merely report
416 errors, make sure the high level code actually handles the errors! Some lan-
417 guages including C++ might give you the option of ignoring the fact that a func-
418 tion is returning an error code. (In C++, you're not required to do anything with
419 a function's return value.) Don't ignore error information! Test the function re-
turn value. If you don't expect the function ever to produce an error, check it
anyway. The whole point of defensive programming is guarding against errors
you don't expect.

423 This guideline holds true for system functions as well as your own functions.
424 Unless you've set an architectural guideline of not checking system calls for er-
425 rors, check for error codes after each call. If you detect an error, include the error
426 number and the description of the error.

427 8.4 Exceptions

428 Exceptions are a specific means by which code can pass along errors or excep-
429 tional events to the code that called it. If code in one routine encounters an unex-
430 pected condition that it doesn't know how to handle, it throws an exception—
431 essentially throwing up its hands and yelling, "I don't know what to do about
432 this; I sure hope somebody else knows how to handle it!" Code that has no sense
433 of the context of an error can return control to other parts of the system that
434 might have a better ability to interpret the error and do something useful about it.

435 Exceptions can also be used to straighten out tangled logic within a single stretch
436 of code, such as the "Rewrite with *try-finally*" example in Section 17.3.

437 The basic structure of an exception in C++, Java, and Visual Basic is that a rou-
438 tine uses *throw* to throw an exception object. Code in some other routine up the
439 calling hierarchy will *catch* the exception within a *try-catch* block.

440 Popular Languages vary in how they implement exceptions. Table 8-1 summa-
441 rizes the major differences:

442 **Table 8-1. Popular Language Support for Exceptions**

Exception At- tribute	C++	Java	Visual Basic
<i>Try-catch</i> support	yes	yes	yes
<i>Try-catch-finally</i> support	no	yes	yes
What can be thrown	<i>Exception</i> object or object derived from <i>Exception</i> class; object pointer; object reference; data type like string or int	<i>Exception</i> object or object derived from <i>Exception</i> class	<i>Exception</i> object or object derived from <i>Exception</i> class

Exception Attribute	C++	Java	Visual Basic
Effect of uncaught exception	Invokes <code>std::unexpected()</code> , which by default invokes <code>std::terminate()</code> , which by default invokes <code>abort()</code>	Terminates thread of execution	Terminates program
Exceptions thrown must be defined in class interface	No	Yes	No
Exceptions caught must be defined in class interface	No	Yes	No

443 *Programs that use exceptions as part of their normal processing suffer from all the readability and maintainability problems of classic spaghetti code.*

449 —Andy Hunt and Dave Thomas

452

453
454
455
456
457

458
459
460
461
462
463

464
465
466

Exceptions have an attribute in common with inheritance: used judiciously, they can reduce complexity. Used imprudently, they can make code almost impossible to follow. This section contains suggestions for realizing the benefits of exceptions and avoiding the difficulties often associated with them.

Use exceptions to notify other parts of the program about errors that should not be ignored

The overriding benefit of exceptions is their ability to signal error conditions in such a way that they cannot be ignored (Meyers 1996). Other approaches to handling errors create the possibility that an error condition can propagate through a code base undetected. Exceptions eliminate that possibility.

Throw an exception only for conditions that are truly exceptional

Exceptions should be reserved for conditions that are truly exceptional, in other words, conditions that cannot be addressed by other coding practices. Exceptions are used in similar circumstances to assertions—for events that are not just infrequent, but that should *never* occur.

Exceptions represent a tradeoff between a powerful way to handle unexpected conditions on the one hand and increased complexity on the other. Exceptions weaken encapsulation by requiring the code that calls a routine to know which exceptions might be thrown inside the code that's called. That increases code complexity, which works against what Chapter 5 refers to as Software's Major Technical Imperative: Managing Complexity.

Don't use an exception to pass the buck

If an error condition can be handled locally, handle it locally. Don't throw an uncaught exception in a section of code if you can handle the error locally.

467 ***Avoid throwing exceptions in constructors and destructors unless you catch
468 them in the same place***

469 The rules for how exceptions are processed become very complicated very
470 quickly when exceptions are thrown in constructors and destructors. In C++, for
471 example, destructors aren't called unless an object is fully constructed, which
472 means if code within a constructor throws an exception, the destructor won't be
473 called, and that sets up a possible resource leak (Meyers 1996, Stroustrup 1997).
474 Similarly complicated rules apply to exceptions within destructors.

475 Language lawyers might say that remembering rule like these is "trivial," but
476 programmers who are mere mortals will have trouble remembering them. It's
477 better programming practice simply to avoid the extra complexity such code cre-
478 ates by not writing that kind of code in the first place.

479 ***Throw exceptions at the right level of abstraction***

480 **CROSS-REFERENCE** For
481 more on maintaining consist-
482 ent interface abstractions,
483 see "Good Abstraction" in
484 Section 6.2.
485

486 **CODING HORROR**

487 **Bad Java Example of a Class That Throws an Exception at an Inconsis-**

488 **tent Level of Abstraction**

```
489           class Employee {  
490           ...  
491           public TaxId getTaxId() E0FException {  
492           ...  
493           }  
494           ...  
495 }
```

496 The *getTaxId()* code passes the lower-level *io_disk_not_ready* exception back to
497 its caller. It doesn't take ownership of the exception itself; it exposes some de-
498 tails about how it is implemented by passing the lower-level exception to its
499 caller. This effectively couples the routine's client's code not the *Employee*
500 class's code, but to the code below the *Employee* class that throws the
501 *io_disk_not_ready exception*. Encapsulation is broken, and intellectual manage-
ability starts to decline.

502 Instead, the *getTaxId()* code should pass back an exception that's consistent with
503 the class interface of which it's a part, like this:

504
505
506
507
508 *Here is the declaration of the exception that contributes to a*
509 *consistent level of abstraction.*
510
511
512
513
514
515

516
517
518
519
520
521
522

523
524
525

CODING HORROR

526
527
528
529
530
531
532
533
534
535
536

537
538
539
540

Good Java Example of a Class That Throws an Exception at a Consistent Level of Abstraction

```
class Employee {  
    ...  
    public TaxId getTaxId() throws EmployeeDataNotAvailable {  
        ...  
    }  
    ...  
}
```

The exception-handling code inside *getTaxId()* will probably just map the *io_disk_not_ready* exception onto the *EmployeeDataNotAvailable* exception, which is fine because that's sufficient to preserve the interface abstraction.

Include all information that led to the exception in the exception message

Every exception occurs in specific circumstances that are detected at the time the code throws the exception. This information is invaluable to the person who reads the exception message. Be sure the message contains the information needed to understand why the exception was thrown. If the exception was thrown because of an array index error, be sure the exception message includes the upper and lower array limits and the value of the illegal index.

Avoid empty catch blocks

Sometimes it's tempting to pass off an exception that you don't know what to do with, like this:

Bad Java Example of Ignoring an Exception

```
try {  
    ...  
    // lots of code  
    ...  
} catch ( AnException exception ) {  
}
```

Such an approach says that either the code within the *try* block is wrong because it raises an exception for no reason, or the code within the *catch* block is wrong because it doesn't handle a valid exception. Determine which is the root cause of the problem, and then fix either the *try* block or the *catch* block.

Occasionally you'll find rare circumstances in which an exception at a lower level really doesn't represent an exception at the level of abstraction of the calling routine. If that's the case, at least document why an empty *catch* block is appropriate.

541
542
543
544
545
546
547548
549
550
551
552553
554

555

556 **FURTHER READING** For a
557 more detailed explanation of
558 this technique, see *Practical*
559 *Standards for Microsoft Vis-*
560 *ual Basic .NET* (Foxall
561 2003).

562

563

564

565

566

567

568

569

570

571

572

573

574

575

576

577

578

579

580

Know the exceptions your library code throws

If you're working in a language that doesn't require a routine or class to define the exceptions it throws, be sure you know what exceptions are thrown by any library code you use. Failing to catch an exception generated by library code will crash your program just as fast as failing to catch an exception you generated yourself. If the library code doesn't document the exceptions it throws, create prototyping code to exercise the libraries and flush out the exceptions.

Consider building a centralized exception reporter

One approach to ensuring consistency in exception handling is to use a centralized exception reporter. The centralized exception reporter provides a central repository for knowledge about what kinds of exceptions there are, how each exception should be handled, formatting of exception messages, and so on.

Here is an example of a simple exception handler that simply prints a diagnostic message:

Visual Basic Example of a Centralized Exception Reporter, Part 1

```
Sub ReportException( _
    ByVal className, _
    ByVal thisException As Exception _
)
    Dim message As String
    Dim caption As String

    message = "Exception: " & thisException.Message & ". " & ControlChars.CrLf & _
        "Class: " & className & ControlChars.CrLf & _
        "Routine: " & thisException.TargetSite.Name & ControlChars.CrLf
    caption = "Exception"
    MessageBox.Show( message, caption, MessageBoxButtons.OK, _
        MessageBoxIcon.Exclamation )

End Sub
```

You would use this generic exception handler with code like this:

Visual Basic Example of a Centralized Exception Reporter, Part 2

```
Try
    ...
Catch exceptionObject As Exception
    ReportException( CLASS_NAME, exceptionObject )
End Try
```

The code in this version of *ReportException()* is simple. In a real application you could make the code as simple or as elaborate as needed to meet your exception-handling needs.

581 If you do decide to build a centralized exception reporter, be sure to consider the
582 general issues involved in centralized error handling, which are discussed in
583 "Call an error processing routine/object" in Section 8.2.

584 ***Standardize your project's use of exceptions***

585 To keep exception handling as intellectually manageable as possible, you can
586 standardize your use of exceptions in several ways.

- 587 • If you're working in a language like C++ that allows you to throw a variety
588 of kinds of objects, data, and pointers, standardize on what specifically you
589 will throw. For compatibility with other languages, consider throwing only
590 objects derived from the *Exception* base class.
- 591 • Define the specific circumstances under which code is allowed to use *throw-*
592 *catch* syntax to perform error processing locally.
- 593 • Define the specific circumstances under which code is allowed to throw an
594 exception that won't be handled locally.
- 595 • Determine whether a centralized exception reporter will be used.
- 596 • Define whether exceptions are allowed in constructors and destructors.

597 ***Consider alternatives to exceptions***

598 **CROSS-REFERENCE** For
599 numerous alternative error
600 handling approaches, see
601 Section 8.2, "Error Handling
Techniques," earlier in this
chapter.

602 Some programmers use exceptions to handle errors just because their language
603 provides that particular error-handling mechanism. You should always consider
604 the full set of error-handling alternatives: handling the error locally, propagating
605 the error using an error code, logging debug information to a file, shutting down
606 the system, or using some other approach. Handling errors with exceptions just
607 because your language provides exception handling is a classic example of pro-
608 gramming *in* a language rather than programming *into* a language. (For details
on that distinction, see Section 4.3, "Your Location on the Technology Wave"
and Section 34.4, "Program Into Your Language, Not In It."

609 Finally, consider whether your program really needs to handle exceptions, pe-
610 riod. As Bjarne Stroustrup points out, sometimes the best response to a serious
611 run-time error is to release all acquired resources and abort. Let the user rerun
612 the program with proper input (Stroustrup 1997).

613 8.5 Barricade Your Program to Contain the 614 Damage Caused by Errors

615 Barricades are a damage-containment strategy. The reason is similar to that for
616 having isolated compartments in the hull of a ship. If the ship runs into an ice-
617 berg and pops open the hull, that compartment is shut off and the rest of the ship
618 isn't affected. They are also similar to firewalls in a building. A building's fire-
619 walls prevent fire from spreading from one part of a building to another part.
620 (Barricades used to be called "firewalls," but the term "firewall" now commonly
621 refers to port blocking.)

622 One way to barricade for defensive programming purposes is to designate certain
623 interfaces as boundaries to "safe" areas. Check data crossing the boundaries of a
624 safe area for validity and respond sensibly if the data isn't valid. Figure 8-2 illus-
625 trates this concept.

626 Error! Objects cannot be created from editing field codes.

627 **F08xx02**

628 **Figure 8-2**

629 *Defining some parts of the software that work with dirty data and some that work
630 with clean can be an effective way to relieve the majority of the code of the responsi-
631 bility for checking for bad data.*

632 This same approach can be used at the class level. The class's public methods
633 assume the data is unsafe, and they are responsible for checking the data and
634 sanitizing it. Once the data has been accepted by the class's public methods, the
635 class's private methods can assume the data is safe.

636 Another way of thinking about this approach is as an operating-room technique.
637 Data is sterilized before it's allowed to enter the operating room. Anything that's
638 in the operating room is assumed to be safe. The key design decision is deciding
639 what to put in the operating room, what to keep out, and where to put the
640 doors—which routines are considered to be inside the safety zone, which are
641 outside, and which sanitize the data. The easiest way to do this is usually by
642 sanitizing external data as it arrives, but data often needs to be sanitized at more
643 than one level, so multiple levels of sterilization are sometimes required.

644 **Convert input data to the proper type at input time**

645 Input typically arrives in the form of a string or number. Sometimes the value
646 will map onto a boolean type like "yes" or "no." Sometimes the value will map
647 onto an enumerated type like *Color_Red*, *Color_Green*, and *Color_Blue*. Carry-
648 ing data of questionable type for any length of time in a program increases com-
649 plexity and increases the chance that someone can crash your program by input-

ting a color like “Yes.” Convert input data to the proper form as soon as possible after it’s input.

652 Relationship between Barricades and Assertions

653 The use of barricades makes the distinction between assertions and error han-
654 dling clean cut. Routines that are outside the barricade should use error handling
655 because it isn't safe to make any assumptions about the data. Routines inside the
656 barricade should use assertions, because the data passed to them is supposed to
657 be sanitized before it's passed across the barricade. If one of the routines inside
658 the barricade detects bad data, that's an error in the program rather than an error
659 in the data.

660 The use of barricades also illustrates the value of deciding at the architectural
661 level how to handle errors. Deciding which code is inside and which is outside
662 the barricade is an architecture-level decision.

663 8.6 Debugging Aids

Another key aspect of defensive programming is the use of debugging aids, which can be a powerful ally in quickly detecting errors.

666 **Don't Automatically Apply Production Constraints**
667 **to the Development Version**

668 **FURTHER READING** For
669 more on using debug code to
670 support defensive program-
671 ming, see *Writing Solid Code*
(Maguire 1993).
672
673
674

A common programmer blind spot is the assumption that limitations of the production software apply to the development version. The production version has to run fast. The development version might be able to run slow. The production version has to be stingy with resources. The development version might be allowed to use resources extravagantly. The production version shouldn't expose dangerous operations to the user. The development version can have extra operations that you can use without a safety net.

675 One program I worked on made extensive use of a quadruply linked list. The
676 linked-list code was error prone, and the linked list tended to get corrupted. I
677 added a menu option to check the integrity of the linked list.

678 In debug mode, Microsoft Word contains code in the idle loop that checks the
679 integrity of the *Document* object every few seconds. This helps to detect data
680 corruption quickly, and makes for easier error diagnosis.

681 **KEY POINT**
682 Be willing to trade speed and resource usage during development in exchange
for built-in tools that can make development go more smoothly.

683

684

685

686

687

688

689 **CROSS-REFERENCE** For
690 more details on handling
691 unanticipated cases, see "Tips
692 for Using case Statements" in
Section 15.2.

693

694

695

696

697

698 *A dead program normally
does a lot less damage
than a crippled one.*

699
700 —Andy Hunt and Dave
701 Thomas

702

703

704

705

706

707

708

709

710

711

712

713

714

715

Introduce Debugging Aids Early

The earlier you introduce debugging aids, the more they'll help. Typically, you won't go to the effort of writing a debugging aid until after you've been bitten by a problem several times. If you write the aid after the first time, however, or use one from a previous project, it will help throughout the project.

Use Offensive Programming

Exceptional cases should be handled in a way that makes them obvious during development and recoverable when production code is running. Michael Howard and David LeBlanc refer to this approach as "offensive programming" (Howard and LeBlanc 2003).

Suppose you have a *case* statement that you expect to handle only five kinds of events. During development, the default case should be used to generate a warning that says "Hey! There's another case here! Fix the program!" During production, however, the default case should do something more graceful, like writing a message to an error-log file.

Here are some ways you can program offensively:

- Make sure *asserts* abort the program. Don't allow programmers to get into the habit of just hitting the ENTER key to bypass a known problem. Make the problem painful enough that it will be fixed.
- Completely fill any memory allocated so that you can detect memory allocation errors.
- Completely fill any files or streams allocated to flush out any file-format errors.
- Be sure the code in each *case* statement's *else* clause fails hard (aborts the program) or is otherwise impossible to overlook.
- Fill an object with junk data just before it's deleted

Sometimes the best defense is a good offense. Fail hard during development so that you can fail softer during production.

Plan to Remove Debugging Aids

If you're writing code for your own use, it might be fine to leave all the debugging code in the program. If you're writing code for commercial use, the performance penalty in size and speed can be prohibitive. Plan to avoid shuffling debugging code in and out of a program. Here are several ways to do that.

716 **CROSS-REFERENCE** For
717 details on version control, see
718 Section 28.2, "Configuration
719 Management."

720

721 **Use a built-in preprocessor**

722 If your programming environment has a preprocessor—as C++ does, for exam-
723 ple—you can include or exclude debug code at the flick of a compiler switch.
724 You can use the preprocessor directly or by writing a macro that works with pre-
725 processor definitions. Here's an example of writing code using the preprocessor
726 directly:

727 **C++ Example of Using the Preprocessor Directly to Control Debug 728 Code**

729 To include the debugging
730 code, use #DEFINE to define
731 the symbol DEBUG. To ex-
732 clude the debugging code,
733 don't define DEBUG.

734

735

736

737

738

739

740

741

742

743

744

```
#define DEBUG
...
#if defined( DEBUG )
// debugging code
...
#endif
```

This theme has several variations. Rather than just defining *DEBUG*, you can assign it a value and then test for the value rather than testing whether it's defined. That way you can differentiate between different levels of debug code. You might have some debug code that you want in your program all the time, so you surround that by a statement like *#if DEBUG > 0*. Other debug code might be for specific purposes only, so you can surround it by a statement like *#if DEBUG == POINTER_ERROR*. In other places, you might want to set debug levels, so you could have statements like *#if DEBUG > LEVEL_A*.

If you don't like having *#if defined()*s spread throughout your code, you can write a preprocessor macro to accomplish the same task. Here's an example:

747 **C++ Example of Using a Preprocessor Macro to Control Debug Code**

```
748 #define DEBUG
749
750 #if defined( DEBUG )
751 #define DebugCode( code_fragment ) { code_fragment }
752 #else
753 #define DebugCode( code_fragment )
754 #endif
755 ...
```

756
757
758 *This code is included or ex-*
759 *cluded depending on whether*
760 *DEBUG has been defined.*

761
762
763
764
765
766

767 **CROSS-REFERENCE** For
768 more information on pre-
769 processors and direction to
sources of information on
770 writing one of your own, see
771 "Macro preprocessors" in
772 "Macro Preprocessors" in
773 Section 30.3.

774

775 **CROSS-REFERENCE** For
776 details on stubs, see "Build-
777 ing Scaffolding to Test Indi-
vidual Routines" in "Building
778 Scaffolding to Test Individ-
ual Classes" in Section 22.5.

780
781
782
783
784

785

786
787
788
789
790
791
792
793 *This line calls the routine to*
794 *check the pointer.*

795

796

```
DebugCode(
    statement 1;
    statement 2;
    ...
    statement n;
);
```

As in the first example of using the preprocessor, this technique can be altered in a variety of ways that make it more sophisticated than completely including all debug code or completely excluding all of it.

Write your own preprocessor

If a language doesn't include a preprocessor, it's fairly easy to write one for including and excluding debug code. Establish a convention for designating debug code and write your precompiler to follow that convention. For example, in Java you could write a precompiler to respond to the keywords `//#BEGIN DEBUG` and `//#END DEBUG`. Write a script to call the preprocessor, and then compile the processed code. You'll save time in the long run, and you won't mistakenly compile the unprocessed code.

Use debugging stubs

In many instances, you can call a routine to do debugging checks. During development, the routine might perform several operations before control returns to the caller. For production code, you can replace the complicated routine with a stub routine that merely returns control immediately to the caller or performs only a couple of quick operations before returning control. This approach incurs only a small performance penalty, and it's a quicker solution than writing your own preprocessor. Keep both the development and production versions of the routines so that you can switch back and forth during future development and production.

You might start with a routine designed to check pointers that are passed to it:

C++ Example of a Routine that Uses a Debugging Stub

```
void DoSomething(
    SOME_TYPE *pointer;
    ...
) {

    // check parameters passed in
    CheckPointer( pointer );
    ...

}
```

797
798

During development, the *CheckPointer()* routine would perform full checking on the pointer. It would be slow but effective. It could look like this:

799

C++ Example of a Routine for Checking Pointers During Development800 *This routine checks any
801 pointer that's passed to it. It
802 can be used during develop-
803 ment to perform as many
804 checks as you can bear.*805
806
807
808
809

```
void CheckPointer( void *pointer ) {  
    // perform check 1--maybe check that it's not NULL  
    // perform check 2--maybe check that its dogtag is legitimate  
    // perform check 3--maybe check that what it points to isn't corrupted  
    ...  
    // perform check n--...  
}
```

When the code is ready for production, you might not want all the overhead associated with this pointer checking. You could swap out the routine above and swap in this routine:

810

C++ Example of a Routine for Checking Pointers During Production811 *This routine just returns im-
812 mediately to the caller.*813
814
815
816

```
void CheckPointer( void *pointer ) {  
    // no code; just return to caller  
}
```

This is not an exhaustive survey of all the ways you can plan to remove debugging aids, but it should be enough to give you an idea for some things that will work in your environment.

817
818

8.7 Determining How Much Defensive Programming to Leave in Production Code

819
820
821
822
823
824

One of the paradoxes of defensive programming is that during development, you'd like an error to be noticeable—you'd rather have it be obnoxious than risk overlooking it. But during production, you'd rather have the error be as unobtrusive as possible, to have the program recover or fail gracefully. Here are some guidelines for deciding which defensive programming tools to leave in your production code and which to leave out:

825
826
827
828
829
830
831
832
833

Leave in code that checks for important errors

Decide which areas of the program can afford to have undetected errors and which areas cannot. For example, if you were writing a spreadsheet program, you could afford to have undetected errors in the screen-update area of the program because the main penalty for an error is only a messy screen. You could not afford to have undetected errors in the calculation engine because the errors might result in subtly incorrect results in someone's spreadsheet. Most users would rather suffer a messy screen than incorrect tax calculations and an audit by the IRS.

834 ***Remove code that checks for trivial errors***
835 If an error has truly trivial consequences, remove code that checks for it. In the
836 previous example, you might remove the code that checks the spreadsheet screen
837 update. “Remove” doesn’t mean physically remove the code. It means use ver-
838 sion control, precompiler switches, or some other technique to compile the pro-
839 gram without that particular code. If space isn’t a problem, you could leave in
840 the error-checking code but have it log messages to an error-log file unobtru-
841 sively.

842 ***Remove code that results in hard crashes***
843 During development, when your program detects an error, you’d like the error to
844 be as noticeable as possible so that you can fix it. Often, the best way to accom-
845 plish such a goal is to have the program print a debugging message and crash
846 when it detects an error. This is useful even for minor errors.

847 During production, your users need a chance to save their work before the pro-
848 gram crashes and are probably willing to tolerate a few anomalies in exchange
849 for keeping the program going long enough for them to do that. Users don’t ap-
850 preciate anything that results in the loss of their work, regardless of how much it
851 helps debugging and ultimately improves the quality of the program. If your
852 program contains debugging code that could cause a loss of data, take it out of
853 the production version.

854 ***Leave in code that helps the program crash gracefully***
855 The opposite is also true. If your program contains debugging code that detects
856 potentially fatal errors, leave the code in that allows the program to crash grace-
857 fully. In the Mars Pathfinder, for example, engineers left some of the debug code
858 in by design. An error occurred after the Pathfinder had landed. By using the
859 debug aids that had been left in, engineers at JPL were able to diagnose the prob-
860 lem and upload revised code to the Pathfinder, and the Pathfinder completed its
861 mission perfectly (March 1999).

862 ***Log errors for your technical support personnel***
863 Consider leaving debugging aids in the production code but changing their be-
864 havior so that it’s appropriate for the production version. If you’ve loaded your
865 code with assertions that halt the program during development, you might con-
866 sidering changing the assertion routine to log messages to a file during produc-
867 tion rather than eliminating them altogether.

868 ***See that the error messages you leave in are friendly***
869 If you leave internal error messages in the program, verify that they’re in lan-
870 guage that’s friendly to the user. In one of my early programs, I got a call from a
871 user who reported that she’d gotten a message that read “You’ve got a bad
872 pointer allocation, Dog Breath!” Fortunately for me, she had a sense of humor. A

873
874

common and effective approach is to notify the user of an “internal error” and list an email address or phone number the user can use to report it.

875
876

8.8 Being Defensive About Defensive Programming

877 *Too much of anything is
878 bad, but too much whis-
879 key is just enough.*
880 —Mark Twain

881
882
883
884

Too much defensive programming creates problems of its own. If you check data passed as parameters in every conceivable way in every conceivable place, your program will be fat and slow. What’s worse, the additional code needed for defensive programming adds complexity to the software. Code installed for defensive programming is not immune to defects, and you’re just as likely to find a defect in defensive-programming code as in any other code—more likely, if you write the code casually. Think about where you need to be defensive, and set your defensive-programming priorities accordingly.

885 CC2E.COM/0868

886

887

888

889

890

891

892

893

894

895

896

897

898

899

900

901

902

903

904

905

906

CHECKLIST: Defensive Programming

General

- Does the routine protect itself from bad input data?
- Have you used assertions to document assumptions, including preconditions and postconditions?
- Have assertions been used only to document conditions that should never occur?
- Does the architecture or high-level design specify a specific set of error handling techniques?
- Does the architecture or high-level design specify whether error handling should favor robustness or correctness?
- Have barricades been created to contain the damaging effect of errors and reduce the amount of code that has to be concerned about error processing?
- Have debugging aids been used in the code?
- Has information hiding been used to contain the effects of changes so that they won’t affect code outside the routine or class that’s changed?
- Have debugging aids been installed in such a way that they can be activated or deactivated without a great deal of fuss?
- Is the amount of defensive programming code appropriate—neither too much nor too little?
- Have you used offensive programming techniques to make errors difficult to overlook during development?

Exceptions

- Has your project defined a standardized approach to exception handling?
- Have you considered alternatives to using an exception?
- Is the error handled locally rather than throwing a non-local exception if possible?
- Does the code avoid throwing exceptions in constructors and destructors?
- Are all exceptions at the appropriate levels of abstraction for the routines that throw them?
- Does each exception include all relevant exception background information?
- Is the code free of empty *catch* blocks? (Or if an empty *catch* block truly is appropriate, is it documented?)

Security Issues

- Does the code that checks for bad input data check for attempted buffer overflows, SQL injection, html injection, integer overflows, and other malicious inputs?
 - Are all error-return codes checked?
 - Are all exceptions caught?
 - Do error messages avoid providing information that would help an attacker break into the system?
-

CC2E.COM/0875

Additional Resources

Howard, Michael, and David LeBlanc. *Writing Secure Code*, 2d Ed., Redmond, WA: Microsoft Press, 2003. Howard and LeBlanc cover the security implications of trusting input. The book is eye opening in that it illustrates just how many ways a program can be breached—some of which have to do with construction practices and many of which don't. The book spans a full range of requirements, design, code, and test issues.

Assertions

Maguire, Steve. *Writing Solid Code*. Redmond, WA: Microsoft Press, 1993. Chapter 2 contains an excellent discussion on the use of assertions, including several interesting examples of assertions in well-known Microsoft products

Stroustrup, Bjarne. *The C++ Programming Language*, 3d Ed., Reading, Mass.: Addison Wesley, 1997. Section 24.3.7.2 describes several variations on the

940 theme of implementing assertions in C++, including the relationship between
941 assertions and preconditions and postconditions.

942 Meyer, Bertrand. *Object-Oriented Software Construction, 2d Ed.* New York:
943 Prentice Hall PTR, 1997. This book contains the definitive discussion of precon-
944 ditions and postconditions.

945 Exceptions

946 Meyer, Bertrand. *Object-Oriented Software Construction, 2d Ed.* New York:
947 Prentice Hall PTR, 1997. Chapter 12 contains a detailed discussion of exception
948 handling.

949 Stroustrup, Bjarne. *The C++ Programming Language, 3d Ed.*, Reading, Mass.:
950 Addison Wesley, 1997. Chapter 14 contains a detailed discussion of exception
951 handling in C++. Section 14.11 contains an excellent summary of 21 tips for
952 handling C++ exceptions.

953 Meyers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs*
954 and *Designs*, Reading, Mass.: Addison Wesley, 1996. Items 9-15 describe nu-
955 merous nuances of exception handling in C++.

956 Arnold, Ken, James Gosling, and David Holmes. *The Java Programming Lan-*
957 *guage, 3d Ed.*, Boston, Mass.: Addison Wesley, 2000. Chapter 8 contains a dis-
958 cussion of exception handling in Java.

959 Bloch, Joshua. *Effective Java Programming Language Guide*, Boston, Mass.:
960 Addison Wesley, 2001. Items 39-47 describe nuances of exception handling in
961 Java.

962 Foxall, James. *Practical Standards for Microsoft Visual Basic .NET*, Redmond,
963 WA: Microsoft Press, 2003. Chapter 10 describes exception handling in Visual
964 Basic.

965 Key Points

- 966 • Production code should handle errors in a more sophisticated way than “gar-
967 bage in, garbage out.”
- 968 • Defensive-programming techniques make errors easier to find, easier to fix,
969 and less damaging to production code.
- 970 • Assertions can help detect errors early, especially in large systems, high-
971 reliability systems, and fast-changing code bases.

- 972 ● The decision about how to handle bad inputs is a key error-handling deci-
- 973 sion, and a key high-level design decision.
- 974 ● Exceptions provide a means of handling errors that operates in a different
- 975 dimension from the normal flow of the code. They are a valuable addition to
- 976 the programmer's toolkit when used with care, and should be weighed
- 977 against other error-processing techniques.
- 978 ● Constraints that apply to the production system do not necessarily apply to
- 979 the development version. You can use that to your advantage, adding code to
- 980 the development version that helps to flush out errors quickly.

9

The Pseudocode Programming Process

CC2E.COM/0936

Contents

- 9.1 Summary of Steps in Building Classes and Routines
- 9.2 Pseudocode for Pros
- 9.3 Constructing Routines Using the PPP
- 9.4 Alternatives to the PPP

Related Topics

Creating high-quality classes: Chapter 6

Characteristics of high-quality routines: Chapter 7

High-level design: Chapter 5

Commenting style: Chapter 32

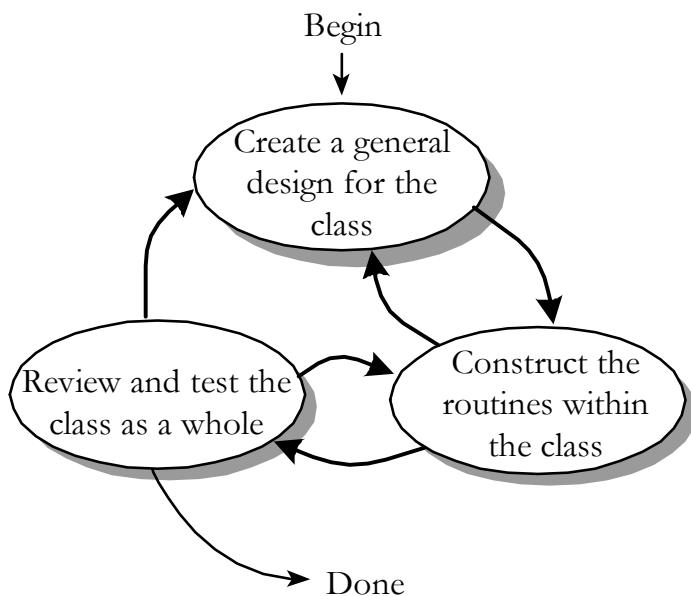
ALTHOUGH YOU COULD VIEW THIS WHOLE BOOK as an extended description of the programming process for creating classes and routines, this chapter puts the steps in context. This chapter focuses on programming in the small—on the specific steps for building an individual class and its routines that are critical on projects of all sizes. The chapter also describes the Pseudocode Programming Process (PPP), which reduces the work required during design and documentation and improves the quality of both.

If you're an expert programmer, you might just skim this chapter. But look at the summary of steps and review the tips for constructing routines using the Pseudocode Programming Process in Section 9.3. Few programmers exploit the full power of the process, and it offers many benefits.

The PPP is not the only procedure for creating classes and routines. Section 9.4 at the end of this chapter describes the most popular alternatives including test-first development and design by contract.

28 9.1 Summary of Steps in Building Classes 29 and Routines

30 Class construction can be approached from numerous directions, but usually it's
31 an iterative process of creating a general design for the class, enumerating
32 specific routines within the class, constructing specific routines, and checking
33 class construction as a whole. As Figure 9-1 suggests, class creation can be a
34 messy process for all the reasons that design is a messy process (which are
35 described in 5.1).



F09xx01

Figure 9-1

Details of class construction vary, but the activities generally occur in the order shown here.

41 Steps in Creating a Class

42 The key steps in constructing a class are:

43 **Create a general design for the class**

44 Class design includes numerous specific issues. Define the class's specific
45 responsibilities. Define what "secrets" the class will hide. Define exactly what
46 abstraction the class interface will capture. Determine whether the class will be
47 derived from another class, and whether other classes will be allowed to derive
48 from it. Identify the class's key public methods. Identify and design any non-
49 trivial data members used by the class. Iterate through these topics as many times

50 as needed to create a straightforward design for the routine. These considerations
51 and many others are discussed in more detail in Chapter 6, "Working Classes."

52 ***Construct each routine within the class***

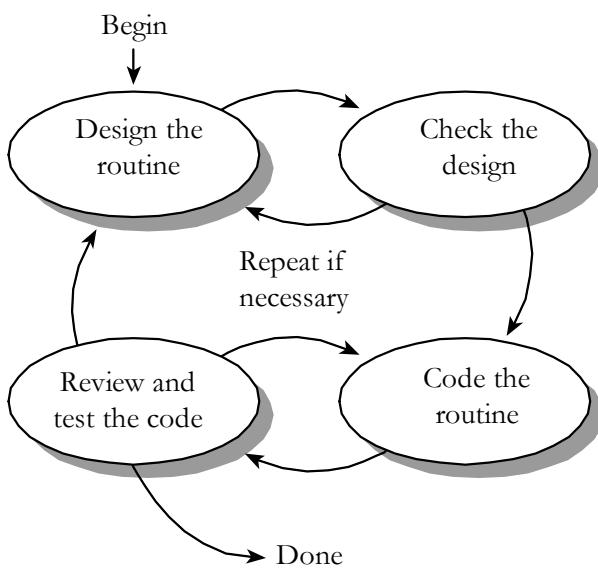
53 Once you've identified the class's major routines in the first step, you must
54 construct each specific routine. Construction of each routine typically unearths
55 the need for additional routines, both minor and major, and issues arising from
56 creating those additional routines often ripple back to the overall class design.

57 ***Review and test the class as a whole***

58 Normally, each routine is tested as it's created. After the class as a whole
59 becomes operational, the class as a whole should be reviewed and tested for any
60 issues that can't be tested at the individual-routine level.

61 **Steps in Building a Routine**

62 Many of a class's routines will be simple and straightforward to implement—
63 accessor routines, pass-throughs to other object's routines, and the like.
64 Implementation of other routines will be more complicated, and creation of those
65 routines benefits from a systematic approach. The major activities involved in
66 creating a routine—designing the routine, checking the design, coding the
67 routine, and checking the code—are typically performed in the order shown in
68 Figure 9-2.



69 **F09xx02**

70 **Figure 9-2**

71 *These are the major activities that go into constructing a routine. They're usually*
72 *performed in the order shown.*

74 Experts have developed numerous approaches to creating routines, and my
75 favorite approach is the Pseudocode Programming Process. That's described in
76 the next section.

77 9.2 Pseudocode for Pros

78 The term "pseudocode" refers to an informal, English-like notation for
79 describing how an algorithm, a routine, a class, or a program will work. The
80 Pseudocode Programming Process (PPP) defines a specific approach to using
81 pseudocode to streamline the creation of code within routines.

82 Because pseudocode resembles English, it's natural to assume that any English-
83 like description that collects your thoughts will have roughly the same effect as
84 any other. In practice, you'll find that some styles of pseudocode are more useful
85 than others. Here are guidelines for using pseudocode effectively:

- 86 • Use English-like statements that precisely describe specific operations.
- 87 • Avoid syntactic elements from the target programming language.
88 Pseudocode allows you to design at a slightly higher level than the code
89 itself. When you use programming-language constructs, you sink to a lower
90 level, eliminating the main benefit of design at a higher level, and you saddle
91 yourself with unnecessary syntactic restrictions.
- 92 • Write pseudocode at the level of intent. Describe the meaning of the
93 approach rather than how the approach will be implemented in the target
94 language.
- 95 • Write pseudocode at a low enough level that generating code from it will be
96 nearly automatic. If the pseudocode is at too high a level, it can gloss over
97 problematic details in the code. Refine the pseudocode in more and more
98 detail until it seems as if it would be easier to simply write the code.

99 Once the pseudocode is written, you build the code around it and the pseudocode
100 turns into programming-language comments. This eliminates most commenting
101 effort. If the pseudocode follows the guidelines, the comments will be complete
102 and meaningful.

103 Here's an example of a design in pseudocode that violates virtually all the
104 principles just described:

105 **CODING HORROR** Example of Bad Pseudocode

```
106 increment resource number by 1
107 allocate a dlg struct using malloc
108 if malloc() returns NULL then return 1
```

```
109 invoke OSsrc_init to initialize a resource for the operating system  
110 *hRsrcPtr = resource number  
111 return 0
```

What is the intent of this block of pseudocode? Because it's poorly written, it's hard to tell. This so-called pseudocode is bad because it includes coding details such as `*hRsrcPtr` in specific C-language pointer notation, and `malloc()`, a specific C-language function. This pseudocode block focuses on how the code will be written rather than on the meaning of the design. It gets into coding details—whether the routine returns a *1* or a *0*. If you think about this pseudocode from the standpoint of whether it will turn into good comments, you'll begin to understand that it isn't much help.

120 Here's a design for the same operation in a much-improved pseudocode:

121 Example of Good Pseudocode

```
122 Keep track of current number of resources in use  
123 If another resource is available  
124     Allocate a dialog box structure  
125     If a dialog box structure could be allocated  
126         Note that one more resource is in use  
127         Initialize the resource  
128         Store the resource number at the location provided by the caller  
129     Endif  
130 Endif  
131 Return TRUE if a new resource was created; else return FALSE
```

132 This pseudocode is better than the first because it's written entirely in English; it
133 doesn't use any syntactic elements of the target language. In the first example,
134 the pseudocode could have been implemented only in C. In the second example,
135 the pseudocode doesn't restrict the choice of languages. The second block of
136 pseudocode is also written at the level of intent. What does the second block of
137 pseudocode mean? It is probably easier for you to understand than the first
138 block.

139 Even though it's written in clear English, the second block of pseudocode is
140 precise and detailed enough that it can easily be used as a basis for
141 programming-language code. When the pseudocode statements are converted to
142 comments, they'll be a good explanation of the code's intent.

143 Here are the benefits you can expect from using this style of pseudocode:

- 144 • Pseudocode makes reviews easier. You can review detailed designs without
145 examining source code. Pseudocode makes low-level design reviews easier
146 and reduces the need to review the code itself.

147
148
149
150
151
152
153

154 **FURTHER READING** For
155 more information on the
156 advantages of making
157 changes at the least-value
158 stage, see Andy Grove's
159 *High Output Management*
(Grove 1983).

160
161
162

163
164
165
166

167
168
169
170
171
172

173 **KEY POINT**

174
175
176
177
178
179

- Pseudocode supports the idea of iterative refinement. You start with a high-level design, refine the design to pseudocode, and then refine the pseudocode to source code. This successive refinement in small steps allows you to check your design as you drive it to lower levels of detail. The result is that you catch high-level errors at the highest level, mid-level errors at the middle level, and low-level errors at the lowest level—before any of them becomes a problem or contaminates work at more detailed levels.
- Pseudocode makes changes easier. A few lines of pseudocode are easier to change than a page of code. Would you rather change a line on a blueprint or rip out a wall and nail in the two-by-fours somewhere else? The effects aren't as physically dramatic in software, but the principle of changing the product when it's most malleable is the same. One of the keys to the success of a project is to catch errors at the "least-value stage," the stage at which the least has been invested. Much less has been invested at the pseudocode stage than after full coding, testing, and debugging, so it makes economic sense to catch the errors early.
- Pseudocode minimizes commenting effort. In the typical coding scenario, you write the code and add comments afterward. In the PPP, the pseudocode statements become the comments, so it actually takes more work to remove the comments than to leave them in.
- Pseudocode is easier to maintain than other forms of design documentation. With other approaches, design is separated from the code, and when one changes, the two fall out of agreement. With the PPP, the pseudocode statements become comments in the code. As long as the inline comments are maintained, the pseudocode's documentation of the design will be accurate.

As a tool for detailed design, pseudocode is hard to beat. One survey found that programmers prefer pseudocode for the way it eases construction in a programming language, for its ability to help them detect insufficiently detailed designs, and for the ease of documentation and ease of modification it provides (Ramsey, Atwood, and Van Doren 1983). Pseudocode isn't the only tool for detailed design, but pseudocode and the PPP are useful tools to have in your programmer's toolbox. Try them. The next section shows you how.

180

9.3 Constructing Routines Using the PPP

181
182
183

This section describes the activities involved in constructing a routine, namely

- Design the routine
- Code the routine

184

185

186

187

188 **CROSS-REFERENCE** For
189 details on other aspects of
190 design, see Chapters 5
through 8.

191

192

193

194

195

196

197

198

199

200

201

202 **CROSS-REFERENCE** For
203 details on checking
204 prerequisites, see Chapter 3,
205 “Measure Twice, Cut Once:
206 Upstream Prerequisites” and
Chapter 4, “Key Construction
Decisions.”

207

208

209

210

211

212

213

- Check the code
- Clean up leftovers
- Repeat as needed

Design the Routine

Once you’ve identified a class’s routines, the first step in constructing any of the class’s more complicated routines is to design it. Suppose that you want to write a routine to output an error message depending on an error code, and suppose that you call the routine *ReportErrorMessage()*. Here’s an informal spec for *ReportErrorMessage()*:

ReportErrorMessage0 takes an error code as an input argument and outputs an error message corresponding to the code. It’s responsible for handling invalid codes. If the program is operating interactively, *ReportErrorMessage()* displays the message to the user. If it’s operating in command line mode, *ReportErrorMessage()* logs the message to a message file. After outputting the message, *ReportErrorMessage()* returns a status value indicating whether it succeeded or failed.

The rest of the chapter uses this routine as a running example. The rest of this section describes how to design the routine.

Check the prerequisites

Before doing any work on the routine itself, check to see that the job of the routine is well defined and fits cleanly into the overall design. Check to be sure that the routine is actually called for, at the very least indirectly, by the project’s requirements

Define the problem the routine will solve

State the problem the routine will solve in enough detail to allow creation of the routine. If the high level design is sufficiently detailed, the job might already be done. The high level design should at least indicate the following:

- The information the routine will hide
- Inputs to the routine
- Outputs from the routine

214 **CROSS-REFERENCE** For
215 details on preconditions and
216 post conditions, see “Use
217 assertions to document
218 preconditions and
219 postconditions” in Section
8.2.

220

221

222

223

224

225

226

227

228

229 **CROSS-REFERENCE** For
230 details on naming routines,
231 see Section 7.3, “Good
Routine Names.”

232

233

234

235

236

237

238

239

240 **FURTHER READING** For a
241 different approach to
242 construction that focuses on
243 writing test cases first, see
Test Driven Development
(Beck 2003).

244

245

246

247

248

- Preconditions that are guaranteed to be true before the routine is called (input values within certain ranges, streams initialized, files opened or closed, buffers filled or flushed, etc.)
- Post conditions that the routine guarantees will be true before it passes control back to the caller (output values within specified ranges, streams initialized, files opened or closed, buffers filled or flushed, etc.)

Here’s how these concerns are addressed in the *ReportErrorMessage()* example.

- The routine hides two facts: the error message text and the current processing method (interactive or command line).
- There are no preconditions guaranteed to the routine.
- The input to the routine is an error code.
- Two kinds of output are called for: The first is the error message; the second is the status that *ReportErrorMessage()* returns to the calling routine.
- The routine guarantees the status value will have a value of either *Success* or *Failure*.

Name the routine

Naming the routine might seem trivial, but good routine names are one sign of a superior program, and they’re not easy to come up with. In general, a routine should have a clear, unambiguous name. If you have trouble creating a good name, that usually indicates that the purpose of the routine isn’t clear. A vague, wishy-washy name is like a politician on the campaign trail. It sounds as if it’s saying something, but when you take a hard look, you can’t figure out what it means. If you can make the name clearer, do so. If the wishy-washy name results from a wishy-washy design, pay attention to the warning sign. Back up and improve the design.

In the example, *ReportErrorMessage()* is unambiguous. It is a good name.

Decide how to test the routine

As you’re writing the routine, think about how you can test it. This is useful for you when you do unit testing and for the tester who tests your routine independently.

In the example, the input is simple, so you might plan to test *ReportErrorMessage()* with all valid error codes and a variety of invalid codes.

Think about error handling

Think about all the things that could possibly go wrong in the routine. Think about bad input values, invalid values returned from other routines, and so on.

249 Routines can handle errors numerous ways, and you should choose consciously
250 how to handle errors. If the program's architecture defines the program's error
251 handling strategy, then you can simply plan to follow that strategy. In other
252 cases, you have to decide what approach will work best for the specific routine.

253 ***Think about efficiency***

254 Depending on your situation, you can address efficiency in one of two ways. In
255 the first situation, in the vast majority of systems, efficiency isn't critical. In such
256 a case, see that the routine's interface is well abstracted and its code is readable
257 so that you can improve it later if you need to. If you have good encapsulation,
258 you can replace a slow, resource-hogging high-level language implementation
259 with a better algorithm or a fast, lean, low-level language implementation, and
260 you won't affect any other routines.

261 **CROSS-REFERENCE** For
262 details on efficiency, see
263 Chapter 25, "Code-Tuning
264 Strategies" and Chapter 26,
265 "Code-Tuning Techniques."

In the second situation—in the minority of systems—performance is critical. The performance issue might be related to scarce database connections, limited memory, few available handles, ambitious timing constraints, or some other scarce resource. The architecture should indicate how many resources each routine (or class) is allowed to use and how fast it should perform its operations.

266 Design your routine so that it will meet its resource and speed goals. If either
267 resources or speed seems more critical, design so that you trade resources for
268 speed or vice versa. It's acceptable during initial construction of the routine to
269 tune it enough to meet its resource and speed budgets.

270 Aside from taking the approaches suggested for these two general situations, it's
271 usually a waste of effort to work on efficiency at the level of individual routines.
272 The big optimizations come from refining the high-level design, not the
273 individual routines. You generally use micro-optimizations only when the high-
274 level design turns out not to support the system's performance goals, and you
275 won't know that until the whole program is done. Don't waste time scraping for
276 incremental improvements until you know they're needed.

277 ***Research functionality available in the standard libraries***

278 The single biggest way to improve both the quality of your code and your
279 productivity is to reuse good code. If you find yourself grappling to design a
280 routine that seems overly complicated, ask whether some or all of the routine's
281 functionality might already be available in the library code of the environment or
282 tools you're using. Many algorithms have already been invented, tested,
283 discussed in the trade literature, reviewed, and improved. Rather than spending
284 your time inventing something when someone has already written a Ph.D.
285 dissertation on it, take a few minutes to look through the code that's already been
286 written, and make sure you're not doing more work than necessary.

287 ***Research the algorithms and data types***

288 If functionality isn't available in the available libraries, it might still be described
289 in an algorithms book. Before you launch into writing complicated code from
290 scratch, check an algorithms book to see what's already available. If you use a
291 predefined algorithm, be sure to adapt it correctly to your programming
292 language.

293 ***Write the pseudocode***

294 You might not have much in writing after you finish the preceding steps. The
295 main purpose of the steps is to establish a mental orientation that's useful when
296 you actually write the routine.

297 **CROSS-REFERENCE** This
298 discussion assumes that good
299 design techniques are used to
300 create the pseudocode
301 version of the routine. For
302 details on design, see Chapter
303 5, "High-Level Design in
304 Construction."

305
306
307
308

309

Example of a Header Comment for a Routine

310 This routine outputs an error message based on an error code
311 supplied by the calling routine. The way it outputs the message
312 depends on the current processing state, which it retrieves
313 on its own. It returns a value indicating success or failure.

314 After you've written the general comment, fill in high-level pseudocode for the
315 routine. Here's the pseudocode for the example:

316

Example of Pseudocode for a Routine

317 This routine outputs an error message based on an error code
318 supplied by the calling routine. The way it outputs the message
319 depends on the current processing state, which it retrieves
320 on its own. It returns a value indicating success or failure.

321
322 set the default status to "fail"
323 look up the message based on the error code
324
325 if the error code is valid

```
326     if doing interactive processing, display the error message  
327         interactively and declare success  
  
329     if doing command line processing, log the error message to the  
330         command line and declare success  
  
332     if the error code isn't valid, notify the user that an internal error  
333         has been detected  
  
335     return status information
```

Note that the pseudocode is written at a fairly high level. It certainly isn't written in a programming language. It expresses in precise English what the routine needs to do.

339 **CROSS-REFERENCE** For
340 details on effective use of
341 variables, see Chapters 10
through 13.

343
344
345

346 **CROSS-REFERENCE** For
347 details on review techniques,
348 see Chapter 21,
“Collaborative Construction.”
349

350
351
352
353
354

355 Make sure you have an easy and comfortable understanding of what the routine
356 does and how it does it. If you don't understand it conceptually, at the
357 pseudocode level, what chance do you have of understanding it at the
358 programming language level? And if you don't understand it, who else will?

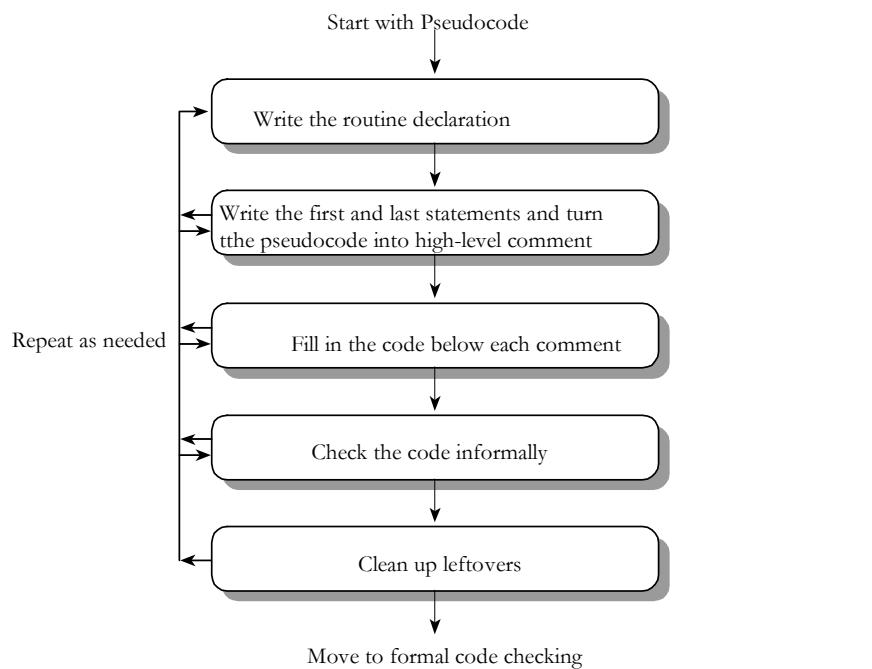
359
360 **CROSS-REFERENCE** For
361 more on iteration, see Section
362 34.8, “Iterate, Repeatedly,
Again and Again.”

363
364

365 and leave the original pseudocode as documentation. Some of the pseudocode
366 from your first attempt might be high-level enough that you need to decompose
367 it further. Be sure you do decompose it further. If you're not sure how to code
368 something, keep working with the pseudocode until you are sure. Keep refining
369 and decomposing the pseudocode until it seems like a waste of time to write it
370 instead of the actual code.

371 **Code the Routine**

372 Once you've designed the routine, construct it. You can perform construction
373 steps in a nearly standard order, but feel free to vary them as you need to. Figure
374 9-3 shows the steps in constructing a routine.



375 **F09xx03**

376 **Figure 9-3**

377 *You'll perform all of these steps as you design a routine but not necessarily in any
378 particular order.*

380 ***Write the routine declaration***

381 Write the routine interface statement—the function declaration in C++, method
382 declaration in Java, function or sub procedure declaration in Visual Basic, or
383 whatever your language calls for. Turn the original header comment into a
384 programming-language comment. Leave it in position above the pseudocode
385 you've already written. Here are the example routine's interface statement and
386 header in C++:

387

388 Here's the header comment
389 that's been turned into a C++-
390 style comment.

391

392

393

394 Here's the interface
395 statement.

396

397

398

399

400

401

402

403

404

405

406

407

408

409

410

411

412

413

414

415

416

417

C++ Example of a Routine Interface and Header Added to Pseudocode

```
/* This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure.

*/
Status ReportErrorMessage(
    ErrorCode errorToReport
)
set the default status to "fail"
look up the message based on the error code

if the error code is valid
    if doing interactive processing, display the error message
        interactively and declare success

    if doing command line processing, log the error message to the
        command line and declare success

if the error code isn't valid, notify the user that an
internal error has been detected

return status information
```

This is a good time to make notes about any interface assumptions. In this case, the interface variable *error* is straightforward and typed for its specific purpose, so it doesn't need to be documented.

Turn the pseudocode into high-level comments

Keep the ball rolling by writing the first and last statements—*{* and *}* in C++. Then turn the pseudocode into comments. Here's how it would look in the example:

C++ Example of Writing the First and Last Statements Around Pseudocode

```
/* This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure. */

Status ReportErrorMessage(
    ErrorCode errorToReport
) {
    // set the default status to "fail"
```

The pseudocode statements

429 from here down have been
430 turned into C++ comments.

```
// look up the message based on the error code
// if the error code is valid
    // if doing interactive processing, display the error message
    // interactively and declare success

    // if doing command line processing, log the error message to the
    // command line and declare success

    // if the error code isn't valid, notify the user that an
    // internal error has been detected

    // return status information
}
```

At this point, the character of the routine is evident. The design work is complete, and you can sense how the routine works even without seeing any code. You should feel that converting the pseudocode to programming-language code will be mechanical, natural, and easy. If you don't, continue designing in pseudocode until the design feels solid.

447
448 **CROSS-REFERENCE** This
449 is a case where the writing
450 metaphor works well—in the
small. For criticism of
451 applying the writing
452 metaphor in the large, see
453 “Software Penmanship:
Writing Code” in Section 2.3.

454
455

466

467

468

469

470

471

472

473

474

475

476

477

478

479

480

481

482

483

484

485

486

487

488

489

490

491

492

493

494

495

496

497

498

499

500

501

502

503

504

505

506

507

508

509

510

511

512

513

514

515

516

517

518

519

520

521

522

523

524

525

526

527

528

529

530

531

532

533

534

535

536

537

538

539

540

541

542

543

544

545

546

547

548

549

550

551

552

553

554

555

556

557

558

559

560

561

562

563

564

565

566

567

568

569

570

571

572

573

574

575

576

577

578

579

580

581

582

583

584

585

586

587

588

589

590

591

592

593

594

595

596

597

598

599

600

601

602

603

604

605

606

607

608

609

610

611

612

613

614

615

616

617

618

619

620

621

622

623

624

625

626

627

628

629

630

631

632

633

634

635

636

637

638

639

640

641

642

643

644

645

646

647

648

649

650

651

652

653

654

655

656

657

658

659

660

661

662

663

664

665

666

667

668

669

670

671

672

673

674

675

676

677

678

679

680

681

682

683

684

685

686

687

688

689

690

691

692

693

694

695

696

697

698

699

700

701

702

703

704

705

706

707

708

709

710

711

712

713

714

715

716

717

718

719

720

721

722

723

724

725

726

727

728

729

730

731

732

733

734

735

736

737

738

739

740

741

742

743

744

745

746

747

748

749

750

751

752

753

754

755

756

757

758

759

760

761

762

763

764

765

766

767

Here's the new variable
errorMessage.

```
470      // if the error code is valid
471          // if doing interactive processing, display the error message
472          // interactively and declare success
473
474          // if doing command line processing, log the error message to the
475          // command line and declare success
476
477          // if the error code isn't valid, notify the user that an
478          // internal error has been detected
479
480          // return status information
481      }
```

This is a start on the code. The variable *errorMessage* is used, so it needs to be declared. If you were commenting after the fact, two lines of comments for two lines of code would nearly always be overkill. In this approach, however, it's the semantic content of the comments that's important, not how many lines of code they comment. The comments are already there, and they explain the intent of the code, so leave them in (for now, at least).

The code below each of the remaining comments needs to be filled in. Here's the completed routine:

C++ Example of a Complete Routine Created with the Pseudocode Programming Process

```
490      /* This routine outputs an error message based on an error code
491         supplied by the calling routine. The way it outputs the message
492         depends on the current processing state, which it retrieves
493         on its own. It returns a value indicating success or failure.
494         */
495
496
497      Status ReportErrorMessage(
498          ErrorCode errorToReport
499      ) {
500          // set the default status to "fail"
501          Status errorMessageStatus = Status_Failure;
502
503          // look up the message based on the error code
504          Message errorMessage = LookupErrorMessage( errorToReport );
505
506          // if the error code is valid
507          if ( errorMessage.ValidCode() ) {
508              // determine the processing method
509              ProcessingMethod errorProcessingMethod = CurrentProcessingMethod();
510
511              // if doing interactive processing, display the error message
512          }
```

The code for each comment
has been filled in from here
down.

```
513 // interactively and declare success
514 if ( errorProcessingMethod == ProcessingMethod_Interactive ) {
515     DisplayInteractiveMessage( errorMessage.Text() );
516     errorMessageStatus = Status_Success;
517 }
518
519 // if doing command line processing, log the error message to the
520 // command line and declare success
521 else if ( errorProcessingMethod == ProcessingMethod_CommandLine ) {
522     This code is a good candidate
523     for being further decomposed
524         into a new routine:
525     DisplayCommandLineMessage
526             e().
527
528     This code and comment are
529         new and are the result of
530             fleshing out the if test.
531     This code and comment are
532         also new.
533
534
535
536 // if the error code isn't valid, notify the user that an
537 // internal error has been detected
538 else {
539     DisplayInteractiveMessage(
540         "Internal Error: Invalid error code in ReportErrorMessage()"
541     );
542
543
544 // return status information
545 return errorMessageStatus;
546 }
```

Each comment has given rise to one or more lines of code. Each block of code forms a complete thought based on the comment. The comments have been retained to provide a higher-level explanation of the code. All variables have been declared and defined close to the point they're first used. Each comment should normally expand to about 2 to 10 lines of code. (Because this example is just for purposes of illustration, the code expansion is on the low side of what you should usually experience in practice.)

Now look again at the spec on page 000 and the initial pseudocode on page 000. The original 5-sentence spec expanded to 15 lines of pseudocode (depending on how you count the lines), which in turn expanded into a page-long routine. Even though the spec was detailed, creation of the routine required substantial design

558 work in pseudocode and code. That low-level design is one reason why “coding”
559 is a nontrivial task and why the subject of this book is important.

560
561 **Check whether code should be further factored**
562 In some cases you’ll see an explosion of code below one of the initial lines of
563 pseudocode. In this case, you should consider taking one of two courses of
action:

564 **CROSS-REFERENCE** For
565 more on refactoring, see
566 Chapter 24, “Refactoring.”

567
568
569
570

- 571
- 572 • Factor the code below the comment into a new routine. If you find one line
573 of pseudocode expanding into more code than you expected, factor the
574 code into its own routine. Write the code to call the routine, including the
routine name. If you’ve used the PPP well, the name of the new routine
should drop out easily from the pseudocode. Once you’ve completed the
routine you were originally creating, you can dive into the new routine and
apply the PPP again to that routine.

- 575 • Apply the PPP recursively. Rather than writing a couple dozen lines of code
576 below one line of pseudocode, take the time to decompose the original line
577 of pseudocode into several more lines of pseudocode. Then continue filling
578 in the code below each of the new lines of pseudocode.

579

580 **KEY POINT**

581
582
583
584

585 **CROSS-REFERENCE** For
586 details on checking for errors
587 in architecture and
588 requirements, see Chapter 3,
589 “Measure Twice, Cut Once:
590 Upstream Prerequisites.”

591
592
593
594

595 After designing and implementing the routine, the third big step in constructing
it is checking to be sure that what you’ve constructed is correct. Any errors you
miss at this stage won’t be found until later testing. They’re more expensive to
find and correct then, so you should find all that you can at this stage.

596 A problem might not appear until the routine is fully coded for several reasons.
An error in the pseudocode might become more apparent in the detailed
implementation logic. A design that looks elegant in pseudocode might become
clumsy in the implementation language. Working with the detailed
implementation might disclose an error in the architecture, high level design, or
requirements. Finally, the code might have an old-fashioned, mongrel coding
error—nobody’s perfect! For all these reasons, review the code before you move
on.

597 **Mentally check the routine for errors**
598 The first formal check of a routine is mental. The clean-up and informal-
599 checking steps mentioned earlier are two kinds of mental checks. Another is
600 executing each path mentally. Mentally executing a routine is difficult, and that
601 difficulty is one reason to keep your routines small. Make sure that you check
602 nominal paths and endpoints and all exception conditions. Do this both by
603 yourself, which is called “desk checking,” and with one or more peers, which is

595
596

called a “peer review,” a “walkthrough,” or an “inspection,” depending on how you do it.

597 | **HARD DATA**

598
599
600
601
602
603
604
605
606
607
608
609

One of the biggest differences between hobbyists and professional programmers is the difference that grows out of moving from superstition into understanding. The word “superstition” in this context doesn’t refer to a program that gives you the creeps or generates extra errors when the moon is full. It means substituting feelings about the code for understanding. If you often find yourself suspecting that the compiler or the hardware made an error, you’re still in the realm of superstition. Only about 5 percent of all errors are hardware, compiler, or operating-system errors (Ostrand and Weyuker 1984). Programmers who have moved into the realm of understanding always suspect their own work first because they know that they cause 95 percent of errors. Understand the role of each line of code and why it’s needed. Nothing is ever right just because it seems to work. If you don’t know why it works, it probably doesn’t—you just don’t know it yet.

610 | **KEY POINT**

611

Bottom line: A working routine isn’t enough. If you don’t know why it works, study it, discuss it, and experiment with alternative designs until you do.

612
613
614
615
616

Compile the routine

After reviewing the routine, compile it. It might seem inefficient to wait this long to compile since the code was completed several pages ago. Admittedly, you might have saved some work by compiling the routine earlier and letting the computer check for undeclared variables, naming conflicts, and so on.

617
618
619
620
621
622
623

You’ll benefit in several ways, however, by not compiling until late in the process. The main reason is that when you compile new code, an internal stopwatch starts ticking. After the first compile, you step up the pressure: Get it right with Just One More Compile. The “Just One More Compile” syndrome leads to hasty, error-prone changes that take more time in the long run. Avoid the rush to completion by not compiling until you’ve convinced yourself that the routine is right.

624
625
626
627
628
629

The point of this book is to show how to rise above the cycle of hacking something together and running it to see if it works. Compiling before you’re sure your program works is often a symptom of the hacker mind-set. If you’re not caught in the hacking-and-compiling cycle, compile when you feel it’s appropriate to. But be conscious of the tug most people feel toward “hacking, compiling, and fixing” your way to a working program.

630

Here are some guidelines for getting the most out of compiling your routine:

- 631 • Set the compiler’s warning level to the pickiest level possible. You can catch
632 an amazing number of subtle errors simply by allowing the compiler to
633 detect them.
634 • Eliminate the causes of all compiler errors and warnings. Pay attention to
635 what the compiler tells you about your code. Large numbers of warnings
636 often indicates low-quality code, and you should try to understand each
637 warning you get. In practice, warnings you’ve seen again and again have one
638 of two possible effects: You ignore them and they camouflage other, more
639 important warnings, or they become annoying, like Chinese water torture.
640 It’s usually safer and less painful to rewrite the code to solve the underlying
641 problem and eliminate the warnings.

642 **Step through the code in the debugger**
643 Once the routine compiles, put it into the debugger and step through each line of
644 code. Make sure each line executes as you expect it to. You can find many errors
645 by following this simple practice.

646 **Test the code**
647 CROSS-REFERENCE For
648 details, see Chapter 22,
649 “Developer Testing.” Also
650 see “Building Scaffolding to
651 Test Individual Classes” in
652 Section 22.5.

Test the code using the test cases you planned or created while you were developing the routine. You might have to develop scaffolding to support your test cases—code that is used to support routines while they’re tested and isn’t included in the final product. Scaffolding can be a test-harness routine that calls your routine with test data, or it can be stubs called by your routine.

652 **Remove errors from the routine**
653 Once an error has been detected, it has to be removed. If the routine you’re
654 developing is buggy at this point, chances are good that it will stay buggy. If you
655 find that a routine is unusually buggy, start over. Don’t hack around it. Rewrite
656 it. Hacks usually indicate incomplete understanding and guarantee errors both
657 now and later. Creating an entirely new design for a buggy routine pays off. Few
658 things are more satisfying than rewriting a problematic routine and never finding
659 another error in it.

660 Clean Up Leftovers

661 When you’ve finished checking your code for problems, check it for the general
662 characteristics described throughout this book. You can take several cleanup
663 steps to make sure that the routine’s quality is up to your standards:

- 664 • Check the routine’s interface. Make sure that all input and output data is
665 accounted for and that all parameters are used. For more details, see Section
666 7.5, “How to Use Routine Parameters.”

- 667 • Check for general design quality. Make sure the routine does one thing and
668 does it well, that it's loosely coupled to other routines, and that it's designed
669 defensively. For details, see Chapter 7, "High-Quality Routines."
670 • Check the routine's data. Check for inaccurate variable names, unused data,
671 undeclared data, and so on. For details, see the chapters on using data,
672 Chapters 10 through 13.
673 • Check the routine's statements and logic. Check for off-by-one errors,
674 infinite loops, and improper nesting. For details, see the chapters on
675 statements, Chapters 14 through 19.
676 • Check the routine's layout. Make sure you've used white space to clarify the
677 logical structure of the routine, expressions, and parameter lists. For details,
678 see Chapter 31, "Layout and Style."
679 • Check the routine's documentation. Make sure the pseudocode that was
680 translated into comments is still accurate. Check for algorithm descriptions,
681 for documentation on interface assumptions and nonobvious dependencies,
682 for justification of unclear coding practices, and so on. For details, see
683 Chapter 32, "Self-Documenting Code."
684 • Remove redundant comments. Sometimes a pseudocode comment turns out
685 to be redundant with the code the comment describes, especially when the
686 PPP has been applied recursively, and the comment just precedes a call to a
687 well-named routine.

688 **Repeat Steps as Needed**

689 If the quality of the routine is poor, back up to the pseudocode. High-quality
690 programming is an iterative process, so don't hesitate to loop through the
691 construction activities again.

692 **9.4 Alternatives to the PPP**

693 For my money, the PPP is the best method for creating classes and routines. Here
694 are some of the alternative approaches recommended by other experts:

695 *Test-first development*

696 Test-first is a popular development style in which test cases are written prior to
697 writing any code. This approach is described in more detail in "Test First or Test
698 Last?" in Section 22.2. A good book on test first programming is Kent Beck's
699 *Test Driven Development* (Beck 2003).

700
701
702
703
704
705

706
707
708
709
710
711
712
713
714
715

CC2E.COM/0943

716

717 **CROSS-REFERENCE** The
718 point of this list is to check
whether you followed a good
719 set of steps to create a
720 routine. For a checklist that
721 focuses on the quality of the
routine itself, see the “High-
722 Quality Routines” checklist
723 in Chapter 5, page TBD.
724

725
726

727
728
729
730
731
732

733
734
735
736
737

Design by contract

Design by contract is a development approach in which each routine is considered to have preconditions and postconditions. This approach is described in “Use assertions to document preconditions and postconditions” in Section 8.2. The best source of information on design by contract is Bertrand Meyers’s *Object-Oriented Software Construction* (Meyer 1997).

Hacking?

Some programmers try to hack their way toward working code rather than using a systematic approach like the PPP. If you’ve ever find that you’ve coded yourself into a corner in a routine and have to start over, that’s an indication that the PPP might work better. If you find yourself losing your train of thought in the middle of coding a routine, that’s another indication that the PPP would be beneficial. Have you ever simply forgotten to write part of a class or part of routine? That hardly ever happens if you’re using the PPP. If you find yourself staring at the computer screen not knowing where to start, that’s a surefire sign that the PPP would make your programming life easier.

CHECKLIST: The Pseudocode Programming Process

- Have you checked that the prerequisites have been satisfied?
- Have you defined the problem that the class will solve?
- Is the high level design clear enough to give the class and each of its routines a good name?
- Have you thought about how to test the class and each of its routines?
- Have you thought about efficiency mainly in terms of stable interfaces and readable implementations, or in terms of meeting resource and speed budgets?
- Have you checked the standard libraries and other code libraries for applicable routines or components?
- Have you checked reference books for helpful algorithms?
- Have you designed each routine using detailed pseudocode?
- Have you mentally checked the pseudocode? Is it easy to understand?
- Have you paid attention to warnings that would send you back to design (use of global data, operations that seem better suited to another class or another routine, and so on)?
- Did you translate the pseudocode to code accurately?
- Did you apply the PPP recursively, breaking routines into smaller routines when needed?
- Did you document assumptions as you made them?
- Did you remove comments that turned out to be redundant?

- 738 Have you chosen the best of several iterations, rather than merely stopping
739 after your first iteration?
740 Do you thoroughly understand your code? Is it easy to understand?
741
-

742

Key Points

- 743
 - 744 • Constructing classes and constructing routines tends to be an iterative
745 process. Insights gained while constructing specific routines tend to ripple
746 back through the class's design.
 - 747 • Writing good pseudocode calls for using understandable English, avoiding
748 features specific to a single programming language, and writing at the level
749 of intent—describing what the design does rather than how it will do it.
 - 750 • The Pseudocode Programming Process is a useful tool for detailed design
751 and makes coding easy. Pseudocode translates directly into comments,
752 ensuring that the comments are accurate and useful.
 - 753 • Don't settle for the first design you think of. Iterate through multiple
754 approaches in pseudocode and pick the best approach before you begin
755 writing code.
 - 756 • Check your work at each step and encourage others to check it too. That
757 way, you'll catch mistakes at the least expensive level, when you've
 invested the least amount of effort.

10

2 3 **General Issues in Using Variables**

4 CC2E.COM/1085

5 **Contents**

- 6 10.1 Data Literacy
- 7 10.2 Making Variable Declarations Easy
- 8 10.3 Guidelines for Initializing Variables
- 9 10.4 Scope
- 10 10.5 Persistence
- 11 10.6 Binding Time
- 12 10.7 Relationship Between Data Types and Control Structures
- 13 10.8 Using Each Variable for Exactly One Purpose

14 **Related Topics**

- 15 Naming variables: Chapter 11
- 16 Fundamental data types: Chapter 12
- 17 Unusual data types: Chapter 13
- 18 Formatting data declarations: “Laying Out Data Declarations” in Section 31.5
- 19 Documenting variables: “Commenting Data Declarations” in Section 32.5

20 IT’S NORMAL AND DESIRABLE FOR construction to fill in small gaps in the
21 requirements and architecture. It would be inefficient to draw blueprints to such
22 a microscopic level that every detail was completely specified. This chapter
describes a nuts and bolts construction issue—ins and outs of using variables.

23 The information in this chapter should be particularly valuable to you if you’re
24 an experienced programmer. It’s easy to start using hazardous practices before
25 you’re fully aware of your alternatives and then to continue to use them out of
26 habit even after you’ve learned ways to avoid them. An experienced programmer
27 might find the discussions on binding time in Section 10.6 and on using each
28 variable for one purpose in Section 10.8 particularly interesting. If you’re not

29 sure whether you qualify as an “experienced programmer,” take the “Data
30 Literacy Test” in the next section, and find out.

31 Throughout this chapter I use the word “variable” to refer to objects as well as to
32 built-in data types like integers and arrays. The phrase “data type” generally
33 refers to built-in data types, while the word “data” refers to either objects or
34 built-in types.

35 10.1 Data Literacy

36 KEY POINT

37 The first step in creating effective data is knowing which kind of data to create.
38 A good repertoire of data types is a key part of a programmer’s toolkit. A tutorial
39 in data types is beyond the scope of this book, but take the “Data Literacy Test”
 below to determine how much more you might need to learn about them.

40 The Data Literacy Test

41 Put a *I* next to each term that looks familiar. If you think you know what a term
42 means but aren’t sure, give yourself a *0.5*. Add the points when you’re done, and
43 interpret your score according to the scoring table below.

<input type="checkbox"/> abstract data type	<input type="checkbox"/> literal
<input type="checkbox"/> array	<input type="checkbox"/> local variable
<input type="checkbox"/> bitmap	<input type="checkbox"/> lookup table
<input type="checkbox"/> boolean variable	<input type="checkbox"/> member data
<input type="checkbox"/> B-tree	<input type="checkbox"/> pointer
<input type="checkbox"/> character variable	<input type="checkbox"/> private
<input type="checkbox"/> container class	<input type="checkbox"/> retroactive synapse
<input type="checkbox"/> double precision	<input type="checkbox"/> referential integrity
<input type="checkbox"/> elongated stream	<input type="checkbox"/> stack
<input type="checkbox"/> enumerated type	<input type="checkbox"/> string
<input type="checkbox"/> floating point	<input type="checkbox"/> structured variable
<input type="checkbox"/> heap	<input type="checkbox"/> tree
<input type="checkbox"/> index	<input type="checkbox"/> typedef
<input type="checkbox"/> integer	<input type="checkbox"/> union
<input type="checkbox"/> linked list	<input type="checkbox"/> value chain

named constant variant
 Total Score

44

Here is how you can interpret the scores (loosely):

0–14	You are a beginning programmer, probably in your first year of computer science in school or teaching yourself your first programming language. You can learn a lot by reading one of the books listed below. Many of the descriptions of techniques in this part of the book are addressed to advanced programmers, and you'll get more out of them after you've read one of these books.
15–19	You are an intermediate programmer or an experienced programmer who has forgotten a lot. Although many of the concepts will be familiar to you, you too can benefit from reading one of the books listed below.
20–24	You are an expert programmer. You probably already have the books listed below on your shelf.
25–29	You know more about data types than I do. Consider writing your own computer book. (Send me a copy!)
30–32	You are a pompous fraud. The terms "elongated stream," "retroactive synapse," and "value chain" don't refer to data types—I made them up. Please read the intellectual-honesty section in Chapter 31!

45

Additional Resources on Data Types

46

These books are good sources of information about data types:

47

Cormen, H. Thomas, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. New York: McGraw Hill. 1990.

48

49

Sedgewick, Robert. *Algorithms in C++, Part 5, 3d ed.* Boston, Mass.: Addison-Wesley, 2002.

50

51

Sedgewick, Robert. *Algorithms in C++, Parts 1-4, 3d ed.* Boston, Mass.: Addison-Wesley, 1998.

52

53

54 **CROSS-REFERENCE** For
55 details on layout of variable
56 declarations, see “Laying Out
57 Data Declarations” in Section
58 31.5. For details on
59 documenting them, see
“Commenting Data
32.5.

60

61

62

63

64

65
66
67
68

69 **KEY POINT**

70
71
72
73
74
75
76

77

78

79

80

81
82
83

84 **CROSS-REFERENCE** For
85 details on the standardization
86 of abbreviations, see
“General Abbreviation
Guidelines” in Section 11.6.

10.2 Making Variable Declarations Easy

This section describes what you can do to streamline the task of declaring variables. To be sure, this is a small task, and you may think it’s too small to deserve its own section in this book. Nevertheless, you spend a lot of time creating variables, and developing the right habits can save time and frustration over the life of a project.

Implicit Declarations

Some languages have implicit variable declarations. For example, if you use a variable in Visual Basic without declaring it, the compiler declares it for you automatically (depending on your compiler settings).

Implicit declaration is one of the most hazardous features available in any language.

If you program in Visual Basic, you know how frustrating it is to try to figure out why *acctNo* doesn’t have the right value and then notice that *acctNum* is the variable that’s reinitialized to 0. This kind of mistake is an easy one to make if your language doesn’t require you to declare variables.

If you’re programming in a language that requires you to declare variables, you have to make two mistakes before your program will bite you. First you have to put both *acctNum* and *acctNo* into the body of the routine. Then you have to declare both variables in the routine. This is a harder mistake to make and virtually eliminates the synonymous-variables problem. Languages that require you explicitly to declare data force you to use data more carefully, which is one of their primary advantages. What do you do if you program in a language with implicit declarations? Here are some suggestions:

Turn off implicit declarations

Some compilers allow you to disable implicit declarations. For example, in Visual Basic you would use an *Option Explicit* statement, which forces you to declare all variables before you use them.

Declare all variables

As you type in a new variable, declare it, even though the compiler doesn’t require you to. This won’t catch all the errors, but it will catch some of them.

Use naming conventions

Establish a naming convention for common suffixes such as *Num* and *No* so that you don’t use two variables when you mean to use one.

87
88
89
90
91**Check variable names**

Use the cross-reference list generated by your compiler or another utility program. Many compilers list all the variables in a routine, allowing you to spot both *acctNum* and *acctNo*. They also point out variables that you've declared and not used.

92

KEY POINT93
94
95

Improper data initialization is one of the most fertile sources of error in computer programming. Developing effective techniques for avoiding initialization problems can save a lot of debugging time.

96
97
98

The problems with improper initialization stem from a variable's containing an initial value that you do not expect it to contain. This can happen for any of the several reasons described on the next page.

99 **CROSS-REFERENCE** For
100 a testing approach based on
data initialization and use
101 patterns, see "Data-Flow
102 Testing" in Section 22.3.

- The variable has never been assigned a value. Its value is whatever bits happened to be in its area of memory when the program started.
- The value in the variable is outdated. The variable was assigned a value at some point, but the value is no longer valid.
- Part of the variable has been assigned a value and part has not.

This last theme has several variations. You can initialize some of the members of an object but not all of them. You can forget to allocate memory and then initialize the "variable" the uninitialized pointer points to. This means that you are really selecting a random portion of computer memory and assigning it some value. It might be memory that contains data. It might be memory that contains code. It might be the operating system. The symptom of the pointer problem can manifest itself in completely surprising ways that are different each time—that's what makes debugging pointer errors harder than debugging other errors.

112 Here are guidelines for avoiding initialization problems.

Initialize each variable as it's declared

113 Initializing variables as they're declared is an inexpensive form of defensive
114 programming. It's a good insurance policy against initialization errors. The
115 example below ensures that *studentName* will be reinitialized each time you call
116 the routine that contains it.
117

118 **CROSS-REFERENCE** Code samples in this book are
119 formatted using a coding
CROSS-REFERENCE Che
120 cking input parameters is a
121 form of defensive
122 programming. For details on
123 defensive programming, see
124 Chapter 8, “Defensive
Programming.”

125 **CODING HORROR**

126 11.4.
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147

148
149
150
151
152
153
154
155 *total is declared and initialized*
156 *close to where it's used.*
157
158
159

C++ Example of Initialization at Declaration Time

```
char studentName [ NAME_LENGTH + 1 ] = {'\0'}; // full name of student
```

Initialize each variable close to where it's first used

Some languages, including Visual Basic, don't support initializing variables as they're declared. That can lead to coding styles like the one below, in which declarations are grouped together, and then initializations are grouped together—all far from the first actual use of the variables.

Visual Basic Example of Bad Initialization

```
' declare all variables
Dim accountIndex As Integer
Dim total As Double
Dim done As Boolean

' initialize all variables
accountIndex = 0
total = 0.0
done = False
...
.

' code using accountIndex
...
.

' code using total
...
.

' code using done
While Not done
    ...

```

A better practice is to initialize variables as close as possible to where they're first used:

Visual Basic Example of Good Initialization

```
Dim accountIndex As Integer
accountIndex = 0
' code using accountIndex
...

Dim total As Double
total = 0.0
' code using total
...

Dim done As Boolean
```

160 *done is also declared and
161 initialized close to where it's
162 used.*

163
164 **CROSS-REFERENCE** For
165 more details on keeping
166 related actions together, see
167 Section 14.2, "Statements
168 Whose Order Doesn't
169 Matter."

170
171
172
173
174

175 This is an example of the Principle of Proximity: Keep related actions together.
176 The same principle applies to keeping comments close to the code they describe,
177 to keeping loop setup code close to the loop, to grouping statements in straight-
178 line code, and to many other areas.

179
180
181
182
183

184 **Java Example of Good Initialization**

185 int accountIndex = 0;
186 // code using accountIndex
187 ...
188
189 *total is initialized close to
190 where it's used.*

191
192
193 *done is also initialized close to
194 where it's used.*

195
196

```
done = False
' code using done
While Not done
    ...
```

The second example is superior to the first for several reasons. By the time execution of the first example gets to the code that uses *done*, *done* could have been modified. If that's not the case when you first write the program, later modifications might make it so. Another problem with the first approach is that throwing all the initializations together creates the impression that all the variables are used throughout the whole routine—when in fact *done* is used only at the end. Finally, as the program is modified (as it will be, if only by debugging), loops might be built around the code that uses *done*, and *done* will need to be reinitialized. The code in the second example will require little modification in such a case. The code in the first example is more prone to producing an annoying initialization error.

This is an example of the Principle of Proximity: Keep related actions together. The same principle applies to keeping comments close to the code they describe, to keeping loop setup code close to the loop, to grouping statements in straight-line code, and to many other areas.

Ideally, declare and define each variable close to where it's used

A declaration establishes a variable's type. A definition assigns the variable a specific value. In languages that support it, such as C++ and Java, variables should be declared and defined close to where they are first used. Ideally, each variable should be defined at the same time it's declared, as shown below.

Java Example of Good Initialization

```
int accountIndex = 0;
// code using accountIndex
...
double total = 0.0;
// code using total
...
boolean done = false;
// code using done
while ( ! done ) {
    ...
```

197 **CROSS-REFERENCE** For
198 more details on keeping
199 related actions together, see
200 Section 14.2, “Statements
Whose Order Doesn’t
Matter.”

201
202
203
204

205
206
207
208
209
210

211
212
213
214
215
216
217
218

219
220
221
222
223
224
225

226
227

228 **CROSS-REFERENCE** For
229 more on checking input
230 parameters, see Section 8.1,
“Protecting Your Program
231 From Invalid Inputs” and the
rest of Chapter 8, “Defensive
232 Programming.”

233

Pay special attention to counters and accumulators

The variables *i*, *j*, *k*, *sum*, and *total* are often counters or accumulators. A common error is forgetting to reset a counter or an accumulator before the next time it’s used.

Initialize a class’s member data in its constructor

Just as a routine’s variables should be initialized within each routine, a class’s data should be initialized within its constructor. If memory is allocated in the constructor, it should be freed in the destructor.

Check the need for reinitialization

Ask yourself whether the variable will ever need to be reinitialized—either because a loop in the routine uses the variable many times or because the variable retains its value between calls to the routine and needs to be reset between calls. If it needs to be reinitialized, make sure that the initialization statement is inside the part of the code that’s repeated.

Initialize named constants once; initialize variables with executable code

If you’re using variables to emulate named constants, it’s OK to write code that initializes them once, at the beginning of the program. To do this, initialize them in a *Startup()* routine. Initialize true variables in executable code close to where they’re used. One of the most common program modifications is to change a routine that was originally called once so that you call it multiple times. Variables that are initialized in a program-level *Startup()* routine aren’t reinitialized the second time through the routine.

Use the compiler setting that automatically initializes all variables

If your compiler supports such an option, having the compiler set to automatically initialize all variables is an easy variation on the theme of relying on your compiler. Relying on specific compiler settings, however, can cause problems when you move the code to another machine and another compiler. Make sure you document your use of the compiler setting; assumptions that rely on specific compiler settings are hard to uncover otherwise.

Take advantage of your compiler’s warning messages

Many compilers warn you that you’re using an uninitialized variable.

Check input parameters for validity

Another valuable form of initialization is checking input parameters for validity. Before you assign input values to anything, make sure the values are reasonable.

Use a memory-access checker to check for bad pointers

In some operating systems, the operating-system code checks for invalid pointer references. In others, you’re on your own. You don’t have to stay on your own,

234 however, because you can buy memory-access checkers that check your
235 program's pointer operations.

236 ***Initialize working memory at the beginning of your program***
237 Initializing working memory to a known value helps to expose initialization
238 problems. You can take any of several approaches:

- 239
- 240
- 241
- 242
- 243
- 244
- 245
- 246
- 247
- 248
- 249
- 250
- 251
- 252
- 253
- 254
- 255
- 256
- 257
- You can use a preprogram memory filler to fill the memory with a predictable value. The value 0 is good for some purposes because it ensures that uninitialized pointers point to low memory, making it relatively easy to detect them when they're used. On the Intel processors, 0xCC is a good value to use because it's the machine code for a breakpoint interrupt; if you are running code in a debugger and try to execute your data rather than your code, you'll be awash in breakpoints. Another virtue of the value 0xCC is that it's easy to recognize in memory dumps—and it's rarely used for legitimate reasons. Alternatively, Brian Kernighan and Rob Pike suggest using the constant 0xDEADBEEF as memory filler that's easy to recognize in a debugger (1999).
 - If you're using a memory filler, you can change the value you use to fill the memory once in awhile. Shaking up the program sometimes uncovers problems that stay hidden if the environmental background never changes.
 - You can have your program initialize its working memory at startup time. Whereas the purpose of using a preprogram memory filler is to expose defects, the purpose of this technique is to hide them. By filling working memory with the same value every time, you guarantee that your program won't be affected by random variations in the startup memory.

258 10.4 Scope

259 “Scope” is a way of thinking about a variable’s celebrity status: how famous is
260 it? Scope, or visibility, refers to the extent to which your variables are known
261 and can be referenced throughout a program. A variable with limited or small
262 scope is known in only a small area of a program—a loop index used in only one
263 small loop, for instance. A variable with large scope is known in many places in
264 a program—a table of employee information that’s used throughout a program,
265 for instance.

266 Different languages handle scope in different ways. In some primitive languages,
267 all variables are global. You therefore don’t have any control over the scope of a
268 variable, and that can create a lot of problems. In C++ and similar languages, a
269 variable can be visible to a block (a section of code enclosed in curly brackets), a

270 routine, a class, or the whole program. In Java and C#, a variable can also be
271 visible to a package or namespace (a collection of classes).

272 The following sections provide guidelines that apply to scope.

273 Localize References to Variables

274 The code between references to a variable is a “window of vulnerability.” In the
275 window, new code might be added, inadvertently altering the variable, or
276 someone reading the code might forget the value the variable is supposed to
277 contain. It’s always a good idea to localize references to variables by keeping
278 them close together.

279 The idea of localizing references to a variable is pretty self-evident, but it’s an
280 idea that lends itself to formal measurement. One method of measuring how
281 close together the references to a variable are is to compute the “span” of a
282 variable. Here’s an example:

283 Java Example of Variable Span

```
284 a = 0;  
285 b = 0;  
286 c = 0;  
287 a = b + c;
```

288 In this case, two lines come between the first reference to *a* and the second, so *a*
289 has a span of two. One line comes between the two references to *b*, so *b* has a
290 span of one, and *c* has a span of zero. Here’s another example:

291 Java Example of Spans of One and Zero

```
292 a = 0;  
293 b = 0;  
294 c = 0;  
295 b = a + 1;  
296 b = b / c;
```

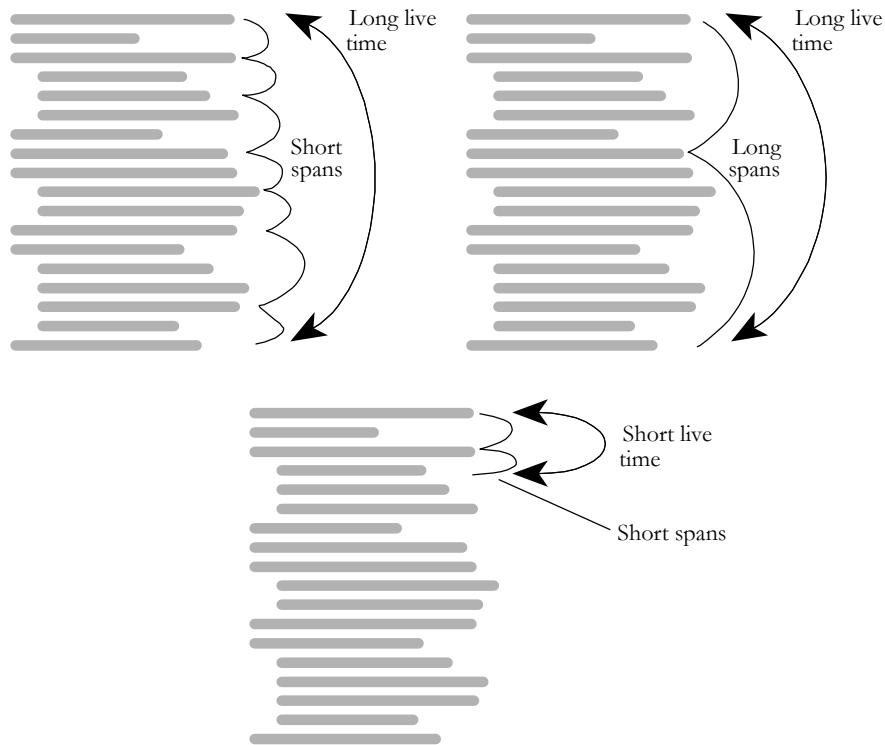
297 In this case, there is one line between the first reference to *b* and the second, for a
298 span of one. There are no lines between the second reference to *b* and the third,
299 for a span of zero.

300 The average span is computed by averaging the individual spans; in *b*’s case,
301 $(1+0)/2$ equals an average span of 0.5. When you keep references to variables
302 close together, you enable the person reading your code to focus on one section
303 at a time. If the references are far apart, you force the reader to jump around in
304 the program. Thus the main advantage of keeping references to variables
305 together is that it improves program readability.

Keep Variables Live for As Short a Time As Possible

A concept that's related to variable span is variable "live time," the total number of statements over which a variable is live. A variable's life begins at the first statement in which it's referenced; its life ends at the last statement in which it's referenced.

Unlike span, live time isn't affected by how many times the variable is used between the first and last times it's referenced. If the variable is first referenced on line 1 and last referenced on line 25, it has a live time of 25 statements. If those are the only two lines in which it's used, it has an average span of 23 statements. If the variable were used on every line from line 1 through line 25, it would have an average span of 0 statements, but it would still have a live time of 25 statements. Figure 10-1 illustrates both span and live time.



F10xx01

Figure 10-1

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

319

320

321

322

323

324

325 As with span, the goal with respect to live time is to keep the number low, to
326 keep a variable live for as short a time as possible. And as with span, the basic
327 advantage of maintaining a low number is that it reduces the window of
328 vulnerability. You reduce the chance of incorrectly or inadvertently altering a
329 variable between the places in which you intend to alter it.

330 A second advantage of keeping the live time short is that it gives you an accurate
331 picture of your code. If a variable is assigned a value in line 10 and not used
332 again until line 45, the very space between the two references implies that the
333 variable is used between lines 10 and 45. If the variable is assigned a value in
334 line 44 and used in line 45, no other uses of the variable are implied, and you can
335 concentrate on a smaller section of code when you're thinking about that
336 variable.

337 A short live time also reduces the chance of initialization errors. As you modify
338 a program, straight-line code tends to turn into loops and you tend to forget
339 initializations that were made far away from the loop. By keeping the
340 initialization code and the loop code closer together, you reduce the chance that
341 modifications will introduce initialization errors.

342 Finally, a short live time makes your code more readable. The fewer lines of
343 code a reader has to keep in mind at once, the easier your code is to understand.
344 Likewise, the shorter the live time, the less code you have to keep on your screen
345 when you want to see all the references to a variable during editing and
346 debugging.

347 Measuring the Live Time of a Variable

348 You can formalize the concept of live time by counting the number of lines
349 between the first and last references to a variable (including both the first and
350 last lines). Here's an example with live times that are too long:

351 Java Example of Variables with Excessively Long Live Times

```
352 // initialize all variables
353 recordIndex = 0;
354 total = 0;
355 done = false;
356 ...
357 while ( recordIndex < recordCount ) {
358     ...
359     Last reference to recordIndex
360     recordIndex = recordIndex + 1;
361     ...
362     while ( !done ) {
363         ...
```

364 *Last reference to total*
 365 *Last reference to done*
 366

```
69     if ( total > projectedTotal ) {  
70         done = true;
```

Here are the live times for the variables in this example:

<i>recordIndex</i>	(line 28 – line 2 + 1) = 27
<i>total</i>	(line 69 – line 3 + 1) = 67
<i>done</i>	(line 70 – line 4 + 1) = 67

Average Live Time	(27 + 67 + 67) / 3 ≈ 54
-------------------	---------------------------

The example has been rewritten below so that the variable references are closer together:

Java Example of Variables with Good, Short Live Times

369
 370
 371 *Initialization of recordIndex is*
 372 *moved down from line 3.*
 373
 374
 375
 376 *Initialization of total and done*
 377 *are moved down from lines 4*
 378 *and 5.*
 379
 380
 381
 382

```
...  
25 recordIndex = 0;  
26 while ( recordIndex < recordCount ) {  
27 ...  
28     recordIndex = recordIndex + 1;  
     ...  
62 total = 0;  
63 done = false;  
64 while ( !done ) {  
    ...  
69     if ( total > projectedTotal ) {  
70         done = true;
```

Here are the live times for the variables in this example:

<i>recordIndex</i>	(line 28-line 25 + 1) = 4
<i>total</i>	(line 69-line 62 + 1) = 8
<i>done</i>	(line 70-line 63 + 1) = 8
Average Live Time	(4 + 8 + 8) / 3 ≈ 7

383 **FURTHER READING** For
 384 more information on “live”
 385 variables, see *Software*
 386 *Engineering Metrics and*
 387 *Models* (Conte, Dunsmore,
 and Shen 1986).

Intuitively, the second example seems better than the first because the initializations for the variables are performed closer to where the variables are used. The measured difference in average live time between the two examples is significant: An average of 54 vs. an average of 7 provides good quantitative support for the intuitive preference for the second piece of code.

388
 389
 390

Does a hard number separate a good live time from a bad one? A good span from a bad one? Researchers haven’t yet produced that quantitative data, but it’s safe to assume that minimizing both span and live time is a good idea.

391
 392
 393

If you try to apply the ideas of span and live time to global variables, you’ll find that global variables have enormous spans and live times—one of many good reasons to avoid global variables.

394

395

396 **CROSS-REFERENCE** For
397 details on initializing
398 variables close to where
399 they're used, see Section
400 10.3, "Guidelines for
401 Initializing Variables," earlier
402 in this chapter.

403 **CROSS-REFERENCE** For
404 more on this style of variable
405 declaration and definition,
406 see "Ideally, declare and
407 Initialize each variable close
408 to where it's first used" in
Section 10.3, earlier in this
409 chapter.

410

411

412
413 **CROSS-REFERENCE** For
414 more details on keeping
415 related statements together,
see Section 14.2, "Statements
Whose Order Doesn't
416 Matter."

417

418

419 *Statements using two sets of
420 variables*

421

422

423

424

425

426

427

428

429

430

431

432

General Guidelines for Minimizing Scope

Here are some specific guidelines you can use to minimize scope.

Initialize variables used in a loop immediately before the loop rather than back at the beginning of the routine containing the loop

Doing this improves the chance that when you modify the loop, you'll remember to make corresponding modifications to the loop initialization. Later, when you modify the program and put another loop around the initial loop, the initialization will work on each pass through the new loop rather than on only the first pass.

Don't assign a value to a variable until just before the value is used

You might have experienced the frustration of trying to figure out where a variable was assigned its value. The more you can do to clarify where a variable receives its value, the better. Languages like C++ and Java support variable initializations like these:

C++ Example of Good Variable Declarations and Initializations

```
int receiptIndex = 0;  
float dailyReceipts = TodaysReceipts();  
double totalReceipts = TotalReceipts( dailyReceipts );
```

Group related statements

The following examples show a routine for summarizing daily receipts and illustrate how to put references to variables together so that they're easier to locate. The first example illustrates the violation of this principle:

C++ Example of Using Two Sets of Variables in a Confusing Way

```
void SummarizeData (...) {  
    ...  
    GetOldData( oldData, &numOldData );  
    GetNewData( newData, &numNewData );  
    totalOldData = Sum( oldData, numOldData );  
    totalNewData = Sum( newData, numNewData );  
    PrintOldDataSummary( oldData, totalOldData, numOldData );  
    PrintNewDataSummary( newData, totalNewData, numNewData );  
    SaveOldDataSummary( totalOldData, numOldData );  
    SaveNewDataSummary( totalNewData, numNewData );  
    ...  
}
```

Note that, in the example above, you have to keep track of *oldData*, *newData*, *numOldData*, *numNewData*, *totalOldData*, and *totalNewData* all at once—six variables for just this short fragment. The example below shows how to reduce that number to only three elements

```
433  
434  
435     Statements using oldData  
436  
437  
438  
439  
440     Statements using newData  
441  
442  
443  
444  
445  
446  
447  
448  
449
```

450 **CROSS-REFERENCE** For
451 more on global variables, see
452 Section 13.3, “Global Data.”

```
453  
454  
455  
456  
457  
458  
459  
460
```

```
461  
462  
463  
464  
465  
466  
467
```

C++ Example of Using Two Sets of Variables More Understandably

```
void SummarizeDaily( ... ) {  
    GetOldData( oldData, &numOldData );  
    totalOldData = Sum( oldData, numOldData );  
    PrintOldDataSummary( oldData, totalOldData, numOldData );  
    SaveOldDataSummary( totalOldData, numOldData );  
    ...  
    GetNewData( newData, &numNewData );  
    totalNewData = Sum( newData, numNewData );  
    PrintNewDataSummary( newData, totalNewData, numNewData );  
    SaveNewDataSummary( totalNewData, numNewData );  
    ...  
}
```

When the code is broken up as shown above, the two blocks are each shorter than the original block and individually contain fewer variables. They’re easier to understand, and if you need to break this code out into separate routines, the shorter blocks with fewer variables make better-defined routines.

Begin with most restricted visibility, and expand the variable’s scope only if necessary

Part of minimizing the scope of a variable is keeping it as local as possible. It is much more difficult to reduce the scope of a variable that has had a large scope than to expand the scope of a variable that has had a small scope—in other words, it’s harder to turn a global variable into a class variable than it is to turn a class variable into a global variable. It’s harder to turn a protected data member into a private data member than vice versa. For that reason, when in doubt, favor the smallest possible scope for a variable—local to an individual routine if possible, then private, then protected, then package (if your programming language supports that), and global only as a last resort.

Comments on Minimizing Scope

Many programmers’ approach to minimizing variables’ scope depends on their views of the issues of “convenience” and “intellectual manageability.” Some programmers make many of their variables global because global scope makes variables convenient to access and the programmers don’t have to fool around with parameter lists and class scoping rules. In their minds, the convenience of being able to access variables at any time outweighs the risks involved.

468 **CROSS-REFERENCE** The
469 idea of minimizing scope is
470 related to the idea of
471 information hiding. For
472 details, see “Hide Secrets
473 (Information Hiding)” in
Section 5.3.

473 **KEY POINT**

474
475
476
477
478
479
480

481 **CROSS-REFERENCE** For
482 details on using access
483 routines, see “Using Access
484 Routines Instead of Global
485 Data” in Section 13.3.

486
487

Other programmers prefer to keep their variables as local as possible because local scope helps intellectual manageability. The more information you can hide, the less you have to keep in mind at any one time. The less you have to keep in mind, the smaller the chance that you’ll make an error because you forgot one of the many details you needed to remember.

The difference between the “convenience” philosophy and the “intellectual manageability” philosophy boils down to a difference in emphasis between writing programs and reading them. Maximizing scope might indeed make programs easy to write, but a program in which any routine can use any variable at any time is harder to understand than a program that uses well-factored routines. In such a program, you can’t understand only one routine; you have to understand all the other routines with which that routine shares global data. Such programs are hard to read, hard to debug, and hard to modify.

Consequently, you should declare each variable to be visible to the smallest segment of code that needs to see it. If you can confine the variable’s scope to a single routine, great. If you can’t confine the scope to one routine, restrict the visibility to the routines in a single class. If you can’t restrict the variable’s scope to the class that’s most responsible for the variable, create access routines to share the variable’s data with other classes. You’ll find that you rarely if ever need to use naked global data.

488

10.5 Persistence

489
490

491
492
493
494
495
496
497
498
499
500
501
502

“Persistence” is another word for the life span of a piece of data. Persistence takes several forms. Some variables persist

- for the life of a particular block of code or routine. Variables declared inside a *for* loop in C++ or Java are examples of this kind of persistence.
- as long as you allow them to. In Java, variables created with *new* persist until they are garbage collected. In C++, variables created with *new* persist until you *delete* them.
- for the life of a program. Global variables in most languages fit this description, as do *static* variables in C++ and Java.
- forever. These variables might include values that you store in a database between executions of a program. For example, if you have an interactive program in which users can customize the color of the screen, you can store their colors in a file and then read them back each time the program is loaded.

503
504
505
506
507
508
509
510

The main problem with persistence arises when you assume that a variable has a longer persistence than it really does. The variable is like that jug of milk in your refrigerator. It's supposed to last a week. Sometimes it lasts a month, and sometimes it turns sour after five days. A variable can be just as unpredictable. If you try to use the value of a variable after its normal life span is over, will it have retained its value? Sometimes the value in the variable is sour, and you know that you've got an error. Other times, the computer leaves the old value in the variable, letting you imagine that you have used it correctly.

511

512 **CROSS-REFERENCE** Deb
513 ug code is easy to include in
514 access routines and is
discussed more in
515 "Advantages of Access
516 Routines" in Section 13.3.

517
518
519520
521

Here are a few steps you can take to avoid this kind of problem:

- Use debug code or assertions in your program to check critical variables for reasonable values. If the values aren't reasonable, display a warning that tells you to look for improper initialization.
- Write code that assumes data isn't persistent. For example, if a variable has a certain value when you exit a routine, don't assume it has the same value the next time you enter the routine. This doesn't apply if you're using language-specific features that guarantee the value will remain the same, such as *static* in C++ and Java.
- Develop the habit of declaring and initializing all data right before it's used. If you see data that's used without a nearby initialization, be suspicious!

522

10.6 Binding Time

523
524
525
526
527

An initialization topic with far-reaching implications for program maintenance and modifiability is "binding time"—the time at which the variable and its value are bound together (Thimbleby 1988). Are they bound together when the code is written? When it is compiled? When it is loaded? When the program is run? Some other time?

528
529
530
531

It can be to your advantage to use the latest binding time possible. In general, the later you make the binding time, the more flexibility you build into your code. The next example shows binding at the earliest possible time, when the code is written.

532

Java Example of a Variable That's Bound at Code-Writing Time

```
titleBar.color = 0xFF; // 0xFF is hex value for color blue
```

The value *0xFF* is bound to the variable *titleBar.color* at the time the code is written because *0xFF* is a literal value hard-coded into the program. Hard-coding like this is nearly always a bad idea because if this *0xFF* changes, it can get out of synch with *0xFF*'s used elsewhere in the code that must be the same value as this one.

539

Here's an example of binding at a slightly later time, when the code is compiled:

540

Java Example of a Variable That's Bound at Compile Time

541

```
private static final int COLOR_BLUE = 0xFF;  
private static final int TITLE_BAR_COLOR = COLOR_BLUE;  
...  
titleBar.color = TITLE_BAR_COLOR;
```

543

544

545

546

547

548

549

550

TITLE_BAR_COLOR is a named constant, an expression for which the compiler substitutes a value at compile time. This is nearly always better than hard-coding, if your language supports it. It increases readability because *TITLE_BAR_COLOR* tells you more about what is being represented than *0xFF* does. It makes changing the title bar color easier because one change accounts for all occurrences. And it doesn't incur a run-time performance penalty.

551

Here's an example of binding later, at run time:

552

Java Example of a Variable That's Bound at Run Time

553

554

555

```
titleBar.color = ReadTitleBarColor();
```

ReadTitleBarColor() is a routine that reads a value while a program is executing, perhaps from the Windows registry.

556

557

558

559

560

The code is more readable and flexible than it would be if a value were hard-coded. You don't need to change the program to change *titleBar.color*; you simply change the contents of the source that's read by *ReadTitleBarColor()*. This approach is commonly used for interactive applications in which a user can customize the application environment.

561

562

563

564

There is still another variation in binding time, which has to do with when the *ReadTitleBarColor()* routine is called. That routine could be called once at program load time, each time the window is created, or each time the window is drawn—each alternative representing successively later binding times.

565

566

To summarize, here are the times a variable can be bound to a value in this example (the details could vary somewhat in other cases):

567

- Coding time (use of magic numbers)
- Compile time (use of a named constant)
- Load time (reading a value from an external source such as the Windows Registry)
- Object instantiation time (such as reading the value each time a window is created)
- Just in time (such as reading the value each time the window is drawn)

568

569

570

571

572

573

574 In general, the earlier the binding time, the lower the flexibility and the lower the
575 complexity. For the first two options, using named constants is preferable to
576 using magic numbers for many reasons, so you can get the flexibility that named
577 constants provide just by using good programming practices. Beyond that, the
578 greater the flexibility desired, the higher the complexity of the code needed to
579 support that flexibility, and the more error-prone the code will be. Because
580 successful programming depends on minimizing complexity, a skilled
581 programmer will build in as much flexibility as needed to meet the software's
582 requirements but will not add flexibility—and related complexity—beyond
583 what's required.

584 585 10.7 Relationship Between Data Types and Control Structures

586 Data types and control structures relate to each other in well-defined ways that
587 were originally described by the British computer scientist Michael Jackson
588 (Jackson 1975). This section sketches the regular relationship between data and
589 control flow.

590 Jackson draws connections between three types of data and corresponding
591 control structures.

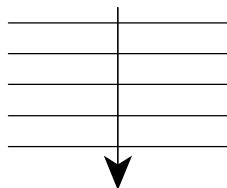
592 **CROSS-REFERENCE** For
593 details on sequences, see
594 Chapter 14, "Organizing
595 Straight-Line Code."
596
597

598
599
600
601

602 **CROSS-REFERENCE** For
603 details on conditionals, see
604 Chapter 15, "Using
605 Conditionals."

Sequential data translates to sequential statements in a program

Sequences consist of clusters of data used together in a certain order. If you have five statements in a row that handle five different values, they are sequential statements. If you read an employee's name, social security number, address, phone number, and age from a file, you'd have sequential statements in your program to read sequential data from the file.



F10xx02

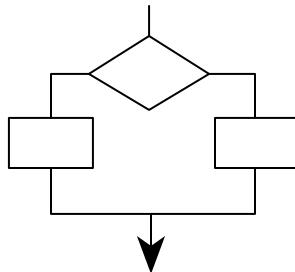
Figure 10-2

Sequential data is data that's handled in a defined order.

Selective data translates to if and case statements in a program

In general, selective data is a collection in which one of several pieces of data is present at any particular time—one of the elements is selected. The corresponding program statements must do the actual selection, and they consist

606
607
608
of *If-Then-Else* or *Case* statements. If you had an employee payroll program, you
might process employees differently depending on whether they were paid
hourly or salaried. Again, patterns in the code match patterns in the data.



609
610
611
612

613 **CROSS-REFERENCE** For
614 details on loops, see Chapter
615 16, "Controlling Loops."

616
617
618

619
620
621
622

623 Your real data can be combinations of the sequential, selective, and iterative
624 types of data. You can combine the simple building blocks to describe more
625 complicated data types.

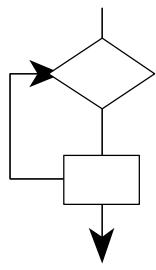
F10xx03

Figure 10-3

Selective data allows you to use one piece or the other, but not both.

Iterative data translates to for, repeat, and while looping structures in a program

Iterative data is the same type of data repeated several times. Typically, iterative data is stored as records in a file or in arrays. You might have a list of social security numbers that you read from a file. The iterative data would match the iterative code loop used to read the data.



F10xx04

Figure 10-4

Iterative data is repeated.

626

627

628 **KEY POINT**

629

630

631

632

633

634

635

636 **CODING HORROR**

637

638

639

640

641

642

643

644

645

646

647 **CROSS-REFERENCE** Routine parameters should also be used for one purpose only.
648 For details on using routine
649 parameters, see Section 7.5,
650 "How to Use Routine
Parameters."

652

653

654

655

656

657

658

659

660

661

662

10.8 Using Each Variable for Exactly One Purpose

It's possible to use variables for more than one purpose in several subtle ways. You're better off without this kind of subtlety.

Use each variable for one purpose only

It's sometimes tempting to use one variable in two different places for two different activities. Usually, the variable is named inappropriately for one of its uses, or a "temporary" variable is used in both cases (with the usual unhelpful name *x* or *temp*). Here's an example that shows a temporary variable that's used for two purposes:

C++ Example of Using One Variable for Two Purposes—Bad Practice

```
// Compute roots of a quadratic equation.  
// This code assumes that (b*b-4*a*c) is positive.  
temp = Sqrt( b*b - 4*a*c );  
root[0] = ( -b + temp ) / ( 2 * a );  
root[1] = ( -b - temp ) / ( 2 * a );  
...  
// swap the roots  
temp = root[0];  
root[0] = root[1];  
root[1] = temp;
```

Question: What is the relationship between *temp* in the first few lines and *temp* in the last few? Answer: The two *temp* have no relationship. Using the same variable in both instances makes it seem as though they're related when they're not. Creating unique variables for each purpose makes your code more readable. Here's an improvement on the example above:

C++ Example of Using Two Variables for Two Purposes—Good Practice

```
// Compute roots of a quadratic equation.  
// This code assumes that (b*b-4*a*c) is positive.  
discriminant = Sqrt( b*b - 4*a*c );  
root[0] = ( -b + discriminant ) / ( 2 * a );  
root[1] = ( -b - discriminant ) / ( 2 * a );  
...  
// swap the roots  
oldRoot = root[0];  
root[0] = root[1];  
root[1] = oldRoot;
```

663 **Avoid variables with hidden meanings**

664 Another way in which a variable can be used for more than one purpose is to
665 have different values for the variable mean different things. For example

666 **CODING HORROR**

667

668

669

670

671

672

673

674

675

676

677

678

679

680

681

682

683

684 **HARD DATA**

685

686

687

688

689

690 **CROSS-REFERENCE** For

691 a checklist that applies to
692 specific types of data rather
than general issues, see the
checklist in Chapter 12,
693 “Fundamental Data Types.”

694 For issues in naming
variables, see the checklist in
Chapter 11, “The Power of
695 Variable Names.”

696

697

698

- 699 Are variables reinitialized properly in code that's executed repeatedly?
700 Does the code compile with no warnings from the compiler?
701 If your language uses implicit declarations, have you compensated for the
702 problems they cause?

703 **Other General Issues in Using Data**

- 704 Do all variables have the smallest scope possible?
705 Are references to variables as close together as possible—both from each
706 reference to a variable to the next and in total live time?
707 Do control structures correspond to the data types?
708 Are all the declared variables being used?
709 Are all variables bound at appropriate times, that is, striking a conscious
710 balance between the flexibility of late binding and the increased complexity
711 associated with late binding?
712 Does each variable have one and only one purpose?
713 Is each variable's meaning explicit, with no hidden meanings?
714
-

715 **Key Points**

- 716
 - 717 • Data initialization is prone to errors, so use the initialization techniques
718 described in this chapter to avoid the problems caused by unexpected initial
719 values.
 - 720 • Minimize the scope of each variable. Keep references to it close together.
721 Keep it local to a routine or class. Avoid global data.
 - 722 • Keep statements that work with the same variables as close together as
723 possible.
 - 724 • Early binding tends to limit flexibility, but minimize complexity. Late
725 binding tends to increase flexibility, but at the price of increased complexity.
 - Use each variable for one and only one purpose.

11

2 The Power of Variable Names

3

4 CC2E.COM/1184

5 Contents

6 11.1 Considerations in Choosing Good Names

7 11.2 Naming Specific Types of Data

8 11.3 The Power of Naming Conventions

9 11.4 Informal Naming Conventions

10 11.5 Standardized Prefixes

11 11.6 Creating Short Names That Are Readable

12 11.7 Kinds of Names to Avoid

13 Related Topics

14 Routine names: Section 7.3

15 Class names: Section 6.2

16 General issues in using variables: Chapter 10

17 Documenting variables: “Commenting Data Declarations” in Section 32.5

18 AS IMPORTANT AS THE TOPIC OF GOOD NAMES IS to effective
19 programming, I have never read a discussion that covered more than a handful of
20 the dozens of considerations that go into creating good names. Many
21 programming texts devote a few paragraphs to choosing abbreviations, spout a
22 few platitudes, and expect you to fend for yourself. I intend to be guilty of the
23 opposite, to inundate you with more information about good names than you will
24 ever be able to use!

11.1 Considerations in Choosing Good Names

You can't give a variable a name the way you give a dog a name—because it's cute or it has a good sound. Unlike the dog and its name, which are different entities, a variable and a variable's name are essentially the same thing.

Consequently, the goodness or badness of a variable is largely determined by its name. Choose variable names with care.

Here's an example of code that uses bad variable names:

CODING HORROR

Java Example of Poor Variable Names

```
x = x - xx;
xxx = aretha + SalesTax( aretha );
x = x + LateFee( x1, x ) + xxx;
x = x + Interest( x1, x );
```

What's happening in this piece of code? What do *x1*, *xx*, and *xxx* mean? What does *aretha* mean? Suppose someone told you that the code computed a total customer bill based on an outstanding balance and a new set of purchases. Which variable would you use to print the customer's bill for just the new set of purchases?

Here's a different version of the same code that makes these questions easier to answer:

Java Example of Good Variable Names

```
balance = balance - lastPayment;
monthlyTotal = NewPurchases + SalesTax( newPurchases );
balance = balance + LateFee( customerID, balance ) + monthlyTotal;
balance = balance + Interest( customerID, balance );
```

In view of the contrast between these two pieces of code, a good variable name is readable, memorable, and appropriate. You can use several general rules of thumb to achieve these goals.

The Most Important Naming Consideration

The most important consideration in naming a variable is that the name fully and accurately describe the entity the variable represents. An effective technique for coming up with a good name is to state in words what the variable represents. Often that statement itself is the best variable name. It's easy to read because it doesn't contain cryptic abbreviations, and it's unambiguous. Because it's a full

KEY POINT

59 description of the entity, it won't be confused with something else. And it's easy
60 to remember because the name is similar to the concept.

61 For a variable that represents the number of people on the U.S. Olympic team,
62 you would create the name *numberOfPeopleOnTheUsOlympicTeam*.

63 A variable that represents the number of seats in a stadium would be
64 *numberOfSeatsInTheStadium*. A variable that represents the maximum number
65 of points scored by a country's team in any modern Olympics would be
66 *maximumNumberOfPointsInModernOlympics*. A variable that contains the
67 current interest rate is better named *rate* or *interestRate* than *r* or *x*. You get the
68 idea.

69 Note two characteristics of these names. First, they're easy to decipher. In fact,
70 they don't need to be deciphered at all because you can simply read them. But
71 second, some of the names are long—too long to be practical. I'll get to the
72 question of variable-name length shortly.

73 Here are several examples of variable names, good and bad:

Table 11-1. Examples of Good and Bad Variable Names

Purpose of Variable	Good Names, Good Descriptors	Bad Names, Poor Descriptors
Running total of checks written to date	<i>runningTotal, checkTotal, nChecks</i>	<i>written, ct, checks, CHKTTL, x, x1, x2</i>
Velocity of a bullet train	<i>velocity, trainVelocity, velocityInMph</i>	<i>velt, v, tv, x, x1, x2, train</i>
Current date	<i>currentDate, todaysDate</i>	<i>cd, current, c, x, x1, x2, date</i>
Lines per page	<i>linesPerPage</i>	<i>lpp, lines, l, x, x1, x2</i>

75 The names *currentDate* and *todaysDate* are good names because they fully and
76 accurately describe the idea of "current date." In fact, they use the obvious
77 words. Programmers sometimes overlook using the ordinary words, which is
78 often the easiest solution. *cd* and *c* are poor names because they're too short and
79 not at all descriptive. *current* is poor because it doesn't tell you what is current.
80 *date* is almost a good name, but it's a poor name in the final analysis because the
81 date involved isn't just any date, but the current date. *date* by itself gives no such
82 indication. *x*, *x1*, and *x2* are poor names because they're always poor names—*x*
83 traditionally represents an unknown quantity; if you don't want your variables to
84 be unknown quantities, think of better names.

85 **KEY POINT**
86 Names should be as specific as possible. Names like *x*, *temp*, and *i* that are
87 general enough to be used for more than one purpose are not as informative as
they could be and are usually bad names.

88

Problem-Orientation

89

A good mnemonic name generally speaks to the problem rather than the solution. A good name tends to express the *what* more than the *how*. In general, if a name refers to some aspect of computing rather than to the problem, it's a *how* rather than a *what*. Avoid such a name in favor of a name that refers to the problem itself.

94

A record of employee data could be called *inputRec* or *employeeData*. *inputRec* is a computer term that refers to computing ideas—input and record. *employeeData* refers to the problem domain rather than the computing universe. Similarly, for a bit field indicating printer status, *bitFlag* is a more computerish name than *printerReady*. In an accounting application, *calcVal* is more computerish than *sum*.

95

96

97

98

99

100

Optimum Name Length

101

The optimum length for a name seems to be somewhere between the lengths of *x* and *maximumNumberOfPointsInModernOlympics*. Names that are too short don't convey enough meaning. The problem with names like *x1* and *x2* is that even if you can discover what *x* is, you won't know anything about the relationship between *x1* and *x2*. Names that are too long are hard to type and can obscure the visual structure of a program.

107 | **HARD DATA**

108

109

110

111

112

113

Gorla, Benander, and Benander found that the effort required to debug a program was minimized when variables had names that averaged 10 to 16 characters (1990). Programs with names averaging 8 to 20 characters were almost as easy to debug. The guideline doesn't mean that you should try to make all of your variable names 9 to 15 or 10 to 16 characters long. It does mean that if you look over your code and see many names that are shorter, you should check to be sure that the names are as clear as they need to be.

114

115

You'll probably come out ahead by taking the Goldilocks-and-the-Three-Bears approach to naming variables:

116

117

Table 11-2. Variable Names That are Too Long, Too Short, and Just Right

Too long: *numberOfPeopleOnTheUsOlympicTeam*
 numberOfSeatsInTheStadium
 maximumNumberOfPointsInModernOlympics

Too short: *n, np, ntm*
 n, ns, nsid
 m, mp, max, points

Just right: *numTeamMembers, teamMemberCount*

*numSeatsInStadium, seatCount
teamPointsMax, pointsRecord*

118

119 **CROSS-REFERENCE** Scop
120 e is discussed in more detail
121 in Section 10.4, “Scope.”

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

The Effect of Scope on Variable Names

Are short variable names always bad? No, not always. When you give a variable a short name like *i*, the length itself says something about the variable—namely, that the variable is a scratch value with a limited scope of operation.

A programmer reading such a variable should be able to assume that its value isn’t used outside a few lines of code. When you name a variable *i*, you’re saying, “This variable is a run-of-the-mill loop counter or array index and doesn’t have any significance outside these few lines of code.”

A study by W. J. Hansen found that longer names are better for rarely used variables or global variables and shorter names are better for local variables or loop variables (Shneiderman 1980). Short names are subject to many problems, however, and some careful programmers avoid them altogether as a matter of defensive-programming policy.

Use qualifiers on names that are in the global name space

If you have variables that are in the global namespace (named constants, class names, and so on), consider whether you need to adopt a convention for partitioning the global namespace and avoiding naming conflicts. In C++ and C#, you can use the *namespace* keyword to partition the global namespace.

C++ Example of Using the *namespace* Keyword to Partition the Global Namespace

```
138 namespace UserInterfaceSubsystem {  
139     ...  
140     // lots of declarations  
141     ...  
142 }  
  
144 namespace DatabaseSubsystem {  
145     ...  
146     // lots of declarations  
147     ...  
148 }
```

If you declare an *Employee* class in both the *UserInterfaceSubsystem* and the *DatabaseSubsystem*, you can identify which you wanted to refer to by writing *UserInterfaceSubsystem::Employee* or *DatabaseSubsystem::Employee*. In Java, you can accomplish the same thing through the use of packages.

153 In languages that don't support namespaces or packages, you can still use
154 naming conventions to partition the global name space. One convention is to
155 require that globally-visible classes be prefixed with subsystem mnemonic. Thus
156 the user interface employee class might become *uiEmployee*, and the database
157 employee class might become *dbEmployee*. This minimizes the risk of global-
158 namespace collisions.

159 **Computed-Value Qualifiers in Variable Names**

160 Many programs have variables that contain computed values: totals, averages,
161 maximums, and so on. If you modify a name with a qualifier like *Total*, *Sum*,
162 *Average*, *Max*, *Min*, *Record*, *String*, or *Pointer*, put the modifier at the end of the
163 name.

164 This practice offers several advantages. First, the most significant part of the
165 variable name, the part that gives the variable most of its meaning, is at the front,
166 so it's most prominent and gets read first. Second, by establishing this
167 convention, you avoid the confusion you might create if you were to use both
168 *totalRevenue* and *revenueTotal* in the same program. The names are semantically
169 equivalent, and the convention would prevent their being used as if they were
170 different. Third, a set of names like *revenueTotal*, *expenseTotal*,
171 *revenueAverage*, and *expenseAverage* has a pleasing symmetry. A set of names
172 like *totalRevenue*, *expenseTotal*, *revenueAverage*, and *averageExpense* doesn't
173 appeal to a sense of order. Finally, the consistency improves readability and
174 eases maintenance.

175 An exception to the rule that computed values go at the end of the name is the
176 customary position of the *Num* qualifier. Placed at the beginning of a variable
177 name, *Num* refers to a total. *numSales* is the total number of sales. Placed at the
178 end of the variable name, *Num* refers to an index. *saleNum* is the number of the
179 current sale. The *s* at the end of *numSales* is another tip-off about the difference
180 in meaning. But, because using *Num* so often creates confusion, it's probably
181 best to sidestep the whole issue by using *Count* or *Total* to refer to a total number
182 of sales and *Index* to refer to a specific sale. Thus, *salesCount* is the total number
183 of sales and *salesIndex* refers to a specific sale.

184 **Common Opposites in Variable Names**

185 **CROSS-REFERENCE** For
186 a similar list of opposites in
187 routine names, see "Provide
188 services in pairs with their
189 opposites" in Section 6.2.

Use opposites precisely. Using naming conventions for opposites helps consistency, which helps readability. Pairs like *begin/end* are easy to understand and remember. Pairs that depart from common-language opposites tend to be hard to remember and are therefore confusing. Here are some common opposites:

- 190 • begin/end
191 • first/last
192 • locked/unlocked
193 • min/max
194 • next/previous
195 • old/new
196 • opened/closed
197 • visible/invisible
198 • source/target
199 • source/destination (less common)
200 • up/down

201 11.2 Naming Specific Types of Data

202 In addition to the general considerations in naming data, special considerations
203 come up in the naming of specific kinds of data. This section describes
204 considerations specifically for loop variables, status variables, temporary
205 variables, boolean variables, enumerated types, and named constants.

206 Naming Loop Indexes

207 **CROSS-REFERENCE** For
208 details on loops, see Chapter
16, “Controlling Loops.”

209 The names *i*, *j*, and *k* are customary:

210 Java Example of a Simple Loop Variable Name

```
211 for ( i = firstItem; i < lastItem; i++ ) {  
212     data[ i ] = 0;  
213 }
```

214 If a variable is to be used outside the loop, it should be given a more meaningful
215 name than *i*, *j*, or *k*. For example, if you are reading records from a file and need
216 to remember how many records you’ve read, a more meaningful name like
217 *recordCount* would be appropriate:

218 Java Example of a Good Descriptive Loop Variable Name

```
219 recordCount = 0;  
220 while ( moreScores() ) {
```

```
221     score[ recordCount ] = GetNextScore();
222     recordCount++;
223 }
224
225 // lines using recordCount
226 ...
227
228
229
230
```

If the loop is longer than a few lines, it's easy to forget what *i* is supposed to stand for, and you're better off giving the loop index a more meaningful name. Because code is so often changed, expanded, and copied into other programs, many experienced programmers avoid names like *i* altogether.

```
231
232
```

One common reason loops grow longer is that they're nested. If you have several nested loops, assign longer names to the loop variables to improve readability.

233 Java Example of Good Loop Names in a Nested Loop

```
234 for ( teamIndex = 0; teamIndex < teamCount; teamIndex++ ) {
235     for ( eventIndex = 0; eventIndex < eventCount[ teamIndex ]; eventIndex++ ) {
236         score[ teamIndex ][ eventIndex ] = 0;
237     }
238 }
```

Carefully chosen names for loop-index variables avoid the common problem of index cross talk: saying *i* when you mean *j* and *j* when you mean *i*. They also make array accesses clearer. *score[teamIndex][eventIndex]* is more informative than *score[i][j]*.

```
243
244
245
246
```

If you have to use *i*, *j*, and *k*, don't use them for anything other than loop indexes for simple loops—the convention is too well established, and breaking it to use them in other ways is confusing. The simplest way to avoid such problems is simply to think of more descriptive names than *i*, *j*, and *k*.

247 Naming Status Variables

```
248
249
```

Status variables describe the state of your program. The rest of this section gives some guidelines for naming them.

250 *Think of a better name than flag for status variables*

```
251 It's better to think of flags as status variables. A flag should never have flag in its
252 name because that doesn't give you any clue about what the flag does. For
253 clarity, flags should be assigned values and their values should be tested with
254 enumerated types, named constants, or global variables that act as named
255 constants. Here are some examples of flags with bad names:
```

256 CODING HORROR

```
257
```

```
if ( flag ) ...
```

```
258     if ( statusFlag & 0x0F ) ...
259     if ( printFlag == 16 ) ...
260     if ( computeFlag == 0 ) ...
261
262     flag = 0x1;
263     statusFlag = 0x80;
264     printFlag = 16;
265     computeFlag = 0;
```

Statements like *statusFlag* = *0x80* give you no clue about what the code does unless you wrote the code or have documentation that tells you both what *statusFlag* is and what *0x80* represents. Here are equivalent code examples that are clearer:

270 C++ Examples of Better Use of Status Variables

```
271     if ( dataReady ) ...
272     if ( characterType & PRINTABLE_CHAR ) ...
273     if ( reportType == ReportType_Annual ) ...
274     if ( recalcNeeded == True ) ...
275
276     dataReady = True;
277     characterType = CONTROL_CHARACTER;
278     reportType = ReportType_Annual;
279     recalcNeeded = False;
```

Clearly, *characterType* = *CONTROL_CHARACTER*, from the second code example, is more meaningful than *statusFlag* = *0x80*, from the first. Likewise, the conditional *if(reportType == ReportType_Annual)* is clearer than *if(printFlag == 16)*. The second example shows that you can use this approach with enumerated types as well as predefined named constants. Here's how you could use named constants and enumerated types to set up the values used in the example:

287 Declaring Status Variables in C++

```
288 // values for CharacterType
289 const int LETTER = 0x01;
290 const int DIGIT = 0x02;
291 const int PUNCTUATION = 0x04;
292 const int LINE_DRAW = 0x08;
293 const int PRINTABLE_CHAR = ( LETTER | DIGIT | PUNCTUATION | LINE_DRAW );
294
295 const int CONTROL_CHARACTER = 0x80;
296
297 // values for ReportType
298 enum ReportType {
299     ReportType_Daily,
300     ReportType_Monthly,
```

```
301     ReportType_Quarterly,  
302     ReportType_Annual,  
303     ReportType_All  
304 };
```

When you find yourself “figuring out” a section of code, consider renaming the variables. It’s OK to figure out murder mysteries, but you shouldn’t need to figure out code. You should be able to read it.

308 Naming Temporary Variables

309 Temporary variables are used to hold intermediate results of calculations, as
310 temporary placeholders, and to hold housekeeping values. They’re usually called
311 *temp*, *x*, or some other vague and nondescriptive name. In general, temporary
312 variables are a sign that the programmer does not yet fully understand the
313 problem. Moreover, because the variables are officially given a “temporary”
314 status, programmers tend to treat them more casually than other variables,
315 increasing the chance of errors.

316 *Be leery of “temporary” variables*

317 It’s often necessary to preserve values temporarily. But in one way or another,
318 most of the variables in your program are temporary. Calling a few of them
319 temporary may indicate that you aren’t sure of their real purposes. Consider the
320 following example.

321 C++ Example of an Uninformative “Temporary” Variable Name

```
322 // Compute roots of a quadratic equation.  
323 // This assumes that (b^2-4*a*c) is positive.  
324 temp = sqrt( b^2 - 4*a*c );  
325 root[0] = ( -b + temp ) / ( 2 * a );  
326 root[1] = ( -b - temp ) / ( 2 * a );
```

327 It’s fine to store the value of the expression *sqrt(b² - 4 * a * c)* in a variable,
328 especially since it’s used in two places later. But the name *temp* doesn’t tell you
329 anything about what the variable does. A better approach is shown in this
330 example:

331 C++ Example with a “Temporary” Variable Name Replaced with a Real 332 Variable

```
333 // Compute roots of a quadratic equation.  
334 // This assumes that (b^2-4*a*c) is positive.  
335 discriminant = sqrt( b^2 - 4*a*c );  
336 root[0] = ( -b + discriminant ) / ( 2 * a );  
337 root[1] = ( -b - discriminant ) / ( 2 * a );
```

338 This is essentially the same code, but it’s improved with the use of an accurate,
339 descriptive variable name.

340 Naming Boolean Variables

341 Here are a few guidelines to use in naming boolean variables:

342 *Keep typical boolean names in mind*

343 Here are some particularly useful boolean variable names:

- 344 • **done** Use *done* to indicate whether something is done. The variable can
345 indicate whether a loop is done or some other operation is done. Set *done* to
346 *False* before something is done, and set it to *True* when something is
347 completed.
- 348 • **error** Use *error* to indicate that an error has occurred. Set the variable to
349 *False* when no error has occurred and to *True* when an error has occurred.
- 350 • **found** Use *found* to indicate whether a value has been found. Set *found* to
351 *False* when the value has not been found and to *True* once the value has
352 been found. Use *found* when searching an array for a value, a file for an
353 employee ID, a list of paychecks for a certain paycheck amount, and so on.
- 354 • **success** Use *success* to indicate whether an operation has been successful.
355 Set the variable to *False* when an operation has failed and to *True* when an
356 operation has succeeded. If you can, replace *success* with a more specific
357 name that describes precisely what it means to be successful. If the program
358 is successful when processing is complete, you might use
359 *processingComplete* instead. If the program is successful when a value is
360 found, you might use *found* instead.

361 *Give boolean variables names that imply True or False*

362 Names like *done* and *success* are good boolean names because the state is either
363 *True* or *False*; something is done or it isn't; it's a success or it isn't. Names like
364 *status* and *sourceFile*, on the other hand, are poor boolean names because they're
365 not obviously *True* or *False*. What does it mean if *status* is *True*? Does it mean
366 that something has a status? Everything has a status. Does *True* mean that the
367 status of something is OK? Or does *False* mean that nothing has gone wrong?
368 With a name like *status*, you can't tell.

369 For better results, replace *status* with a name like *error* or *statusOK*, and replace
370 *sourceFile* with *sourceFileAvailable* or *sourceFileFound*, or whatever the
371 variable represents.

372 Some programmers like to put *Is* in front of their boolean names. Then the
373 variable name becomes a question: *isdone?* *isError?* *isFound?*
374 *isProcessingComplete?* Answering the question with *True* or *False* provides the
375 value of the variable. A benefit of this approach is that it won't work with vague
376 names: *isStatus?* makes no sense at all.

377 Use positive boolean variable names

378 Negative names like *notFound*, *notdone*, and *notSuccessful* are difficult to read
379 when they are negated—for example,

380 if not *notFound*

381 Such a name should be replaced by *found*, *done*, or *processingComplete* and then
382 negated with an operator as appropriate. If what you’re looking for is found, you
383 have *found* instead of *not notFound*.

384 Naming Enumerated Types

385 **CROSS-REFERENCE** For
386 details on using enumerated
387 types, see Section 12.6,
388 “Enumerated Types.”

**389 Visual Basic Example of Using a Suffix Naming Convention for
390 Enumerated Types**

```
391      Public Enum Color
392            Color_Red
393            Color_Green
394            Color_Blue
395      End Enum
396
397      Public Enum Planet
398            Planet_Earth
399            Planet_Mars
400            Planet_Venus
401      End Enum
402
403      Public Enum Month
404            Month_January
405            Month_February
406            ...
407            Month_December
408      End Enum
```

409 In addition, the enum type itself (*Color*, *Planet*, or *Month*) can be identified in
410 various ways, including all caps or prefixes (*e_Color*, *e_Planet*, or *e_Month*). A
411 person could argue that an enum is essentially a user-defined type, and so the
412 name of the enum should be formatted the same as other user-defined types like
413 classes. A different argument would be that enums are types, but they are also
414 constants, so the enum type name should be formatted as constants. This book
415 uses the convention of all caps for enumerated type names.

416

417 **CROSS-REFERENCE** For
418 details on using named
419 constants, see Section 12.7,
420 "Named Constants."

421

422

423

424

425

426

427

428

429

430

431

432

433

434

435

436

437

438

439

440

441

442

443

444

445

446

447

448

449

450

Naming Constants

When naming constants, name the abstract entity the constant represents rather than the number the constant refers to. *FIVE* is a bad name for a constant (regardless of whether the value it represents is *5.0*). *CYCLES_NEEDED* is a good name. *CYCLES_NEEDED* can equal *5.0* or *6.0*. *FIVE = 6.0* would be ridiculous. By the same token, *BAKERS_DOZEN* is a poor constant name; *DONUTS_MAX* is a good constant name.

11.3 The Power of Naming Conventions

Some programmers resist standards and conventions—and with good reason. Some standards and conventions are rigid and ineffective—destructive to creativity and program quality. This is unfortunate since effective standards are some of the most powerful tools at your disposal. This section discusses why, when, and how you should create your own standards for naming variables.

Why Have Conventions?

Conventions offer several specific benefits:

- They let you take more for granted. By making one global decision rather than many local ones, you can concentrate on the more important characteristics of the code.
- They help you transfer knowledge across projects. Similarities in names give you an easier and more confident understanding of what unfamiliar variables are supposed to do.
- They help you learn code more quickly on a new project. Rather than learning that Anita's code looks like this, Julia's like that, and Kristin's like something else, you can work with a more consistent set of code.
- They reduce name proliferation. Without naming conventions, you can easily call the same thing by two different names. For example, you might call total points both *pointTotal* and *totalPoints*. This might not be confusing to you when you write the code, but it can be enormously confusing to a new programmer who reads it later.
- They compensate for language weaknesses. You can use conventions to emulate named constants and enumerated types. The conventions can differentiate among local, class, and global data and can incorporate type information for types that aren't supported by the compiler.
- They emphasize relationships among related items. If you use object data, the compiler takes care of this automatically. If your language doesn't

451 support objects, you can supplement it with a naming convention. Names
452 like *address*, *phone*, and *name* don't indicate that the variables are related.
453 But suppose you decide that all employee-data variables should begin with
454 an *Employee* prefix. *employeeAddress*, *employeePhone*, and *employeeName*
455 leave no doubt that the variables are related. Programming conventions can
456 make up for the weakness of the language you're using.

457 **KEY POINT**

458 The key is that any convention at all is often better than no convention. The
459 convention may be arbitrary. The power of naming conventions doesn't come
460 from the specific convention chosen but from the fact that a convention exists,
adding structure to the code and giving you fewer things to worry about.

461 **When You Should Have a Naming Convention**

462 There are no hard-and-fast rules for when you should establish a naming
463 convention, but here are a few cases in which conventions are worthwhile:

- 464
- 465 • When multiple programmers are working on a project
 - 466 • When you plan to turn a program over to another programmer for
modifications and maintenance (which is nearly always)
 - 467 • When your programs are reviewed by other programmers in your
organization
 - 468 • When your program is so large that you can't hold the whole thing in your
brain at once and must think about it in pieces
 - 469 • When the program will be long-lived enough that you might put it aside for
a few weeks or months before working on it again
 - 470 • When you have a lot of unusual terms that are common on a project and
want to have standard terms or abbreviations to use in coding

475 **KEY POINT**

476 You always benefit from having some kind of naming convention. The
477 considerations above should help you determine the extent of the convention to
use on a particular project.

478

479 **CROSS-REFERENCE** For
480 details on the differences in
481 formality in small and large
482 projects, see Chapter 27,
483 "How Program Size Affects
Construction."

484

485

486

Degrees of Formality

Different conventions have different degrees of formality. An informal convention might be as simple as the rule "Use meaningful names." Somewhat more formal conventions are described in the next section. In general, the degree of formality you need is dependent on the number of people working on a program, the size of the program, and the program's expected life span. On tiny, throwaway projects, a strict convention might be unnecessary overhead. On larger projects in which several people are involved, either initially or over the program's life span, formal conventions are an indispensable aid to readability.

487

488

489

490

491

492

493

494

495

496

497 KEY POINT

498

499

500

501

502

503

504

505

506

507

508

509

510

511

512

513

514

515

516

517

518

11.4 Informal Naming Conventions

Most projects use relatively informal naming conventions such as the ones laid out in this section.

Guidelines for a Language-Independent Convention

Here are some guidelines for creating a language-independent convention:

Differentiate between variable names and routine names

A convention associated with Java programming is to begin variable and object names with lower case and routine names with upper case: *variableName* vs. *RoutineName()*.

Differentiate between classes and objects

The correspondence between class names and object names—or between types and variables of those types—can get tricky. There are several standard options, as shown in the following examples:

Option 1: Differentiating Types and Variables via Initial Capitalization

```
Widget widget;
LongerWidget longerWidget;
```

Option 2: Differentiating Types and Variables via All Caps

```
WIDGET widget;
LONGERWIDGET longerWidget
```

Option 3: Differentiating Types and Variables via the “t_” Prefix for Types

```
t_Widget Widget;
t_LongerWidget LongerWidget;
```

Option 4: Differentiating Types and Variables via the “a” Prefix for Variables

```
Widget aWidget;
LongerWidget aLongerWidget;
```

Option 5: Differentiating Types and Variables via Using More Specific Names for the Variables

```
Widget employeeWidget;
LongerWidget fullEmployeeWidget;
```

519

Each of these options has strengths and weaknesses.

520

Option 1 is a common convention in case-sensitive languages including C++ and Java, but some programmers are uncomfortable differentiating names solely on the basis of capitalization. Indeed, creating names that differ only in the capitalization of the first letter in the name seems to provide too little “psychological distance” and too small a visual distinction between the two names.

521

522

523

524

525

526

527

528

529

530

531

532

533

534

535

536

537

538

539

540

541

542

543

544

545

546

547

548

549

550

551

552

553

The Option 1 approach can't be applied consistently in mixed-language environments if any of the languages are case insensitive. In Visual Basic, for example,

```
Dim widget as Widget
```

will generate a syntax error, because *widget* and *Widget* are treated as the same token.

Option 2 creates a more obvious distinction between the type name and the variable name. For historical reasons, all caps are used to indicate constants in C++ and Java, however, and the approach is subject to the same problems in work in mixed-language environments that Option 1 is subject to.

Option 3 works adequately in all languages, but some programmers dislike the idea of prefixes for aesthetic reasons.

Option 4 is sometimes used as an alternative to Option 3, but it has the drawback of altering the name of every instance of a class instead of just the one class name.

Option 5 requires more thought on a variable-by-variable basis. In most instances, being forced to think of a specific name for a variable results in more readable code. But sometimes a *widget* truly is just a generic *widget*, and in those instances you'll find yourself coming up with less-than-obvious names, like *genericWidget*, which are arguably less readable. The code in this book uses Option 5 because it's the most understandable in situations in which the person reading the code isn't necessarily familiar with a less intuitive naming convention.

In short, each of the available options involves tradeoffs. I tend to prefer Option 3 because it works across multiple languages, and I'd rather have the odd prefix on the class name than on each and every object name. It's also easy to extend the convention consistently to named constants, enumerated types, and other kinds of types if desired.

554
555
556
557
On balance, Option 3 is a little like Winston's Churchill's description of
democracy: It has been said that democracy is the worst form of government that
has been tried, except for all the others. Option 3 is a terrible naming convention,
except for all the others that have been tried.

558 ***Identify global variables***

559 One common programming problem is misuse of global variables. If you give all
560 global variable names a *g_* prefix, for example, a programmer seeing the variable
561 *g_RunningTotal* will know it's a global variable and treat it as such.

562 ***Identify member variables***

563 Identify a class's member data. Make it clear that the variable isn't a local
564 variable and that it isn't a global variable either. For example, you can identify
565 class member variables with an *m_* prefix to indicate that it is member data.

566 ***Identify type definitions***

567 Naming conventions for types serve two purposes: They explicitly identify a
568 name as a type name, and they avoid naming clashes with variables. To meet
569 those considerations, a prefix or suffix is a good approach. In C++, the
570 customary approach is to use all uppercase letters for a type name—for example,
571 *COLOR* and *MENU*. (This convention applies to *typedefs* and *structs*, not class
572 names.) But this creates the possibility of confusion with named preprocessor
573 constants. To avoid confusion, you can prefix the type names with *t_*, such as
574 *t_Color* and *t_Menu*.

575 ***Identify named constants***

576 Named constants need to be identified so that you can tell whether you're
577 assigning a variable a value from another variable (whose value might change)
578 or from a named constant. In Visual Basic you have the additional possibility
579 that the value might be from a function. Visual Basic doesn't require function
580 names to use parentheses, whereas in C++ even a function with no parameters
581 uses parentheses.

582 One approach to naming constants is to use a prefix like *c_* for constant names.
583 That would give you names like *c_RecsMax* or *c_LinesPerPageMax*. In C++ and
584 Java, the convention is to use all uppercase letters, possibly with underscores to
585 separate words, *RECSMAX* or *RECS_MAX* and *LINESPERPAGEMAX* or
586 *LINES_PER_PAGE_MAX*.

587 ***Identify elements of enumerated types***

588 Elements of enumerated types need to be identified for the same reasons that
589 named constants do: to make it easy to tell that the name is for an enumerated
590 type as opposed to a variable, named constant, or function. The standard
591 approach applies; you can use all caps or an *e_* or *E_* prefix for the name of the

592 type itself, and use a prefix based on the specific type like *Color_* or *Planet_* for
593 the members of the type.

594 ***Identify input-only parameters in languages that don't enforce them***
595 Sometimes input parameters are accidentally modified. In languages such as
596 C++ and Visual Basic, you must indicate explicitly whether you want a value
597 that's been modified to be returned to the calling routine. This is indicated with
598 the *, &, and *const* qualifiers in C++ or *ByRef* and *ByVal* in Visual Basic.

599 In other languages, if you modify an input variable it is returned whether you
600 like it or not. This is especially true when passing objects. In Java, for example,
601 all objects are passed "by value," but the contents of an object can be changed
602 within the called routine (Arnold, Gosling, Holmes 2000).

603 **CROSS-REFERENCE** Aug
604 menting a language with a
605 naming convention to make
606 up for limitations in the
607 language itself is an example
608 of programming *into* a
language instead of just
609 programming in it. For more
610 details on programming *into*
611 a language, see Section 34.4,
612 "Program Into Your
Language, Not In It."

613
614
615
616
617
618
619
620
621
622

623 Guidelines for Language-Specific Conventions

624 Follow the naming conventions of the language you're using. You can find
625 books for most languages that describe style guidelines. Guidelines for C, C++,
626 Java, and Visual Basic are provided in the sections below.

627

628 **FURTHER READING** For
629 more on Java programming
style, see *The Elements of
630 Java Style*, 2d ed.
(Vermeulen et al, 2000).

631

632

633

634

635

636

637

638

639

640

641

642 **FURTHER READING** For
more on C++ programming
643 style, see *The Elements of
C++ Style* (Bumgardner,
644 Gray, and Misfeldt 2004).

645

646

647

648

649

650

651

652 **FURTHER READING** The
classic book on C
653 programming style is *C
654 Programming Guidelines*
(Plum 1984).

655

656

657

658

Java Conventions

In contrast with C and C++, Java style conventions have been well established since the beginning.

- *i* and *j* are integer indexes.
- Constants are in *ALL_CAPS* separated by underscores.
- Class and interface names capitalize the first letter of each word, including the first—for example, *ClassOrInterfaceName*.
- Variable and method names use lowercase for the first word, with the first letter of each following word capitalized—for example, *variableOrRoutineName*.
- The underscore is not used as a separator within names except for names in all caps.
- *get* and *set* prefixes are used for methods within a class that is currently a *Bean* or planned to become a *Bean* at a later time.

C++ Conventions

Here are the conventions that have grown up around C++ programming.

- *i* and *j* are integer indexes.
- *p* is a pointer.
- Constants, typedefs, and preprocessor macros are in *ALL_CAPS*.
- Class, variable and routine names are in *MixedUpperAndLowerCase()*.
- The underscore is not used as a separator within names, except for names in all caps and certain kinds of prefixes (such as to identify global variables).

As with C programming, this convention is far from standard, and different environments have standardized on different convention details.

C Conventions

Several naming conventions apply specifically to the C programming language. You may use these conventions in C, or you may adapt them to other languages.

- *c* and *ch* are character variables.
- *i* and *j* are integer indexes.
- *n* is a number of something.
- *p* is a pointer.
- *s* is a string.

- 659 • Preprocessor macros are in *ALL_CAPS*. This is usually extended to include
660 typedefs as well.
661 • Variable and routine names are in *all_lower_case*.
662 • The underscore (_) character is used as a separator: *lower_case* is more
663 readable than *lowercase*.

664 These are the conventions for generic, UNIX-style and Linux-style C
665 programming, but C conventions are different in different environments. In
666 Microsoft Windows, C programmers tend to use a form of the Hungarian naming
667 convention and mixed uppercase and lowercase letters for variable names. On
668 the Macintosh, C programmers tend to use mixed-case names for routines
669 because the Macintosh toolbox and operating-system routines were originally
670 designed for a Pascal interface.

671 **Visual Basic Conventions**

672 Visual Basic has not really established firm conventions. The next section
673 recommends a convention for Visual Basic.

674 **Mixed-Language Programming Considerations**

675 When programming in a mixed-language environment, the naming conventions
676 (as well as formatting conventions, documentation conventions, and other
677 conventions) may be optimized for overall consistency and readability—even if
678 that means going against convention for one of the languages that's part of the
679 mix.

680 In this book, for example, variable names all begin with lower case, which is
681 consistent with conventional Java programming practice and some but not all
682 C++ conventions. This book formats all routine names with an initial capital
683 letter, which follows the C++ convention; the Java convention would be to begin
684 method names with lower case, but this book uses routine names that begin in
685 uppercase across all languages for the sake of overall readability.

686 **Sample Naming Conventions**

687 The standard conventions above tend to ignore several important aspects of
688 naming that were discussed over the past few pages—including variable scoping
689 (private, class, or global), differentiating between class, object, routine, and
690 variable names, and other issues.

691 The naming-convention guidelines can look complicated when they're strung
692 across several pages. They don't need to be terribly complex, however, and you
693 can adapt them to your needs. Variable names include three kinds of
694 information:

- 695 • The contents of the variable (what it represents)
 696 • The kind of data (named constant, primitive variable, user-defined type, or
 697 class)
 698 • The scope of the variable (private, class, package, or global)

699 Here are examples of naming conventions for C, C++, Java, and Visual Basic
 700 that have been adapted from the guidelines presented earlier. These specific
 701 conventions aren't necessarily recommended, but they give you an idea of what
 702 an informal naming convention includes.

703 **Table 11-3. Sample Naming Convention for C++, and Java**

Entity	Description
<i>ClassName</i>	Class names are in mixed upper and lower case with an initial capital letter.
<i>TypeName</i>	Type definitions including enumerated types and typedefs use mixed upper and lower case with an initial capital letter
<i>EnumeratedTypes</i>	In addition to the rule above, enumerated types are always stated in the plural form.
<i>localVariable</i>	Local variables are in mixed uppercase and lowercase with an initial lower case letter. The name should be independent of the underlying data type and should refer to whatever the variable represents.
<i>RoutineName()</i>	Routines are in mixed uppercase and lowercase. (Good routine names are discussed in Section 5.2.)
<i>m_ClassVariable</i>	Member variables that are available to multiple routines within a class, but only within a class, are prefixed with an <i>m_</i> .
<i>g_GlobalVariable</i>	Global variables are prefixed with a <i>g_</i> .
<i>CONSTANT</i>	Named constants are in <i>ALL_CAPS</i> .
<i>MACRO</i>	Macros are in <i>ALL_CAPS</i> .
<i>Base_EnumeratedType</i>	Enumerated types are prefixed with a mnemonic for their base type stated in the singular—for example, <i>Color_Red</i> , <i>Color_Blue</i> .

704 **Table 11-4. Sample Naming Convention for C**

Entity	Description
<i>TypeName</i>	Type definitions use mixed upper and lower case with an initial capital letter
<i>GlobalRoutineName()</i>	Public routines are in mixed uppercase and lowercase.
<i>f_FileRoutineName()</i>	Routines that are private to a single module (file) are prefixed with an f-underscore.

706

707

708

<i>LocalVariable</i>	Local variables are in mixed uppercase and lowercase. The name should be independent of the underlying data type and should refer to whatever the variable represents.
<i>f_FileStaticVariable</i>	Module (file) variables are prefixed with an f- underscore.
<i>G_GLOBAL_GlobalVariable</i>	Global variables are prefixed with a <i>G_</i> and a mnemonic of the module (file) that defines the variable in all uppercase—for example, <i>SCREEN_Dimensions</i> .
<i>LOCAL_CONSTANT</i>	Named constants that are private to a single routine or module (file) are in all uppercase—for example, <i>ROWS_MAX</i> .
<i>G_GLOBALCONSTANT</i>	Global named constants are in all uppercase and are prefixed with <i>G_</i> and a mnemonic of the module (file) that defines the named constant in all uppercase—for example, <i>G_SCREEN_ROWS_MAX</i> .
<i>LOCALMACRO()</i>	Macro definitions that are private to a single routine or module (file) are in all uppercase.
<i>G_GLOBAL_MACRO()</i>	Global macro definitions are in all uppercase and are prefixed with <i>G_</i> and a mnemonic of the module (file) that defines the macro in all uppercase—for example, <i>G_SCREEN_LOCATION()</i> .

Because Visual Basic is not case sensitive, special rules apply for differentiating between type names and variable names.

Table 11-5. Sample Naming Convention for Visual Basic

Entity	Description
<i>C_ClassName</i>	Class names are in mixed upper and lower case with an initial capital letter and a <i>C_</i> prefix.
<i>T_TypeName</i>	Type definitions including enumerated types and typedefs used mixed upper and lower case with an initial capital letter and a <i>T_</i> prefix.
<i>T_EnumeratedTypes</i>	In addition to the rule above, enumerated types are always stated in the plural form.
<i>localVariable</i>	Local variables are in mixed uppercase and lowercase with an initial lower case letter. The name should be independent of the underlying data type and should refer to whatever the variable represents.
<i>RoutineName()</i>	Routines are in mixed uppercase and lowercase. (Good routine names are discussed in Section 5.2.)
<i>m_ClassVariable</i>	Member variables that are available to multiple routines within a class, but only within a class, are prefixed with an <i>m_</i> .
<i>g_GlobalVariable</i>	Global variables are prefixed with a <i>g_</i> .

<i>CONSTANT</i>	Named constants are in <i>ALL_CAPS</i> .
<i>Base_EnumeratedType</i>	Enumerated types are prefixed with a mnemonic for their base type stated in the singular—for example, <i>Color_Red</i> , <i>Color_Blue</i> .

709

710 **FURTHER READING** For
 711 further details on the
 712 Hungarian naming
 713 convention, see “The
 714 Hungarian Revolution”
 715 (Simonyi and Heller 1991).
 716

717
 718

719

720
 721
 722
 723

724
 725
 726
 727

728

11.5 Standardized Prefixes

Standardizing prefixes for common meanings provides a terse but consistent and readable approach to naming data. The best known scheme for standardizing prefixes is the Hungarian naming convention, which is a set of detailed guidelines for naming variables and routines (not Hungarians!) that was widely used at one time in Microsoft Windows programming. Although the Hungarian naming convention is no longer in widespread use, the basic idea of standardizing on terse, precise abbreviations continues to have value.

Standardized Prefixes are composed of two parts: the user-defined–data type (UDT) abbreviation and the semantic prefix.

User-Defined-Type (UDT) Abbreviation

The UDT abbreviation identifies the data type of the object or variable being named. UDT abbreviations might refer to entities such as windows, screen regions, and fonts. A UDT abbreviation generally doesn’t refer to any of the predefined data types offered by the programming language.

UDTs are described with short codes that you create for a specific program and then standardize on for use in that program. The codes are mnemonics such as *wn* for windows and *scr* for screen regions. Here’s a sample list of UDTs that you might use in a program for a word processor:

Table 11-6. Sample of UDTs for a Word Processor

UDT Abbreviation	Meaning
<i>ch</i>	Character (a character not in the C++ sense, but in the sense of the data type a word-processing program would use to represent a character in a document)
<i>doc</i>	Document
<i>pa</i>	Paragraph
<i>scr</i>	Screen region
<i>sel</i>	Selection
<i>wn</i>	Window

729 When you use UDTs, you also define programming-language data types that use
730 the same abbreviations as the UDTs. Thus, if you had the UDTs in the table
731 above, you'd see data declarations like these:

732 CH chCursorPosition;
733 SCR scrUserWorkspace;
734 DOC docActive
735 PA firstPaActiveDocument;
736 PA lastPaActiveDocument;
737 WN wnMain;

738 These examples are from a word processor. For use on your own projects, you
739 would create UDT abbreviations for the UDTs that are used most commonly
740 within your environment.

741 Semantic Prefix

742 Semantic prefixes go a step beyond the UDT and describe how the variable or
743 object is used. Unlike UDTs, which vary project to project, semantic prefixes are
744 somewhat standard across projects. Table 11-7 shows a list of standard semantic
745 prefixes.

746 **Table 11-7. Semantic Prefixes**

Semantic Prefix	Meaning
c	Count (as in the number of records, characters, and so on)
first	The first element that needs to be dealt with in an array. <i>first</i> is similar to <i>min</i> but relative to the current operation rather than to the array itself.
g	Global variable
i	Index into an array
last	The last element that needs to be dealt with in an array. <i>last</i> is the counterpart of <i>first</i> .
lim	The upper limit of elements that need to be dealt with in an array. <i>lim</i> is not a valid index. Like <i>last</i> , <i>lim</i> is used as a counterpart of <i>first</i> . Unlike <i>last</i> , <i>lim</i> represents a noninclusive upper bound on the array; <i>last</i> represents a final, legal element. Generally, <i>lim</i> equals <i>last</i> + 1.
m	Class-level variable
max	The absolute last element in an array or other kind of list. <i>max</i> refers to the array itself rather than to operations on the array.
min	The absolute first element in an array or other kind of list.
p	Pointer
747	Semantic prefixes are formatted in lowercase or mixed upper and lower case and are combined with the UDTs and with each other as needed. For example, the
748	first paragraph in a document would be named <i>pa</i> to show that it's a paragraph
749	and <i>first</i> to show that it's the first paragraph: <i>firstPa</i> . An index into the set of
750	

751 paragraphs would be named *iPa*; *cPa* is the count, or the number of paragraphs.
752 *firstPaActiveDocument* and *lastPaActiveDocument* are the first and last
753 paragraphs in the current active document.

754

755 **KEY POINT**

756 Standardized Prefixes give you all the general advantages of having a naming
757 convention as well as several other advantages. Because so many names are
standard, there are fewer names to remember in any single program or class.

758

759 Standardized Prefixes add precision to several areas of naming that tend to be
760 imprecise. The precise distinctions between *min*, *first*, *last*, and *max* are
particularly helpful.

761

762 Standardized Prefixes make names more compact. For example, you can use *cpa*
763 for the count of paragraphs rather than *totalParagraphs*. You can use *ipa* to
764 identify an index into an array of paragraphs rather than *indexParagraphs* or
paragraphsIndex.

765

766 Finally, standardized Prefixes allow you to check types accurately when you're
767 using abstract data types that your compiler can't necessarily check: *paReformat*
= *docReformat* is probably wrong because *pa* and *doc* are different UDTs.

768

769 The main pitfall with standardized prefixes is neglecting to give the variable a
770 meaningful name in addition to its prefix. If *ipa* unambiguously designates an
index into an array of paragraphs, it is tempting not to make the name more
771 descriptive, not to name it something more meaningful like *ipaActiveDocument*.
772 Thus, readability is not as good as it would be with a more descriptive name.

773

774 Ultimately, this complaint about standardized prefixes is not a pitfall as much as
775 a limitation. No technique is a silver bullet, and individual discipline and
776 judgment will always be needed with any technique. *ipa* is a better variable name
than *i*, which is at least a step in the right direction.

777

778

779 **KEY POINT**

780 The desire to use short variable names is in some ways a historical remnant of an
781 earlier age of computing. Older languages like assembler, generic Basic, and
782 Fortran limited variable names to two to eight characters and forced
783 programmers to create short names. Early computing was more closely linked to
784 mathematics, and it's use of terms like *i*, *j*, and *k* as the variables in summations
and other equations. In modern languages like C++, Java, and Visual Basic, you

785 can create names of virtually any length; you have almost no reason to shorten
786 meaningful names.

787 If circumstances do require you to create short names, note that some methods of
788 shortening names are better than others. You can create good short variable
789 names by eliminating needless words, using short synonyms, and using other
790 abbreviation techniques. You can use any of several abbreviation strategies. It's
791 a good idea to be familiar with multiple techniques for abbreviating because no
792 single technique works well in all cases.

793 General Abbreviation Guidelines

794 Here are several guidelines for creating abbreviations. Some of them contradict
795 others, so don't try to use them all at the same time.

- 796 • Use standard abbreviations (the ones in common use, which are listed in a
797 dictionary).
- 798 • Remove all nonleading vowels. (*computer* becomes *cptr*, and *screen*
799 becomes *scrn*. *apple* becomes *appl*, and *integer* becomes *intgr*.)
- 800 • Remove articles: *and*, *or*, *the*, and so on.
- 801 • Use the first letter or first few letters of each word.
- 802 • Truncate after the first, second, or third (whichever is appropriate) letter of
803 each word.
- 804 • Keep the first and last letters of each word.
- 805 • Use every significant word in the name, up to a maximum of three words.
- 806 • Remove useless suffixes—*ing*, *ed*, and so on.
- 807 • Keep the most noticeable sound in each syllable.
- 808 • Iterate through these techniques until you abbreviate each variable name to
809 between 8 to 20 characters, or the number of characters to which your
810 language limits variable names.

811 Phonetic Abbreviations

812 Some people advocate creating abbreviations based on the sound of the words
813 rather than their spelling. Thus *skating* becomes *sk8ing*, *highlight* becomes *hilite*,
814 *before* becomes *b4*, *execute* becomes *xqt*, and so on. This seems too much like
815 asking people to figure out personalized license plates to me, and I don't
816 recommend it. As an exercise, figure out what these names mean:

817 *ILV2SK8* *XMEQWK* *S2DTM8O* *NXTc* *TRMN8R*

Comments on Abbreviations

You can fall into several traps when creating abbreviations. Here are some rules for avoiding pitfalls:

Don't abbreviate by removing one character from a word

Typing one character is little extra work, and the one-character savings hardly justifies the loss in readability. It's like the calendars that have "Jun" and "Jul." You have to be in a big hurry to spell June as "Jun." With most one-letter deletions, it's hard to remember whether you removed the character. Either remove more than one character or spell out the word.

Abbreviate consistently

Always use the same abbreviation. For example, use *Num* everywhere or *No* everywhere, but don't use both. Similarly, don't abbreviate a word in some names and not in others. For instance, don't use the full word *Number* in some places and the abbreviation *Num* in others.

Create names that you can pronounce

Use *xPos* rather than *xPstn* and *needsComp* rather than *ndxCmptg*. Apply the telephone test—if you can't read your code to someone over the phone, rename your variables to be more distinctive (Kernighan and Plauger 1978).

Avoid combinations that result in mispronunciation

To refer to the end of *B*, favor *ENDB* over *BEND*. If you use a good separation technique, you won't need this guideline since *B-END*, *BEnd*, or *b_end* won't be mispronounced.

Use a thesaurus to resolve naming collisions

One problem in creating short names is naming collisions—names that abbreviate to the same thing. For example, if you're limited to three characters and you need to use *fired* and *full revenue disbursal* in the same area of a program, you might inadvertently abbreviate both to *frd*.

One easy way to avoid naming collisions is to use a different word with the same meaning, so a thesaurus is handy. In this example, *dismissed* might be substituted for *fired* and *complete revenue disbursal* might be substituted for *full revenue disbursal*. The three-letter abbreviations become *dsm* and *crd*, eliminating the naming collision.

Document extremely short names with translation tables in the code

In languages that allow only very short names, include a translation table to provide a reminder of the mnemonic content of the variables. Include the table as comments at the beginning of a block of code. Here's an example in Fortran:

853

854

855

856

857

858

859

860

861

862

863

864

865

866

867

868

869

870

871

872

873

874

875

876

877

878

879

880

881

882

883

884

885

886

887

888

889

890

891

892

Fortran Example of a Good Translation Table

```
C ****
C      Translation Table
C
C      Variable      Meaning
C      -----      -----
C      XPOS          x-Coordinate Position (in meters)
C      YPOS          Y-Coordinate Position (in meters)
C      NDSCMP        Needs Computing (=0 if no computation is needed;
C                           =1 if computation is needed)
C      PTGTTL        Point Grand Total
C      PTVLMAX       Point Value Maximum
C      PSCRMX        Possible Score Maximum
C ****
```

You might think that this technique is outdated, but as recently as mid-2003 I worked with a client that had hundreds of thousands of lines of code written in RPG that was subject to a 6-character-variable-name limitation. These issues still come up from time to time.

Document all abbreviations in a project-level “Standard Abbreviations” document

Abbreviations in code create two general risks:

- A reader of the code might not understand the abbreviation
- Other programmers might use multiple abbreviations to refer to the same word, which creates needless confusion

To address both these potential problems, you can create a “Standard Abbreviations” document that captures all the coding abbreviations used on your project. The document can be a word processor document or a spreadsheet. On a very large project, it could be a database. The document is checked into version control and checked out anytime anyone creates a new abbreviation in the code. Entries in the document should be sorted by the full word, not the abbreviation.

This might seem like a lot of overhead, but aside from a small amount of startup-overhead, it really just sets up a mechanism that helps the project use abbreviations effectively. It addresses the first of the two general risks described above by documenting all abbreviations in use. The fact that a programmer can't create a new abbreviation without the overhead of checking the Standard Abbreviations document out of version control, entering the abbreviation, and checking it back in *is a good thing*. It means that an abbreviation won't be created unless it is so common that it's worth the hassle of documenting it.

It addresses the second risk by reducing the likelihood that a programmer will create a redundant abbreviation. A programmer who wants to abbreviate

893 something will check out the abbreviations document and enter the new
894 abbreviation. If there is already an abbreviation for the word the programmer
895 wants to abbreviate, the programmer will notice that and will then use the
896 existing abbreviation instead of creating a new one.

897 The general issue illustrated by this guideline is the difference between write-
898 time convenience and read-time convenience. This approach clearly creates a
899 write-time *inconvenience*, but programmers over the lifetime of a system spend
900 far more time reading code than writing code. This approach increases read-time
901 convenience. By the time all the dust settles on a project, it might well also have
902 improved write-time convenience.

903 ***Remember that names matter more to the reader of the code than to the***
904 ***writer***

905 Read code of your own that you haven't seen for at least six months and notice
906 where you have to work to understand what the names mean. Resolve to change
907 the practices that cause confusion.

908 11.7 Kinds of Names to Avoid

909 Here are some kinds of variable names to avoid:

910 ***Avoid misleading names or abbreviations***

911 Be sure that a name is unambiguous. For example, *FALSE* is usually the opposite
912 of *TRUE* and would be a bad abbreviation for "Fig and Almond Season."

913 ***Avoid names with similar meanings***

914 If you can switch the names of two variables without hurting the program, you
915 need to rename both variables. For example, *input* and *inputValue*, *recordNum*
916 and *numRecords*, and *fileNumber* and *fileIndex* are so semantically similar that if
917 you use them in the same piece of code you'll easily confuse them and install
918 some subtle, hard-to-find errors.

919 **CROSS-REFERENCE** The
920 technical term for differences
921 like this is "psychological
922 distance." For details, see
923 "How "Psychological
924 Distance" Can Help" in
925 Section 23.4.

919 ***Avoid variables with different meanings but similar names***

920 If you have two variables with similar names and different meanings, try to
921 rename one of them or change your abbreviations. Avoid names like *clientRecs*
922 and *clientReps*. They're only one letter different from each other, and the letter is
923 hard to notice. Have at least two-letter differences between names, or put the
924 differences at the beginning or at the end. *clientRecords* and *clientReports* are
925 better than the original names.

926 ***Avoid names that sound similar, such as wrap and rap***

927 Homonyms get in the way when you try to discuss your code with others. One of
928 my pet peeves about Extreme Programming (Beck 2000) is its overly clever use

929 of the terms Goal Donor and Gold Owner, which are virtually indistinguishable
930 when spoken. You end up having conversations like this:

931 *I was just speaking with the Goal Donor—*

932 *Did you say “Gold Owner” or “Goal Donor?”*

933 *I said “Goal Donor.”*

934 *What?*

935 *GOAL - - - DONOR!*

936 *OK, Goal Donor. You don’t have to yell, Goll’ Darn it.*

937 *Did you say “Gold Donut?”*

938 Remember that the telephone test applies to similar sounding names just as it
939 does to oddly abbreviated names.

940 **Avoid numerals in names**

941 If the numerals in a name are really significant, use an array instead of separate
942 variables. If an array is inappropriate, numerals are even more inappropriate. For
943 example, avoid *file1* and *file2*, or *total1* and *total2*. You can almost always think
944 of a better way to differentiate between two variables than by tacking a 1 or a 2
945 onto the end of the name. I can’t say *never* use numerals, but you should be
946 desperate before you do.

947 **Avoid misspelled words in names**

948 It’s hard enough to remember how words are supposed to be spelled. To require
949 people to remember “correct” misspellings is simply too much to ask. For
950 example, misspelling *highlight* as *hilite* to save three characters makes it
951 devilishly difficult for a reader to remember how *highlight* was misspelled. Was
952 it *highlite*? *hilite*? *hilight*? *hilit*? *jai-a-lai-t*? Who knows?

953 **Avoid words that are commonly misspelled in English**

954 *Absense, accummulate, acsend, calender, concieve, defferred, definate,*
955 *independance, occassionally, prefered, receipt, superseed*, and many others are
956 common misspellings in English. Most English handbooks contain a list of
957 commonly misspelled words. Avoid using such words in your variable names.

958 **Don’t differentiate variable names solely by capitalization**

959 If you’re programming in a case-sensitive language such as C++, you may be
960 tempted to use *frd* for *fired*, *FRD* for *final review duty*, and *Frd* for *full revenue*
961 *disbursal*. Avoid this practice. Although the names are unique, the association of

962 each with a particular meaning is arbitrary and confusing. *Frd* could just as
963 easily be associated with *final review duty* and *FRD* with *full revenue disbursal*,
964 and no logical rule will help you or anyone else to remember which is which.

965 **Avoid multiple natural languages**

966 In multi-national projects, enforce use of a single natural language for all code
967 including class names, variable names, and so on. Reading another
968 programmer's code can be a challenge; reading another programmer's code in
969 Southeast Martian is impossible.

970 **Avoid the names of standard types, variables, and routines**

971 All programming-language guides contain lists of the language's reserved and
972 predefined names. Read the list occasionally to make sure you're not stepping on
973 the toes of the language you're using. For example, the following code fragment
974 is legal in PL/I, but you would be a certifiable idiot to use it:

975 **CODING HORROR**
976 if if = then then
977 then = else;
 else else = if;

978 **Don't use names that are totally unrelated to what the variables represent**

979 Sprinkling names such as *margaret* and *pookie* throughout your program
980 virtually guarantees that no one else will be able to understand it. Avoid your
981 boyfriend's name, wife's name, favorite beer's name, or other clever (aka silly)
982 names for variables, unless the program is really about your boyfriend, wife, or
983 favorite beer. Even then, you would be wise to recognize that each of these
984 might change, and that therefore the generic names *boyFriend*, *wife*, and
985 *favoriteBeer* are superior!

986 **Avoid names containing hard-to-read characters**

987 Be aware that some characters look so similar that it's hard to tell them apart. If
988 the only difference between two names is one of these characters, you might
989 have a hard time telling the names apart. For example, try to circle the name that
990 doesn't belong in each of the following sets:

<i>eyeChartl</i>	<i>eyeChartl</i>	<i>eyeChartl</i>
<i>TTLCONFUSION</i>	<i>TTLCONFUSION</i>	<i>TTLC0NFUSION</i>
<i>hard2Read</i>	<i>hardZRead</i>	<i>hard2Read</i>
<i>GRANDTOTAL</i>	<i>GRANDTOTAL</i>	<i>6RANDTOTAL</i>
<i>ttl5</i>	<i>ttlS</i>	<i>ttlS</i>

991 Pairs that are hard to distinguish include (1 and l), (1 and I), (. and ,), (0 and O),
992 (2 and Z), (; and :), (S and 5), and (G and 6).

993
994
995
996

997 **GROSS SORRY** For
considerations in using data,
see the checklist in Chapter
998 10, "General Issues in Using
Variables."

999

1000

1001

1002

1003

1004

1005

1006

1007

1008

1009

1010

1011

1012

1013

1014

1015

1016

1017

1018

1019

1020

1021

1022

1023

1024

1025

1026

1027

Do details like these really matter? Indeed! Gerald Weinberg reports that in the 1970s, a comma was used in a Fortran *FORMAT* statement where a period should have been used. The result was that scientists miscalculated a spacecraft's trajectory and lost a space probe—to the tune of \$1.6 billion (Weinberg 1983).

CHECKLIST: Naming Variables

General Naming Considerations

- Does the name fully and accurately describe what the variable represents?
- Does the name refer to the real-world problem rather than to the programming-language solution?
- Is the name long enough that you don't have to puzzle it out?
- Are computed-value qualifiers, if any, at the end of the name?
- Does the name use *Count* or *Index* instead of *Num*?

Naming Specific Kinds Of Data

- Are loop index names meaningful (something other than *i*, *j*, or *k* if the loop is more than one or two lines long or is nested)?
- Have all "temporary" variables been renamed to something more meaningful?
- Are boolean variables named so that their meanings when they're *True* are clear?
- Do enumerated-type names include a prefix or suffix that indicates the category—for example, *Color_* for *Color_Red*, *Color_Green*, *Color_Blue*, and so on?
- Are named constants named for the abstract entities they represent rather than the numbers they refer to?

Naming Conventions

- Does the convention distinguish among local, class, and global data?
- Does the convention distinguish among type names, named constants, enumerated types, and variables?
- Does the convention identify input-only parameters to routines in languages that don't enforce them?
- Is the convention as compatible as possible with standard conventions for the language?
- Are names formatted for readability?

Short Names

- Does the code use long names (unless it's necessary to use short ones)?

- 1028 Does the code avoid abbreviations that save only one character?
1029 Are all words abbreviated consistently?
1030 Are the names pronounceable?
1031 Are names that could be mispronounced avoided?
1032 Are short names documented in translation tables?

1033 **Common Naming Problems: Have You Avoided...**

- 1034 ...names that are misleading?
1035 ...names with similar meanings?
1036 ...names that are different by only one or two characters?
1037 ...names that sound similar?
1038 ...names that use numerals?
1039 ...names intentionally misspelled to make them shorter?
1040 ...names that are commonly misspelled in English?
1041 ...names that conflict with standard library-routine names or with predefined
1042 variable names?
1043 ...totally arbitrary names?
1044 ...hard-to-read characters?
1045
-

1046 **Key Points**

- 1047
 - Good variable names are a key element of program readability. Specific
1048 kinds of variables such as loop indexes and status variables require specific
1049 considerations.
 - Names should be as specific as possible. Names that are vague enough or
1050 general enough to be used for more than one purpose are usually bad names.
 - Naming conventions distinguish among local, class, and global data. They
1052 distinguish among type names, named constants, enumerated types, and
1053 variables.
 - Regardless of the kind of project you're working on, you should adopt a
1055 variable naming convention. The kind of convention you adopt depends on
1056 the size of your program and the number of people working on it.
 - Abbreviations are rarely needed with modern programming languages. If
1058 you do use abbreviations, keep track of abbreviations in a project dictionary
1059 or use the Standardized Prefixes approach.
1060

12

2 Fundamental Data Types

3 CC2E.COM/1278

4 Contents

- 5 12.1 Numbers in General
- 6 12.2 Integers
- 7 12.3 Floating-Point Numbers
- 8 12.4 Characters and Strings
- 9 12.5 Boolean Variables
- 10 12.6 Enumerated Types
- 11 12.7 Named Constants
- 12 12.8 Arrays
- 13 12.9 Creating Your Own Types

14 Related Topics

15 Naming data: Chapter 11

16 Unusual data types: Chapter 13

17 General issues in using variables: Chapter 10

18 Formatting data declarations: “Laying Out Data Declarations” in Section 31.5

19 Documenting variables: “Commenting Data Declarations” in Section 32.5

20 Creating classes: Chapter 6

21 THE FUNDAMENTAL DATA TYPES ARE the basic building blocks for all
22 other data types. This chapter contains tips for using integers, floating-point
23 numbers, characters and strings, boolean variables, enumerated types, named
24 constants, and arrays. The final section in this chapter describes how to create
your own types.

25 This chapter covers basic troubleshooting for the fundamental types of data. If
26 you’ve got your fundamental-data bases covered, skip to the end of the chapter,
27 review the checklist of problems to avoid, and move on to the discussion of
28 unusual data types in Chapter 13.

29

12.1 Numbers in General

30

Here are several guidelines for making your use of numbers less error prone.

31 **CROSS-REFERENCE** For
32 more details on using named
33 constants instead of magic
34 numbers, see Section 12.7,
35 “Named Constants,” later in
this chapter.

36

Avoiding magic numbers yields three advantages:

- Changes can be made more reliably. If you use named constants, you won’t overlook one of the *100*s, or change a *100* that refers to something else.
- Changes can be made more easily. When the maximum number of entries changes from *100* to *200*, if you’re using magic numbers you have to find all the *100*s and change them to *200*s. If you use *100+1* or *100-1* you’ll also have to find all the *101*s and *99*s and change them to *201*s and *199*s. If you’re using a named constant, you simply change the definition of the constant from *100* to *200* in one place.
- Your code is more readable. Sure, in the expression

```
for i = 0 to 99 do ...
```

you can guess that *99* refers to the maximum number of entries. But the expression

```
for i = 0 to MAX_ENTRIES-1 do ...
```

leaves no doubt. Even if you’re certain that a number will never change, you get a readability benefit if you use a named constant.

Use hard-coded 0s and 1s if you need to

The values *0* and *1* are used to increment, decrement, and start loops at the first element of an array. The *0* in

```
for i = 0 to CONSTANT do ...
```

is OK, and the *1* in

```
total = total + 1
```

is OK. A good rule of thumb is that the only literals that should occur in the body of a program are *0* and *1*. Any other literals should be replaced with something more descriptive.

61
62
63
64***Anticipate divide-by-zero errors***

Each time you use the division symbol (/ in most languages), think about whether it's possible for the denominator of the expression to be 0. If the possibility exists, write code to prevent a divide-by-zero error.

65
66
67***Make type conversions obvious***

Make sure that someone reading your code will be aware of it when a conversion between different data types occurs. In C++ you could say

68
69

```
y = x + (float) i
```

and in Visual Basic you could say

70
71
72
73

```
y = x + CSng( i )
```

This practice also helps to ensure that the conversion is the one you want to occur—different compilers do different conversions, so you're taking your chances otherwise.

74 **CROSS-REFERENCE** For
75 a variation on this example,
see "Avoid equality
76 comparisons" in Section
77 12.3.***Avoid mixed-type comparisons***

If *x* is a floating-point number and *i* is an integer, the test

```
if ( i == x ) ...
```

is almost guaranteed not to work. By the time the compiler figures out which type it wants to use for the comparison, converts one of the types to the other, does a bunch of rounding, and determines the answer, you'll be lucky if your program runs at all. Do the conversion manually so that the compiler can compare two numbers of the same type and you know exactly what's being compared.

83 **KEY POINT**84
85
86
87
88
89***Heed your compiler's warnings***

Many modern compilers tell you when you have different numeric types in the same expression. Pay attention! Every programmer has been asked at one time or another to help someone track down a pesky error, only to find that the compiler had warned about the error all along. Top programmers fix their code to eliminate all compiler warnings. It's easier to let the compiler do the work than to do it yourself.

90

12.2 Integers

91

Here are a few considerations to bear in mind when using integers:

92
93
94
95***Check for integer division***

When you're using integers, 7/10 does not equal 0.7. It usually equals 0. This applies equally to intermediate results. In the real world $10 * (7/10) = (10*7) / 10 = 7$. Not so in the world of integer arithmetic. $10 * (7/10)$ equals 0 because the

96 integer division ($7/10$) equals 0. The easiest way to remedy this problem is to
97 reorder the expression so that the divisions are done last: $(10^7) / 10$.

98 ***Check for integer overflow***

99 When doing integer multiplication or addition, you need to be aware of the
100 largest possible integer. The largest possible unsigned integer is often 65,535, or
101 $2^{32}-1$. The problem comes up when you multiply two numbers that produce a
102 number bigger than the maximum integer. For example, if you multiply $250 * 300$, the right answer is 75,000. But if the maximum integer is 65,535, the
103 answer you'll get is probably 9464 because of integer overflow ($75,000 - 65,536 = 9464$). Here are the ranges of common integer types:

Integer Type	Range
Signed 8-bit	-128 through 127
Unsigned 8-bit	0 through 255
Signed 16-bit	-32,768 through 32,767
Unsigned 16-bit	0 through 65,535
Signed 32-bit	-2,147,483,648 through 2,147,483,647
Unsigned 32-bit	0 through 4,294,967,295
Signed 64-bit	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807
Unsigned 64-bit	0 through 18,446,744,073,709,551,615

106 The easiest way to prevent integer overflow is to think through each of the terms
107 in your arithmetic expression and try to imagine the largest value each can
108 assume. For example, if in the integer expression $m = j * k$, the largest expected
109 value for j is 200 and the largest expected value for k is 25, the largest value you
110 can expect for m is $200 * 25 = 5,000$. This is OK on a 32-bit machine since the
111 largest integer is 2,147,483,647. On the other hand, if the largest expected value
112 for j is 200,000 and the largest expected value for k is 100,000, the largest value
113 you can expect for m is $200,000 * 100,000 = 20,000,000,000$. This is not OK
114 since 20,000,000,000 is larger than 2,147,483,647. In this case, you would have
115 to use 64-bit integers or floating-point numbers to accommodate the largest
116 expected value of m .

117 Also consider future extensions to the program. If m will never be bigger than
118 5,000, that's great. But if you expect m to grow steadily for several years, take
119 that into account.

120 ***Check for overflow in intermediate results***

121 The number at the end of the equation isn't the only number you have to worry
122 about. Suppose you have the following code:

123
124
125
126
127
128
129
130
131
132

Java Example of Overflow of Intermediate Results

```
int termA = 1000000;
int termB = 1000000;
int product = termA * termB / 1000000;
System.out.println( "(" + termA + " * " + termB + " ) / 1000000 = " + product );
```

If you think the *Product* assignment is the same as $(100,000 * 100,000) / 100,000$, you might expect to get the answer *100,000*. But the code has to compute the intermediate result of $100,000 * 100,000$ before it can divide by the final *100,000*, and that means it needs a number as big as *1,000,000,000,000*. Guess what?

Here's the result:

133 $(1000000 * 1000000) / 1000000 = -727$

134 If your integers go to only 2,147,483,647, the intermediate result is too large for
135 the integer data type. In this case, the intermediate result that should be
136 *1,000,000,000,000* is 727,379,968, so when you divide by *100,000*, you get *-727*,
137 rather than *100,000*.

138 You can handle overflow in intermediate results the same way you handle
139 integer overflow, by switching to a long-integer or floating-point type.

140

141 KEY POINT

142
143
144
145
146
147

12.3 Floating-Point Numbers

The main consideration in using floating-point numbers is that many fractional decimal numbers can't be represented accurately using the 1s and 0s available on a digital computer. Nonterminating decimals like $1/3$ or $1/7$ can usually be represented to only 7 or 15 digits of accuracy. In my version of Visual Basic, a 32 bit floating-point representation of $1/3$ equals 0.33333330. It's accurate to 7 digits. This is accurate enough for most purposes, but inaccurate enough to trick you sometimes.

148

Here are a few specific guidelines for using floating-point numbers:

149
150
151
152
153
154

Avoid additions and subtractions on numbers that have greatly different magnitudes

With a 32-bit floating-point variable, $1,000,000.00 + 0.1$ probably produces an answer of $1,000,000.00$ because 32 bits don't give you enough significant digits to encompass the range between $1,000,000$ and 0.1 . Likewise, $5,000,000.02 - 5,000,000.01$ is probably 0.0 .

155 **CROSS-REFERENCE** For
156 algorithms books that
157 describe ways to solve these
158 problems, see “Additional
159 Resources on Data Types” in
160 Section 10.1.

161 *1 is equal to 2 for
162 sufficiently large values
163 of 1.*

164 —Anonymous

165
166

167

168 The variable nominal is a 64-
169 bit real.
170

171
172 sum is computed as 10×0.1 . It
173 should be 1.0.

174
175 Here's the bad comparison.

176
177
178
179
180
181

182
183

184 0.1
185 0.2
186 0.30000000000000004
187 0.4
188 0.5
189 0.6
190 0.7
191 0.7999999999999999
192 0.8999999999999999
193 0.9999999999999999

194 Thus, it's a good idea to find an alternative to using an equality comparison for
195 floating point numbers. One effective approach is to determine a range of
196 accuracy that is acceptable and then use a boolean function to determine whether

Solutions? If you have to add a sequence of numbers that contains huge differences like this, sort the numbers first, and then add them starting with the smallest values. Likewise, if you need to sum an infinite series, start with the smallest term—essentially, sum the terms backwards. This doesn't eliminate round-off problems, but it minimizes them. Many algorithms books have suggestions for dealing with cases like this.

Avoid equality comparisons

Floating-point numbers that should be equal are not always equal. The main problem is that two different paths to the same number don't always lead to the same number. For example, 0.1 added 10 times rarely equals 1.0. The first example on the next page shows two variables, *nominal* and *sum*, that should be equal but aren't.

Java Example of a Bad Comparison of Floating-Point Numbers

```
double nominal = 1.0;
double sum = 0.0;

for ( int i = 0; i < 10; i++ ) {
    sum += 0.1;
}

if ( nominal == sum ) {
    System.out.println( "Numbers are the same." );
}
else {
    System.out.println( "Numbers are different." );
}
```

As you can probably guess, the output from this program is

Numbers are different.

The line-by-line values of *sum* in the *for* loop look like this:

197
198
199

200 **CROSS-REFERENCE** This
example is proof of the
201 maxim that there's an
exception to every rule.
202 Variables in this realistic
203 example have digits in their
204 names. For the rule *against*
205 using digits in variable
206 names, see Section 11.7,
207 "Kinds of Names to Avoid."
208
209
210
211
212

213 **if (Equals(Nominal, Sum)) ...**
214

215 **Numbers are the same.**
216 Depending on the demands of your application, it might be inappropriate to use a
217 hard-coded value for *AcceptableDelta*. You might need to compute
218 *AcceptableDelta* based on the size of the two numbers being compared.

219 ***Anticipate rounding errors***
220 Rounding-error problems are no different from the problem of numbers with
221 greatly different magnitudes. The same issue is involved, and many of the same
222 techniques help to solve rounding problems. In addition, here are common
223 specific solutions to rounding problems:

224 First, change to a variable type that has greater precision. If you're using single-
225 precision floating point, change to double-precision floating point, and so on.

226 Second, change to binary coded decimal (BCD) variables. The BCD scheme is
227 typically slower and takes up more storage space but prevents many rounding
228 errors. This is particularly valuable if the variables you're using represent dollars
229 and cents or other quantities that must balance precisely.

230 Third, change from floating-point to integer variables. This is a roll-your-own
231 approach to BCD variables. You will probably have to use 64-bit integers to get
232 the precision you want. This technique requires you to keep track of the
233 fractional part of your numbers yourself. Suppose you were originally keeping
234 track of dollars using floating point with cents expressed as fractional parts of

235 dollars. This is a normal way to handle dollars and cents. When you switch to
236 integers, you have to keep track of cents using integers and of dollars using
237 multiples of 100 cents. In other words, you multiply dollars by 100 and keep the
238 cents in the 0-to-99 range of the variable. This might seem absurd at first glance,
239 but it's an effective solution in terms of both speed and accuracy. You can make
240 these manipulations easier by creating a *DollarsAndCents* class that hides the
241 integer representation and supports the necessary numeric operations.

242 **Check language and library support for specific data types**
243 Some languages including Visual Basic have data types such as *Currency* that
244 specifically support data that is sensitive to rounding errors. If your language has
245 a built-in data type that provides such functionality, use it!

246 12.4 Characters and Strings

247 Here are some tips for using strings. The first applies to strings in all languages.

248 **CROSS-REFERENCE** Issu
249 es for using magic characters
250 and strings are similar to
251 those for magic numbers
252 discussed in Section 12.1,
253 “Numbers in General.”

- 254
- 255
- 256
- 257
- 258
- 259
- 260
- 261
- 262
- 263
- 264
- 265
- 266
- 267
- 268
- 269
- 270
- Avoid magic characters and strings**
- Magic characters are literal characters (such as <,\$QS>A<,\$QS>) and magic strings are literal strings (such as <,\$QD>*Gigamatic Accounting Program*<,\$QD>) that appear throughout a program. If you program in a language that supports the use of named constants, use them instead. Otherwise, use global variables. Several reasons for avoiding literal strings follow.
- For commonly occurring strings like the name of your program, command names, report titles, and so on, you might at some point need to change the string's contents. For example, “*Gigamatic Accounting Program*” might change to “*New and Improved! Gigamatic Accounting Program*” for a later version.
 - International markets are becoming increasingly important, and it's easier to translate strings that are grouped in a string resource file than it is to translate to them *in situ* throughout a program.
 - String literals tend to take up a lot of space. They're used for menus, messages, help screens, entry forms, and so on. If you have too many, they grow beyond control and cause memory problems. String space isn't a concern in many environments, but in embedded systems programming and other applications in which storage space is at a premium, solutions to string-space problems are easier to implement if the strings are relatively independent of the source code.
 - Character and string literals are cryptic. Comments or named constants clarify your intentions. In the example below, the meaning of

271 `<QS>\027<QS>` isn't clear. The use of the *ESCAPE* constant makes the
272 meaning more obvious.

C++ Examples of Comparisons Using Strings

273 *Bad!* `if (input_char == '\027') ...`
274 *Better!* `if (input_char == ESCAPE) ...`

Watch for off-by-one errors

276 Because substrings can be indexed much as arrays are, watch for off-by-one
277 errors that read or write past the end of a string.
278

Know how your language and environment support Unicode

280 In some languages such as Java, all strings are Unicode. In others such as C and
281 C++, handling Unicode strings requires its own set of functions. Conversion
282 between Unicode and other character sets is often required for communication
283 with standard and third-party libraries. If some strings won't be in Unicode (for
284 example, in C or C++), decide early on whether to use the Unicode character set
285 at all. If you decide to use Unicode strings, decide where and when to use them.

Decide on an internationalization/localization strategy early in the lifetime of a program

286 Issues related to internationalization and localization are major issues. Key
287 considerations are deciding whether to store all strings in an external resource
288 and whether to create separate builds for each language or to determine the
289 specific language at run-time.
290
291

If you know you only need to support a single alphabetic language, consider using an ISO 8859 character set

292 CC2E.COM/1292 For applications that need to support only a single alphabetic language such as
293 English, and that don't need to support multiple languages or an ideographic
294 language such as written Chinese, the ISO 8859 extended-ASCII-type standard
295 makes a good alternative to Unicode.
296
297

If you need to support multiple languages, use Unicode

298 CC2E.COM/1293 Unicode provides more comprehensive support for international character sets
299 than ISO 8859 or other standards.
300

Decide on a consistent conversion strategy among string types

301 CC2E.COM/1294 If you use multiple string types, one common approach that helps keep the string
302 types distinct is to keep all strings in a single format within the program, and
303 convert the strings to other formats as close as possible to input and output
304 operations.
305

306

307

308

309

310

311

312

313

314

315

316

317

318

319

320

321

322

323

324

325

326

327

328

329

330

331

332

333

334

335

336

337

338

339

340 The string is declared to be of
341 length NAME_LENGTH +1.

342

343

344

Strings in C

C++'s standard template library string class has eliminated most of the traditional problems with strings in C. For those programmers working directly with C strings, here are some ways to avoid common pitfalls.

Be aware of the difference between string pointers and character arrays

The problem with string pointers and character arrays arises because of the way C handles strings. Be alert to the difference between them in two ways:

- Be suspicious of any expression containing a string that involves an equal sign. String operations in C are nearly always done with *strcmp()*, *strcpy()*, *strlen()*, and related routines. Equal signs often imply some kind of pointer error. In C, assignments do not copy string literals to a string variable.

Suppose you have a statement like

```
StringPtr = "Some Text String";
```

In this case, *<\$QD>Some Text String<\$QD>* is a pointer to a literal text string and the assignment merely sets the pointer *StringPtr* to point to the text string. The assignment does not copy the contents to *StringPtr*.

- Use a naming convention to indicate whether the variables are arrays of characters or pointers to strings. One common convention is to use *ps* as a prefix to indicate a *pointer* to a *string* and *ach* as a prefix for an *array of characters*. Although they're not always wrong, you should regard expressions involving both *ps* and *ach* prefixes with suspicion.

Declare C-style strings to have length CONSTANT+1

In C and C++, off-by-one errors with C-style strings are easy to make because it's easy to forget that a string of length *n* requires *n + 1* bytes of storage and to forget to leave room for the null terminator (the byte set to 0 at the end of the string). An easy and effective way to avoid such problems is to use named constants to declare all strings. A key in this approach is that you use the named constant the same way every time. Declare the string to be length *CONSTANT+1*, and then use *CONSTANT* to refer to the length of a string in the rest of the code. Here's an example:

C Example of Good String Declarations

```
/* Declare the string to have length of "constant+1".  
   Every other place in the program, "constant" rather  
   than "constant+1" is used. */  
  
char string[ NAME_LENGTH + 1 ] = { 0 }; /* string of length NAME_LENGTH */  
  
...  
/* Example 1: Set the string to all 'A's using the constant,  
   NAME_LENGTH, as the number of 'A's that can be copied.
```

345
346 Operations on the string
347 NAME_LENGTH here...
348
349
350
351 ...and here.

359
360
361
362

363 **CROSS-REFERENCE** For
364 more details on initializing
365 data, see Section 10.3,
366 “Guidelines for Initializing
367 Variables.”

368
369

370
371
372
373
374
375

376 **CROSS-REFERENCE** For
377 more discussion of arrays,
378 read Section 12.8, “Arrays,”
379 later in this chapter.

380
381
382
383

```
Note that NAME_LENGTH rather than NAME_LENGTH + 1 is used. */  
for ( i = 0; i < NAME_LENGTH; i++ )  
    string[ i ] = 'A';  
...  
  
/* Example 2: Copy another string into the first string using  
the constant as the maximum length that can be copied. */  
strncpy( string, some_other_string, NAME_LENGTH );
```

If you don’t have a convention to handle this, you’ll sometimes declare the string to be of length *NAME_LENGTH* and have operations on it with *NAME_LENGTH-1*; at other times you’ll declare the string to be of length *NAME_LENGTH+1* and have operations on it work with length *NAME_LENGTH*. Every time you use a string, you’ll have to remember which way you declared it.

When you use strings the same way every time, you don’t have to remember how you dealt with each string individually and you eliminate mistakes caused by forgetting the specifics of an individual string. Having a convention minimizes mental overload and programming errors.

Initialize strings to null to avoid endless strings

C determines the end of a string by finding a null terminator, a byte set to 0 at the end of the string. No matter how long you think the string is, C doesn’t find the end of the string until it finds a 0 byte. If you forget to put a null at the end of the string, your string operations might not act the way you expect them to.

You can avoid endless strings in two ways. First, initialize arrays of characters to 0 when you declare them, as shown below:

C Example of a Good Declaration of a Character Array

```
char EventName[ MAX_NAME_LENGTH + 1 ] = { 0 };
```

Second, when you allocate strings dynamically, initialize them to 0 by using *calloc()* instead of *malloc()*. *calloc()* allocates memory and initializes it to 0. *malloc()* allocates memory without initializing it so you get potluck when you use memory allocated by *malloc()*.

Use arrays of characters instead of pointers in C

If memory isn’t a constraint—and often it is not—declare all your string variables as arrays of characters. This helps to avoid pointer problems, and the compiler will give you more warnings when you do something wrong.

Use strncpy() instead of strcpy() to avoid endless strings

String routines in C come in safe versions and dangerous versions. The more dangerous routines such as *strcpy()* and *strcmp()* keep going until they run into a NULL terminator. Their safer companions, *strncpy()* and *strncmp()*, take a

384
385

parameter for maximum length, so that even if the strings go on forever, your function calls won't.

386

387
388

389 **CROSS-REFERENCE** For details on using comments to document your program, see Chapter 32, "Self-Documenting Code."

393

394 **CROSS-REFERENCE** For an example of using a boolean function to document 395 your program, see "Making 396 Complicated Expressions 397 Simple" in Section 19.1.

399

400
401

402

403

404

405

406

407

408

409

410

411

412

413

414

415

416

417

418 **CODING HORROR**

12.5 Boolean Variables

It's hard to misuse logical or boolean variables, and using them thoughtfully makes your program cleaner.

Use boolean variables to document your program

Instead of merely testing a boolean expression, you can assign the expression to a variable that makes the implication of the test unmistakable. For example, in the fragment below, it's not clear whether the purpose of the *if* test is to check for completion, for an error condition, or for something else:

Java Example of Boolean Test in Which the Purpose Is Unclear

```
if ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex ) ||  
    ( elementIndex == lastElementIndex )  
) {  
    ...  
}
```

In the next fragment, the use of boolean variables makes the purpose of the *if* test clearer:

Java Example of Boolean Test in Which the Purpose Is Clear

```
finished = ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex ) );  
repeatedEntry = ( elementIndex == lastElementIndex );  
if ( finished || repeatedEntry ) {  
    ...  
}
```

Use boolean variables to simplify complicated tests

Often when you have to code a complicated test, it takes several tries to get it right. When you later try to modify the test, it can be hard to understand what the test was doing in the first place. Logical variables can simplify the test. In the example above, the program is really testing for two conditions: whether the routine is finished and whether it's working on a repeated entry. By creating the boolean variables *finished* and *repeatedEntry*, you make the *if* test simpler—easier to read, less error prone, and easier to modify.

Here's another example of a complicated test:

Visual Basic Example of a Complicated Test

```
If ( ( document.AtEndOfStream() ) And ( Not inputError ) ) And _
```

```
419   ( ( MIN_LINES <= LineCount ) And ( LineCount <= MAX_LINES ) ) And _  
420   ( Not ErrorProcessing() ) Then  
421     ' do something or other  
422     ...  
423 End If
```

The test in the example is fairly complicated but not uncommonly so. It places a heavy mental burden on the reader. My guess is that you won't even try to understand the *if* test but will look at it and say, "I'll figure it out later if I really need to." Pay attention to that thought because that's exactly the same thing other people do when they read your code and it contains tests like this.

429 Here's a rewrite of the code with boolean variables added to simplify the test:

430 Visual Basic Example of a Simplified Test

```
431 allDataRead = ( document.AtEndOfStream() ) And ( Not inputError )  
432 legalLineCount = ( MIN_LINES <= LineCount ) And ( LineCount <= MAX_LINES )  
433 Here's the simple test.  
434 If ( allDataRead ) And ( legalLineCount ) And ( Not ErrorProcessing() ) Then  
435   ' do something or other  
436   ...  
437 End If
```

438 This second version is simpler. My guess is that you'll read the boolean expression in the *if* test without any difficulty.

439 Create your own boolean type, if necessary

440 Some languages, such as C++, Java, and Visual Basic have a predefined boolean type. Others, such as C, do not. In languages such as C, you can define your own boolean type. In C, you'd do it this way:

443 C Example of Defining the BOOLEAN Type

```
444 typedef int BOOLEAN; // define the boolean type
```

445 Declaring variables to be *BOOLEAN* rather than *int* makes their intended use
446 more obvious and makes your program a little more self-documenting.

447 12.6 Enumerated Types

448 An enumerated type is a type of data that allows each member of a class of
449 objects to be described in English. Enumerated types are available in C++, and
450 Visual Basic and are generally used when you know all the possible values of a
451 variable and want to express them in words. Here are several examples of
452 enumerated types in Visual Basic:

453 Visual Basic Examples of Enumerated Types

```
454 Public Enum Color
```

```
455     Color_Red  
456     Color_Green  
457     Color_Blue  
458 End Enum  
459  
460 Public Enum Country  
461     Country_China  
462     Country_England  
463     Country_France  
464     Country_Germany  
465     Country_India  
466     Country_Japan  
467     Country_Usa  
468 End Enum  
469  
470 Public Enum Output  
471     Output_Screen  
472     Output_Printer  
473     Output_File  
474 End Enum
```

475 Enumerated types are a powerful alternative to shopworn schemes in which you
476 explicitly say, “1 stands for red, 2 stands for green, 3 stands for blue,...” This
477 ability suggests several guidelines for using enumerated types.

478 ***Use enumerated types for readability***

479 Instead of writing statements like

```
480     if chosenColor = 1  
481 you can write more readable expressions like
```

```
482     if chosenColor = Color_Red
```

483 Anytime you see a numeric literal, ask whether it makes sense to replace it with
484 an enumerated type.

485 ***Use enumerated types for reliability***

486 With a few languages (Ada in particular), an enumerated type lets the compiler
487 perform more thorough type checking than it can with integer values and
488 constants. With named constants, the compiler has no way of knowing that the
489 only legal values are *Color_Red*, *Color_Green*, and *Color_Blue*. The compiler
490 won’t object to statements like *color* = *Country_England* or *country* =
491 *Output_Printer*. If you use an enumerated type, declaring a variable as *Color*, the
492 compiler will allow the variable to be assigned only the values *Color_Red*,
493 *Color_Green*, and *Color_Blue*.

494 ***Use enumerated types for modifiability***
495 Enumerated types make your code easy to modify. If you discover a flaw in your
496 “1 stands for red, 2 stands for green, 3 stands for blue” scheme, you have to go
497 through your code and change all the 1s, 2s, 3s, and so on. If you use an
498 enumerated type, you can continue adding elements to the list just by putting
499 them into the type definition and recompiling.

500 ***Use enumerated types as an alternative to boolean variables***
501 Often, a boolean variable isn’t rich enough to express the meanings it needs to.
502 For example, suppose you have a routine return *True* if it has successfully
503 performed its task and *False* otherwise. Later you might find that you really have
504 two kinds of *False*. The first kind means that the task failed, and the effects are
505 limited to the routine itself; the second kind means that the task failed, and
506 caused a fatal error that will need to be propagated to the rest of the program. In
507 this case, an enumerated type with the values *Status_Success*, *Status_Warning*,
508 and *Status_FatalError* would be more useful than a boolean with the values *True*
509 and *False*. This scheme can easily be expanded to handle additional distinctions
510 in the kinds of success or failure.

511 ***Check for invalid values***
512 When you test an enumerated type in an *if* or *case* statement, check for invalid
513 values. Use the *else* clause in a *case* statement to trap invalid values:

514 **Good Visual Basic Example of Checking for Invalid Values in an** 515 **Enumerated Type**

```
516                   Select Case screenColor
517                   Case Color_Red
518                   ...
519                   Case Color_Blue
520                   ...
521                   Case Color_Green
522                   ...
523                   Case Else
524                    DisplayInternalError( False, "Internal Error 752: Invalid color." )
525                   End Select
```

526 ***Define the first and last entries of an enumeration for use as loop limits***
527 Defining the first and last elements in an enumeration to be *Color_First*,
528 *Color_Last*, *Country_First*, *Country_Last*, and so on allows you to write a loop
529 that loops through the elements of an enumeration. You set up the enumerated
530 type using explicit values, as shown below:

531 **Visual Basic Example of Setting *First* and *Last* Values in an Enumerated**
532 **Type**

```
533       Public Enum Country  
534           Country_First = 0  
535           Country_China = 0  
536           Country_England = 1  
537           Country_France = 2  
538           Country_Germany = 3  
539           Country_India = 4  
540           Country_Japan = 5  
541           Country_Usa = 6  
542           Country_Last = 6  
543       End Enum
```

544 Now the *Country_First* and *Country_Last* values can be used as loop limits, as
545 shown below:

546 **Good Visual Basic Example of Looping Through Elements in an**
547 **Enumeration**

```
548       ' compute currency conversions from US currency to target currency  
549       Dim usaCurrencyConversionRate( Country_Last ) As Single  
550       Dim iCountry As Country  
551       For iCountry = Country_First To Country_Last  
552           usaCurrencyConversionRate( iCountry ) = ConversionRate( Country_Usa, iCountry )  
553       Next
```

554 ***Reserve the first entry in the enumerated type as invalid***

555 When you declare an enumerated type, reserve the first value as an invalid value.
556 Examples of this were shown earlier in the Visual Basic declarations of *Color*,
557 *Country*, and *Output* types. Many compilers assign the first element in an
558 enumerated type to the value *0*. Declaring the element that's mapped to *0* to be
559 invalid helps to catch variables that were not properly initialized since they are
560 more likely to be *0* than any other invalid value.

561 Here is how the *Country* declaration would look with that approach:

562 **Visual Basic Example of Declaring the First Value in an Enumeration to**
563 **be Invalid**

```
564       Public Enum Country  
565           Country_InvalidFirst = 0  
566           Country_First = 1  
567           Country_China = 1  
568           Country_England = 2  
569           Country_France = 3  
570           Country_Germany = 4
```

```
571     Country_India = 5  
572     Country_Japan = 6  
573     Country_Usa = 7  
574     Country_Last = 7  
575 End Enum
```

Define precisely how First and Last elements are to be used in the project coding standard, and use them consistently

Using *InvalidFirst*, *First*, and *Last* elements in enumerations can make array declarations and loops more readable. But it has the potential to create confusion about whether the valid entries in the enumeration begin at *0* or *1* and whether the first and last elements of the enumeration are valid. If this technique is used, the project's coding standard should require that *InvalidFirst*, *First*, and *Last* elements be used consistently in all enumerations to reduce errors.

Beware of pitfalls of assigning explicit values to elements of an enumeration

Some languages allow you to assign specific values to elements within an enumeration, as shown in the C++ example below:

C++ Example of Explicitly Assigning Values to an Enumeration

```
589 enum Color {  
590     Color_InvalidFirst = 0,  
591     Color_Red = 1,  
592     Color_Green = 2,  
593     Color_Blue = 4,  
594     Color_InvalidLast = 8  
595 };
```

In this C++ example, if you declared a loop index of type Color and attempted to loop through *Colors*, you would loop through the invalid values of 3, 5, 6, and 7 as well as the valid values of 1, 2, and 4.

If Your Language Doesn't Have Enumerated Types

If your language doesn't have enumerated types, you can simulate them with global variables or classes. For example, here are declarations you could use in Java:

603 **CROSS-REFERENCE** At
the time I'm writing this,
604 Java does not support
605 enumerated types. By the
606 time you read this, it
607 probably will. This is a good
608 example of the "rolling wave
609 of technology" discussed in
610 Section 4.3, "Your Location
611 on the Technology Wave."

612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634

635
636
637

638

Java Example of Simulating Enumerated Types

```
// set up Color enumerated type
class Color {
    private Color() {}
    public static final Color Red = new Color();
    public static final Color Green = new Color();
    public static final Color Blue = new Color();
}

// set up Country enumerated type
class Country {
    private Country() {}
    public static final Country China = new Country();
    public static final Country England = new Country();
    public static final Country France = new Country();
    public static final Country Germany = new Country();
    public static final Country India = new Country();
    public static final Country Japan = new Country();
}

// set up Output enumerated type
class Output {
    private Output() {}
    public static final Output Screen = new Output();
    public static final Output Printer = new Output();
    public static final Output File = new Output();
}
```

These enumerated types make your program more readable because you can use the public class members such as *Color.Red* and *Country.England* instead of named constants. This particular method of creating enumerated types is also typesafe; because each type is declared as a class, the compiler will check for invalid assignments such as *Output output = Country.England* (Bloch 2001).

In languages that don't support classes, the same basic effect could be achieved through disciplined use of global variables for each of the elements of the enumeration.

12.7 Named Constants

A named constant is like a variable except that you can't change the constant's value once you've assigned it. Named constants enable you to refer to fixed quantities such as the maximum number of employees by a name rather than a number—*MaximumEmployees* rather than *1000*, for instance.

643
644
645
646
647
648
649
650

Using a named constant is a way of “parameterizing” your program—putting an aspect of your program that might change into a parameter that you can change in one place rather than having to make changes throughout the program. If you have ever declared an array to be as big as you think it will ever need to be and then run out of space because it wasn’t big enough, you can appreciate the value of named constants. When an array size changes, you change only the definition of the constant you used to declare the array. This “single-point control” goes a long way toward making software truly “soft”—easy to work with and change.

651
652
653
654
655

Use named constants in data declarations
Using named constants helps program readability and maintainability in data declarations and in statements that need to know the size of the data they are working with. In the example below, you use *PhoneLength_c* to describe the length of employee phone numbers rather than the literal 7.

656
657

Good Visual Basic Example of Using a Named Constant in a Data Declaration

658
659 LOCAL_NUMBER_LENGTH
660 is declared as a constant
661 here.
662
663 It's used here.
664
665
666
667 It's used here too.
668
669
670
671
672
673

```
Const AREA_CODE_LENGTH = 3
Const LOCAL_NUMBER_LENGTH = 7
...
Type PHONE_NUMBER
    areaCode( AREA_CODE_LENGTH ) As String
    localNumber( LOCAL_NUMBER_LENGTH ) As String
End Type
...
' make sure all characters in phone number are digits
For iDigit = 1 To LOCAL_NUMBER_LENGTH
    If ( phoneNumber.localNumber( iDigit ) < "0" ) Or _
        ( "9" < phoneNumber.localNumber( iDigit ) ) Then
        ' do some error processing
    ...

```

This is a simple example, but you can probably imagine a program in which the information about the phone-number length is needed in many places.

674
675
676
677
678

At the time you create the program, the employees all live in one country, so you need only seven digits for their phone numbers. As the company expands and branches are established in different countries, you’ll need longer phone numbers. If you have parameterized, you can make the change in only one place: in the definition of the named constant *LOCAL_NUMBER_LENGTH*.

679 **FURTHER READING** For
680 more details on the value of
681 single-point control, see
682 pages 57-60 of *Software
Conflict* (Glass 1991).

As you might expect, the use of named constants has been shown to greatly aid program maintenance. As a general rule, any technique that centralizes control over things that might change is a good technique for reducing maintenance efforts (Glass 1991).

683 **Avoid literals, even “safe” ones**

684 In the loop below, what do you think the *i* represents?

685 **Visual Basic Example of Unclear Code**

```
686           For i = 1 To 12  
687              profit(i) = revenue(i) - expense(i)  
688           Next
```

689 Because of the specific nature of the code, it appears that the code is probably
690 looping through the 12 months in a year. But are you *sure*? Would you bet your
691 Monty Python collection on it?

692 In this case, you don’t need to use a named constant to support future flexibility:
693 it’s not very likely that the number of months in a year will change anytime
694 soon. But if the way the code is written leaves any shadow of a doubt about its
695 purpose, clarify it with a well-named constant, as shown below.

696 **Visual Basic Example of Clearer Code**

```
697           For i = 1 To NUM_MONTHS_IN_YEAR  
698              profit(i) = revenue(i) - expense(i)  
699           Next
```

700 This is better, but, to complete the example, the loop index should also be named
701 something more informative.

702 **Visual Basic Example of Even Clearer Code**

```
703           For month = 1 To NUM_MONTHS_IN_YEAR  
704              profit(month) = revenue(month) - expense(month)  
705           Next
```

706 This example seems quite good, but we can push it even one step further through
707 using an enumerated type:

708 **Visual Basic Example of Very Clear Code**

```
709           For month = Month_January To Month_December  
710              profit(month) = revenue(month) - expense(month)  
711           Next
```

712 With this final example, there can be no doubt about the purpose of the loop.

713 Even if you think a literal is safe, use named constants instead. Be a fanatic
714 about rooting out literals in your code. Use a text editor to search for 2, 3, 4, 5, 6,
715 7, 8, and 9 to make sure you haven’t used them accidentally.

716 **CROSS-REFERENCE** For
717 details on simulating
718 enumerated types, see “If
719 Your Language Doesn’t
720 Have Enumerated Types” in
721 Section 12.6, earlier in this
722 chapter.

722
723
724
725
726
727
728
729

Simulate named constants with appropriately scoped variables or classes
If your language doesn’t support named constants, you can create your own. By using an approach similar to the approach suggested in the earlier Java example in which enumerated types were simulated, you can gain many of the advantages of named constants. Typical scoping rules apply—prefer local scope, class scope, and global scope in that order.

Use named constants consistently

It’s dangerous to use a named constant in one place and a literal in another to represent the same entity. Some programming practices beg for errors; this one is like calling an 800 number and having errors delivered to your door. If the value of the named constant needs to be changed, you’ll change it and think you’ve made all the necessary changes. You’ll overlook the hard-coded literals, your program will develop mysterious defects and fixing them will be a lot harder than picking up the phone and yelling for help.

730

12.8 Arrays

731
732
733
734

Arrays are the simplest and most common type of structured data. In some languages, arrays are the only type of structured data. An array contains a group of items that are all of the same type and that are directly accessed through the use of an array index. Here are some tips on using arrays.

KEY POINT

735
736
737
738
739
740

Make sure that all array indexes are within the bounds of the array

In one way or another, all problems with arrays are caused by the fact that array elements can be accessed randomly. The most common problem arises when a program tries to access an array element that’s out of bounds. In some languages, this produces an error; in others, it simply produces bizarre and unexpected results.

741
742
743
744
745
746
747

Think of arrays as sequential structures

Some of the brightest people in computer science have suggested that arrays never be accessed randomly, but only sequentially (Mills and Linger 1986). Their argument is that random accesses in arrays are similar to random *gos* in a program: Such accesses tend to be undisciplined, error prone, and hard to prove correct. Instead of arrays, they suggest using sets, stacks, and queues, whose elements are accessed sequentially.

HARD DATA

748
749
750

In a small experiment, Mills and Linger found that designs created this way resulted in fewer variables and fewer variable references. The designs were relatively efficient and led to highly reliable software.

751
752

753 **CROSS-REFERENCE** Issues in using arrays and loops
754 are similar and related. For details on loops, see Chapter
755 16, “Controlling Loops.”
756
757
758
759

760
761
762
763
764
765

766
767
768
769
770

771
772
773
774
775

776
777
778

779
780
781

782
783
784
785
786

Consider using container classes that you access sequentially—sets, stacks, queues, and so on—as alternatives before you automatically choose an array.

Check the end points of arrays

Just as it's helpful to think through the end points in a loop structure, you can catch a lot of errors by checking the end points of arrays. Ask yourself whether the code correctly accesses the first element of the array or mistakenly accesses the element before or after the first element. What about the last element? Will the code make an off-by-one error? Finally, ask yourself whether the code correctly accesses the middle elements of the array.

If an array is multidimensional, make sure its subscripts are used in the correct order

It's easy to say *Array[i][j]* when you mean *Array[j][i]*, so take the time to double-check that the indexes are in the right order. Consider using more meaningful names than *i* and *j* in cases in which their roles aren't immediately clear.

Watch out for index cross talk

If you're using nested loops, it's easy to write *Array[j]* when you mean *Array[i][j]*. Switching loop indexes is called “index cross talk.” Check for this problem. Better yet, use more meaningful index names than *i* and *j* and make it harder to commit cross-talk mistakes in the first place.

Throw in an extra element at the end of an array

Off-by-one errors are common with arrays. If your array access is off by one and you write beyond the end of an array, you can cause a serious error. When you declare the array to be one bigger than the size you think you'll need, you give yourself a cushion and soften the consequences of an off-by-one error.

This is admittedly a sloppy way to program, and you should consider what you're saying about yourself before you do it. But if you decide that it's the least of your evils, it can be an effective safeguard.

In C, use the ARRAY_LENGTH() macro to work with arrays

You can build extra flexibility into your work with arrays by defining an *ARRAY_LENGTH()* macro that looks like this:

C Example of Defining an ARRAY_LENGTH() Macro

```
#define ARRAY_LENGTH( x ) (sizeof(x)/sizeof(x[0]))
```

When you use operations on an array, instead of using a named constant for the upper bound of the array size, use the *ARRAY_LENGTH()* macro. Here's an example:

787

788

789

790

791

792

793 Here's where the macro is
794 used.

795

796

797

798

799

800

801 **CROSS-REFERENCE** In
802 many cases, it's better to
803 create a class than to create a
804 simple data type. For details,
805 see Chapter 6, "Working
Classes."

806

807

808

809

810

811

812

813

814

815

816

817

818

819

820

821

822

823

C Example of Using the `ARRAY_LENGTH()` Macro for Array Operations

```
ConsistencyRatios[] =  
{ 0.0, 0.0, 0.58, 0.90, 1.12,  
1.24, 1.32, 1.41, 1.45, 1.49,  
1.51, 1.48, 1.56, 1.57, 1.59 };  
...  
for ( RatioIdx = 0; RatioIdx < ARRAY_LENGTH( ConsistencyRatios ); RatioIdx++ );  
...
```

This technique is particularly useful for dimensionless arrays such as the one in the example. If you add or subtract entries, you don't have to remember to change a named constant that describes the array's size. Of course, the technique works with dimensioned arrays too, but if you use this approach, you don't always need to set up an extra named constant for the array definition.

12.9 Creating Your Own Types

Programmer-defined variable types are one of the most powerful capabilities a language can give you to clarify your understanding of a program. They protect your program against unforeseen changes and make it easier to read—all without requiring you to design, construct, and test new classes. If you're using C, C++ or another language that allows user-defined types, take advantage of them!

To appreciate the power of type creation, suppose you're writing a program to convert coordinates in an *x*, *y*, *z* system to latitude, longitude, and elevation. You think that double-precision floating-point numbers might be needed but would prefer to write a program with single-precision floating-point numbers until you're absolutely sure. You can create a new type specifically for coordinates by using a `typedef` statement in C or C++ or the equivalent in another language. Here's how you'd set up the type definition in C++:

C++ Example of Creating a Type

```
typedef float Coordinate; // for coordinate variables
```

This type definition declares a new type, *Coordinate*, that's functionally the same as the type *float*. To use the new type, you declare variables with it just as you would with a predefined type such as *float*. Here's an example:

C++ Example of Using the Type You've Created

```
Routine1( ... ) {  
    Coordinate latitude; // latitude in degrees  
    Coordinate longitude; // longitude in degrees  
    Coordinate elevation; // elevation in meters from earth center  
    ...
```

```
824 }  
825 ...  
826  
827 Routine2( ... ) {  
828     Coordinate x; // x coordinate in meters  
829     Coordinate y; // y coordinate in meters  
830     Coordinate z; // z coordinate in meters  
831     ...  
832 }  
833  
834 In this code, the variables latitude, longitude, elevation, x, y, and z are all  
declared to be of type Coordinate.
```

835 Now suppose that the program changes and you find that you need to use
836 double-precision variables for coordinates after all. Because you defined a type
837 specifically for coordinate data, all you have to change is the type definition.
838 And you have to change it in only one place: in the *typedef* statement. Here's the
839 changed type definition:

C++ Example of Changed Type Definition

840
841 *The original float has changed
842 to double.*

```
843     typedef double Coordinate; // for coordinate variables
```

844 Here's a second example—this one in Pascal. Suppose you're creating a payroll
845 system in which employee names are a maximum of 30 characters long. Your
846 users have told you that no one *ever* has a name longer than 30 characters. Do
847 you hard-code the number *30* throughout your program? If you do, you trust
848 your users a lot more than I trust mine! A better approach is to define a type for
849 employee names:

Pascal Example of Creating a Type for Employee Names

849
850 **Type**

```
851     EmployeeName_t = array[ 1..30 ] of char;
```

852 When a string or an array is involved, it's usually wise to define a named
853 constant that indicates the length of the string or array and then use the named
854 constant in the type definition. You'll find many places in your program in
855 which to use the constant—this is just the first place in which you'll use it.
856 Here's how it looks:

Pascal Example of Better Type Creation

856
857
858 *Here's the declaration of the
859 named constant.*

```
860     Const
```

```
861         NAMELENGTH_C = 30;  
862         ...
```

```
863     Type
```

```
864         EmployeeName_t = array[ 1..NAMELENGTH_C ] of char;
```

862 A more powerful example would combine the idea of creating your own types
863 with the idea of information hiding. In some cases, the information you want to
864 hide is information about the type of the data.

865 The coordinates example in C++ is about halfway to information hiding. If you
866 always use *Coordinate* rather than *float* or *double*, you effectively hide the type
867 of the data. In C++, this is about all the information hiding the language does for
868 you. For the rest, you or subsequent users of your code have to have the
869 discipline not to look up the definition of *Coordinate*. C++ gives you figurative,
870 rather than literal, information-hiding ability.

871 Other languages such as Ada go a step further and support literal information
872 hiding. Here's how the *Coordinate* code fragment would look in an Ada package
873 that declares it:

Ada Example of Hiding Details of a Type Inside a Package

```
874 package Transformation is
875   type Coordinate is private;
876   ...
877   This statement declares
878   Coordinate as private to the
879   package.
```

Here's how *Coordinate* looks in another package, one that uses it:

Ada Example of Using a Type from Another Package

```
880 with Transformation;
881 ...
882 procedure Routine1(...) ...
883   latitude: Coordinate;
884   longitude: Coordinate;
885 begin
886   -- statements using latitude and longitude
887   ...
888 end Routine1;
```

889 Notice that the *Coordinate* type is declared as *private* in the package
890 specification. That means that the only part of the program that knows the
891 definition of the *Coordinate* type is the private part of the *Transformation*
892 package. In a development environment with a group of programmers, you could
893 distribute only the package specification, which would make it harder for a
894 programmer working on another package to look up the underlying type of
895 *Coordinate*. The information would be literally hidden. Languages like C++ that
896 require you to distribute the definition of *Coordinate* in header files undermine
897 true information hiding.

898 These examples have illustrated several reasons to create your own types:

- 899 • To make modifications easier. It's little work to create a new type, and it
900 gives you a lot of flexibility.
901 • To avoid excessive information distribution. Hard typing spreads data-typing
902 details around your program instead of centralizing them in one place. This
903 is an example of the information-hiding principle of centralization discussed
904 in Section 6.2.
905 • To increase reliability. In Ada you can define types such as *type Age_t is*
906 *range 0..99*. The compiler then generates run-time checks to verify that any
907 variable of type *Age_t* is always within the range *0..99*.
908 • To make up for language weaknesses. If your language doesn't have the
909 predefined type you want, you can create it yourself. For example, C doesn't
910 have a boolean or logical type. This deficiency is easy to compensate for by
911 creating the type yourself:
912

```
typedef int Boolean_t;
```

913

Why Are the Examples of Creating Your Own 914 Types in Pascal and Ada?

915 Pascal and Ada have gone the way of the stegosaurus and, in general, the
916 languages that have replaced them are more usable. In the area of simple type
917 definitions, however, I think C++, Java, and Visual Basic represent a case of
918 three steps forward and one step back. An Ada declaration like

919

```
currentTemperature: INTEGER range 0..212;
```


920 contains important semantic information that a statement like

921

```
int temperature;
```


922 does not. Going a step further, a type declaration like

923

```
type Temperature is range 0..212;  
...  
currentTemperature: Temperature;
```

924 allows the compiler to ensure that *currentTemperature* is assigned only to other
925 variables with the *Temperature* type, and very little extra coding is required to
926 provide that extra safety margin.

927 Of course a programmer could create a *Temperature* class to enforce the same
928 semantics that were enforced automatically by the Ada language, but the step
929 from creating a simple data type in one line of code to creating a class is a big
930 step. In many situations, a programmer would create the simple type but would
931 not step up to the additional effort of creating a class.

934

935 **CROSS-REFERENCE** In
936 each case, consider whether
937 creating a class might work
938 better than a simple data type.
939 For details, see Chapter 6,
940 “Working Classes.”

941
942943
944
945
946
947
948
949
950951
952
953
954
955
956
957
958959
960
961
962
963964
965
966
967
968
969
970

Guidelines for Creating Your Own Types

Here are a few guidelines to keep in mind as you create your own “user-defined” types:

Create types with functionally oriented names

Avoid type names that refer to the kind of computer data underlying the type. Use type names that refer to the parts of the real-world problem that the new type represents. In the examples above, the definitions created well-named types for coordinates and names—real-world entities. Similarly, you could create types for currency, payment codes, ages, and so on—aspects of real-world problems.

Be wary of creating type names that refer to predefined types. Type names like *BigInteger* or *LongString* refer to computer data rather than the real-world problem. The big advantage of creating your own type is that it provides a layer of insulation between your program and the implementation language. Type names that refer to the underlying programming-language types poke holes in the insulation. They don’t give you much advantage over using a predefined type. Problem-oriented names, on the other hand, buy you easy modifiability and data declarations that are self-documenting.

Avoid predefined types

If there is any possibility that a type might change, avoid using predefined types anywhere but in *typedef* or *type* definitions. It’s easy to create new types that are functionally oriented, and it’s hard to change data in a program that uses hard-wired types. Moreover, use of functionally oriented type declarations partially documents the variables declared with them. A declaration like *Coordinate x* tells you a lot more about *x* than a declaration like *float x*. Use your own types as much as you can.

Don’t redefine a predefined type

Changing the definition of a standard type can create confusion. For example, if your language has a predefined type *Integer*, don’t create your own type called *Integer*. Readers of your code might forget that you’ve redefined the type and assume that the *Integer* they see is the *Integer* they’re used to seeing.

Define substitute types for portability

In contrast to the advice that you not change the definition of a standard type, you might want to define substitutes for the standard types so that on different hardware platforms you can make the variables represent exactly the same entities. For example, you can define a type *INT* and use it instead of *int*, or a type *LONG* instead of *long*. Originally, the only difference between the two types would be their capitalization. But when you moved the program to a new

971 hardware platform, you could redefine the capitalized versions so that they could
972 match the data types on the original hardware.

973 If your language isn't case sensitive, you'll have to differentiate the names by
974 some means other than capitalization.

975 ***Consider creating a class rather than using a typedef***
976 Simple typedefs can go a long way toward hiding information about a variable's
977 underlying type. In some cases, however, you might want the additional
978 flexibility and control you'll achieve by creating a class. For details, see Chapter
979 6, "Working Classes."

CROSS-REFERENCE For
980 a checklist that applies to
general data issues rather
981 than to issues with specific
types of data, see the
982 checklist in Chapter 10,
983 "General Issues in Using
Variables." For a checklist of
984 considerations in naming
985 varieties, see the checklist in
986 Chapter 11, "The Power of
Variable Names."
987

988

989

990

991

992

993

994

995

996

997

998

999

1000

1001

1002

1003

- 1004 Does C code initialize strings to *NULLs* to avoid endless strings?
1005 Does C code use *strncpy()* rather than *strcpy()*? And *strncat()* and *strcmp()*?

1006 **Boolean Variables**

- 1007 Does the program use additional boolean variables to document conditional
1008 tests?
1009 Does the program use additional boolean variables to simplify conditional
1010 tests?

1011 **Enumerated Types**

- 1012 Does the program use enumerated types instead of named constants for their
1013 improved readability, reliability, and modifiability?
1014 Does the program use enumerated types instead of boolean variables when a
1015 variable's use cannot be completely captured with *TRUE* and *FALSE*?
1016 Do tests using enumerated types test for invalid values?
1017 Is the first entry in an enumerated type reserved for "invalid"?

1018 **Named Constants**

- 1019 Does the program use named constants for data declarations and loop limits
1020 rather than magic numbers?
1021 Have named constants been used consistently—not named constants in some
1022 places, literals in others?

1023 **Arrays**

- 1024 Are all array indexes within the bounds of the array?
1025 Are array references free of off-by-one errors?
1026 Are all subscripts on multidimensional arrays in the correct order?
1027 In nested loops, is the correct variable used as the array subscript, avoiding
1028 loop-index cross talk?

1029 **Creating Types**

- 1030 Does the program use a different type for each kind of data that might
1031 change?
1032 Are type names oriented toward the real-world entities the types represent
1033 rather than toward programming-language types?
1034 Are the type names descriptive enough to help document data declarations?
1035 Have you avoided redefining predefined types?
1036 Have you considered creating a new class rather than simply redefining a
1037 type?

1039

1040

1041

1042

1043

1044

1045

1046

Key Points

- Working with specific data types means remembering many individual rules for each type. Use the checklist to make sure that you've considered the common problems.
- Creating your own types makes your programs easier to modify and more self-documenting, if your language supports that capability.
- When you create a simple type using *typedef* or its equivalent, consider whether you should be creating a new class instead.

13

2 Unusual Data Types

3 CC2E.COM/1378

4 Contents

5 13.1 Structures

6 13.2 Pointers

7 13.3 Global Data

8 Related Topics

9 Fundamental data types: Chapter 12

10 Defensive programming: Chapter 8

11 Unusual control structures: Chapter 17

12 Complexity in software development: Section 5.2.

13 Some languages support exotic kinds of data in addition to the data types
14 discussed in the preceding chapter. Section 13.1 describes when you might still
15 use structures rather than classes in some circumstances. Section 13.2 describes
16 the ins and outs of using pointers. If you've ever encountered problems
17 associated with using global data, Section 13.3 explains how to avoid such
difficulties.

18 13.1 Structures

19 The term "structure" refers to data that's built up from other types. Because
20 arrays are a special case, they are treated separately in Chapter 12. This section
21 deals with user-created structured data—*structs* in C and C++ and *Structures* in
22 Visual Basic. In Java and C++, classes also sometimes perform as structures
23 (when the class consists entirely of public data members with no public
24 routines).

25 You'll generally want to create classes rather than structures so that you can take
26 advantage of the functionality and privacy offered by classes in addition to the
27 public data supported by structures. But sometimes directly manipulating blocks
28 of data can be useful, so here are some reasons for using structures:

29 *Use structures to clarify data relationships*

30 Structures bundle groups of related items together. Sometimes the hardest part of
31 figuring out a program is figuring out which data goes with which other data. It's
32 like going to a small town and asking who's related to whom. You come to find
33 out that everybody's kind of related to everybody else, but not really, and you
34 never get a good answer.

35 If the data has been carefully structured, figuring out what goes with what is
36 much easier. Here's an example of data that hasn't been structured:

37 Visual Basic Example of Misleading, Unstructured Variables

```
38 name = inputName  
39 address = inputAddress  
40 phone = inputPhone  
41 title = inputTitle  
42 department = inputDepartment  
43 bonus = inputBonus
```

44 Because this data is unstructured, it looks as if all the assignment statements
45 belong together. Actually, *name*, *address*, and *phone* are variables associated
46 with individual employees and *title*, *department*, and *bonus* are variables
47 associated with a supervisor. The code fragment provides no hint that there are
48 two kinds of data at work. In the code fragment below, the use of structures
49 makes the relationships clearer:

50 Visual Basic Example of More Informative, Structured Variables

```
51 employee.name = inputName  
52 employee.address = inputAddress  
53 employee.phone = inputPhone  
54  
55 supervisor.title = inputTitle  
56 supervisor.department = inputDepartment  
57 supervisor.bonus = inputBonus
```

58 In the code that uses structured variables, it's clear that some of the data is
59 associated with an employee, other data with a supervisor.

60 *Use structures to simplify operations on blocks of data*

61 You can combine related elements into a structure and perform operations on the
62 structure. It's easier to operate on the structure than to perform the same
63 operation on each of the elements. It's also more reliable, and it takes fewer lines
64 of code.

65 Suppose you have a group of data items that belong together—for instance, data
66 about an employee in a personnel database. If the data isn't combined into a

structure, merely copying the group of data can involve a lot of statements. Here's an example in Visual Basic:

Visual Basic Example of Copying a Group of Data Items Clumsily

```
newName = oldName  
newAddress = oldAddress  
newPhone = oldPhone  
newSsn = oldSsn  
newGender = oldGender  
newSalary = oldSalary
```

Every time you want to transfer information about an employee, you have to have this whole group of statements, if you ever add a new piece of employee information—for example, `numWithholdings`—you have to find every place at which you have a block of assignments and add an assignment for `newNumWithholdings = oldNumWithholdings`.

Imagine how horrible swapping data between two employees would be. You don't have to use your imagination—here it is:

CODING HORROR

Visual Basic Example of Swapping Two Groups of Data the Hard Way

```
' swap new and old employee data
previousOldName = oldName
previousOldAddress = oldAddress
previousOldPhone = oldPhone
previousOldSsn = oldSsn
previousOldGender = oldGender
previousOldSalary = oldSalary

oldName = newName
oldAddress = newAddress
oldPhone = newPhone
oldSsn = newSsn
oldGender = newGender
oldSalary = newSalary

newName = previousOldName
newAddress = previousOldAddress
newPhone = previousOldPhone
newSsn = previousOldSsn
newGender = previousOldGender
newSalary = previousOldSalary
```

An easier way to approach the problem is to declare a structured variable. An example of the technique is shown at the top of the next page.

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

Visual Basic Example of Declaring Structures

```
Structure Employee
    name As String
    address As String
    phone As String
    ssn As String
    gender As String
    salary As long
End Structure

Dim newEmployee As Employee
Dim oldEmployee As Employee
Dim previousOldEmployee As Employee
```

Now you can switch all the elements in the old and new employee structures with three statements:

122

Visual Basic Example of an Easier Way to Swap Two Groups of Data

```
previousOldEmployee = oldEmployee
oldEmployee = newEmployee
newEmployee = previousOldEmployee
```

If you want to add a field such as *numWithholdings*, you simply add it to the *Structure* declaration. Neither the three statements above nor any similar statements throughout the program need to be modified. C++ and other languages have similar capabilities.

130 **CROSS-REFERENCE** For
131 details on how much data to
132 share between routines, see
133 “Keep Coupling Loose” in
134 Section 5.3.
135

Use structures to simplify parameter lists

You can simplify routine parameter lists by using structured variables. The technique is similar to the one just shown. Rather than passing each of the elements needed individually, you can group related elements into a structure and pass the whole enchilada as a group structure. Here’s an example of the hard way to pass a group of related parameters.

136

137

138

139

Visual Basic Example of a Clumsy Routine Call without a Structure

```
HardWayRoutine( name, address, phone, ssn, gender, salary )
```

Here’s an example of the easy way to call a routine by using a structured variable that contains the elements of the first parameter list:

140

141

142

143

144

145

Visual Basic Example of an Elegant Routine Call with a Structure

```
EasyWayRoutine( employee )
```

If you want to add *numWithholdings* to the first kind of call, you have to wade through your code and change every call to *HardWayRoutine()*. If you add a *numWithholdings* element to *Employee*, you don’t have to change the parameters to *EasyWayRoutine()* at all.

146 **CROSS-REFERENCE** For
147 details on the hazards of
148 passing too much data, see
149 “Keep Coupling Loose” in
Section 5.3.

150
151
152
153

154
155
156
157
158
159
160
161
162
163

164
165
166
167
168
169
170
171
172

173

174 **KEY POINT**

175
176
177
178
179
180

181
182

You can carry this technique to extremes, putting all the variables in your program into one big, juicy variable and then passing it everywhere. Careful programmers avoid bundling data any more than is logically necessary. Furthermore, careful programmers avoid passing a structure as a parameter when only one or two fields from the structure are needed—they pass the specific fields needed instead. This is an aspect of information hiding: Some information is hidden *in* routines; some is hidden *from* routines. Information is passed around on a need-to-know basis.

Use structures to reduce maintenance

Because you group related data when you use structures, changing a structure requires fewer changes throughout a program. This is especially true in sections of code that aren’t logically related to the change in the structure. Since changes tend to produce errors, fewer changes mean fewer errors. If your *Employee* structure has a *title* field and you decide to delete it, you don’t need to change any of the parameter lists or assignment statements that use the whole structure. Of course, you have to change any code that deals specifically with employee titles, but that is conceptually related to deleting the *title* field and is hard to overlook.

The big advantage of having structured the data comes in sections of code that bear no logical relation to the *title* field. Sometimes programs have statements that refer conceptually to a collection of data rather than to individual components. In such cases, individual components such as the *title* field are referenced merely because they are part of the collection. Such sections of code don’t have any logical reason to work with the *title* field specifically and those sections are easy to overlook when you change *title*. If you use a structure, it’s all right to overlook such sections because the code refers to the collection of related data rather than to each component individually.

13.2 Pointers

Pointer usage is one of the most error-prone areas of modern programming. It’s error-prone to such an extent that modern languages including Java and Visual Basic don’t provide a pointer data type. Using pointers is inherently complicated, and using them correctly requires that you have an excellent understanding of your compiler’s memory-management scheme. Many common security problem, especially buffer overruns, can be traced back to erroneous use of pointers (Howard and LeBlanc 2003).

Even if your language doesn’t require you to use pointers, however, a good understanding of pointers will help your understanding of how your

183 programming language works, and a liberal dose of defensive programming
184 practices will help even further.

185 **Paradigm for Understanding Pointers**

186 Conceptually, every pointer consists of two parts: a location in memory and a
187 knowledge of how to interpret the contents of that location.

188 **Location in Memory**

189 The location in memory is an address, often expressed in hexadecimal notation.
190 An address on a 32-bit processor would be a 32-bit value such as 0x0001EA40.
191 The pointer itself contains only this address. To use the data the pointer points to,
192 you have to go to that address and interpret the contents of memory at that
193 location. If you were to look at the memory in that location, it would be just a
194 collection of bits. It has to be interpreted to be meaningful.

195 **Knowledge of How to Interpret the Contents**

196 The knowledge of how to interpret the contents of a location in memory is
197 provided by the base type of the pointer. If a pointer points to an integer, what
198 that really means is that the compiler interprets the memory location given by the
199 pointer as an integer. Of course, you can have an integer pointer, a string pointer,
200 and a floating-point pointer all pointing at the same memory location. But only
201 one of the pointers interprets the contents at that location correctly.

202 In thinking about pointers, it's helpful to remember that memory doesn't have
203 any inherent interpretation associated with it. It is only through use of a specific
204 type of pointer that the bits in a particular location are interpreted as meaningful
205 data.

206 Figure 9-1 shows several views of the same location in memory, interpreted in
207 several different ways.

208 **F13XX01**

209 **Figure 13-1.**

210 *The amount of memory used by each data type is shown by double lines.*

211 In each of the cases in Figure 13-1, the pointer points to the location containing
212 the hex value 0xA. The number of bytes used beyond the 0A depends on how
213 the memory is interpreted. The way memory contents are used also depends on
214 how the memory is interpreted. (It also depends on what processor you're using,
215 so keep that in mind if you try to duplicate these results on your desktop-
216 CRAY.) The same raw memory contents can be interpreted as a string, an
217 integer, a floating point, or anything else—it all depends on the base type of the
218 pointer that points to the memory.

219 General Tips on Pointers

220 With many types of defects, locating the error is the easiest part of correcting the
221 error. Correcting it is the hard part. Pointer errors are different. A pointer error is
222 usually the result of a pointer's pointing somewhere it shouldn't. When you
223 assign a value to a bad pointer variable, you write data into an area of memory
224 you shouldn't. This is called memory corruption. Sometimes memory corruption
225 produces horrible, fiery system crashes; sometimes it alters the results of a
226 calculation in another part of the program; sometimes it causes your program to
227 skip routines unpredictably; sometimes it doesn't do anything at all. In the last
228 case, the pointer error is a ticking time bomb, waiting to ruin your program five
229 minutes before you show it to your most important customer. In short, symptoms
230 of pointer errors tend to be unrelated to causes of pointer errors. Thus, most of
231 the work in correcting a pointer error is locating the cause.

232 KEY POINT

233 Working with pointers successfully requires a two-pronged strategy. First, avoid
234 installing pointer errors in the first place. Pointer errors are so difficult to find
235 that extra preventive measures are justified. Second, detect pointer errors as soon
236 after they are coded as possible. Symptoms of pointer errors are so erratic that
237 extra measures to make the symptoms more predictable are justified. Here's
 how to achieve these key goals:

238 *Isolate pointer operations in routines or classes*

239 Suppose you use a linked list in several places in a program. Rather than
240 traversing the list manually each place it's used, write access routines such as
241 *NextLink()*, *PreviousLink()*, *InsertLink()*, and *DeleteLink()*. By minimizing the
242 number of places in which pointers are accessed, you minimize the possibility of
243 making careless mistakes that spread throughout your program and take forever
244 to find. Because the code is then relatively independent of data-implementation
245 details, you also improve the chance that you can reuse it in other programs.
246 Writing routines for pointer allocation is another way to centralize control over
247 your data.

248 *Declare and define pointers at the same time*

249 Assigning a variable its initial value close to where it is declared is generally
250 good programming practice, and it's all the more valuable when working with
251 pointers. Here is an example of what not to do:

252 CODY HORROR

```
253 Employee *employeePtr;  
254 // lots of code  
255 ...  
256 employeePtr = new Employee;
```

257 If even this code works correctly initially, it is error prone under modification
258 because there is a chance that someone will try to use *employeePtr* between the
259 point where the pointer is declared and the time it's initialized.

260 Here's a safer approach:

261 C++ Example of Bad Pointer Initialization

```
262 Employee *employeePtr = new Employee;  
263 // lots of code  
264 ...
```

265 ***Check pointers before using them***

266 Before you use a pointer in a critical part of your program, make sure the
267 memory location it points to is reasonable. For example, if you expect memory
268 locations to be between *StartData* and *EndData*, you should view a pointer that
269 points before *StartData* or after *EndData* suspiciously. You'll have to determine
270 what the values of *StartData* and *EndData* are in your environment. You can set
271 this up to work automatically if you use pointers through access routines rather
272 than manipulating them directly.

273 ***Check the variable referenced by the pointer before using it***

274 Sometimes you can perform reasonableness checks on the value the pointer
275 points to. For example, if you are supposed to be pointing to an integer value
276 between 0 and 1000, you should be suspicious of values over 1000. If you are
277 pointing to a C++-style string, you might be suspicious of strings with lengths
278 greater than 100. This can also be done automatically if you work with pointers
279 through access routines.

280 ***Use dog-tag fields to check for corrupted memory***

281 A "tag field" or "dog tag" is a field you add to a structure solely for the purpose
282 of error checking. When you allocate a variable, put a value that should remain
283 unchanged into its tag field. When you use the structure—especially when you
284 delete the memory—check the tag field's value. If the tag field doesn't have the
285 expected value, the data has been corrupted.

286 When you delete the pointer, corrupt the field so that if you accidentally try to
287 free the same pointer again, you'll detect the corruption. For example, let's say
288 that you need to allocate 100 bytes:

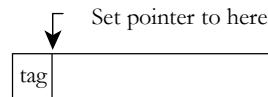
- 289 1. *new* 104 bytes, 4 bytes more than requested.

290
291
292
293

104 bytes

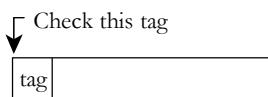
G13XX01294
295
296

2. Set the first 4 bytes to a dog-tag value, and then return a pointer to the memory that starts after that.

297
298
299
300
301
302**G13XX02**

303

3. When the time comes to delete the pointer, check the tag.

304
305

306 Putting a dog tag at the beginning of the memory block you've allocated allows
307 you to check for redundant attempts to deallocate the memory block without
308 needing to maintain a list of all the memory blocks you've allocated. Putting the
309 dog tag at the end of the memory block allows you to check for overwriting
310 memory beyond the location that was supposed to be used. You can use tags at
311 the beginning and the end of the block to accomplish both objectives.

312 You can use this approach in concert with the reasonableness check suggested
313 earlier—checking that the pointers are between *StartData* and *EndData*. To be
314 sure that a pointer points to a reasonable location, rather than checking for a
315 probable range of memory, check to see that the pointer is in the list of allocated
316 pointers.

317 You could check the tag field just once before you delete the variable. A
318 corrupted tag would then tell you that sometime during the life of that variable
319 its contents were corrupted. The more often you check the tag field, however, the
320 closer to the root of the problem you will detect the corruption.

321 **Add explicit redundancies**

322 An alternative to using a tag field is to use certain fields twice. If the data in the
323 redundant fields doesn't match, you know memory has been corrupted. This can
324 result in a lot of overhead if you manipulate pointers directly. If you isolate
325 pointer operations in routines, however, it adds duplicate code in only a few
326 places.

327 **Use extra pointer variables for clarity**

328 By all means, don't skimp on pointer variables. The point is made elsewhere that
329 a variable shouldn't be used for more than one purpose. This is especially true
330 for pointer variables. It's hard enough to figure out what someone is doing with a
331 linked list without having to figure out why one *genericLink* variable is used
332 over and over again or what *pointer->next->last->next* is pointing at. Consider
333 this code fragment:

334 **C++ Example of Traditional Node Insertion Code**

335 void InsertLink(
336 Node *currentNode,
337 Node *insertNode
338) {
339 // insert "insertNode" after "currentNode"
340 insertNode->next = currentNode->next;
341 insertNode->previous = currentNode;
342 if (currentNode->next != NULL) {
343 This line is needlessly difficult.
344 currentNode->next->previous = insertNode;
345 }
346 currentNode->next = insertNode;
347 }

348 This is traditional code for inserting a node in a linked list, and it's needlessly
349 hard to understand. Inserting a new node involves three objects: the current node,
350 the node currently following the current node, and the node to be inserted
351 between them. The code fragment explicitly acknowledges only two objects—
352 *insertNode*, and *currentNode*. It forces you to figure out and remember that
353 *currentNode->next* is also involved. If you tried to diagram what is happening
354 without the node originally following *currentNode*, you would get something
like this:

355 **G13XX05**

356 A better diagram would identify all three objects. It would look like this:

357 **G13XX06**

358 Here's code that explicitly references all three of the objects involved:

359

```
360 void InsertLink(
361     Node *startNode,
362     Node *newMiddleNode
363 ) {
364     // insert "newMiddleNode" between "startNode" and "followingNode"
365     Node *followingNode = startNode->next;
366     newMiddleNode->next = followingNode;
367     newMiddleNode->previous = startNode;
368     if ( followingNode != NULL ) {
369         followingNode->previous = newMiddleNode;
370     }
371     startNode->next = newMiddleNode;
372 }
```

373

374

375

376

377

378

CODING HORROR

379

C++ Example of a Pointer Expression That's Hard to Understand

```
380 for ( rateIndex = 0; rateIndex < numRates; rateIndex++ ) {
381     netRate[ rateIndex ] = baseRate[ rateIndex ] * rates->discounts->factors->net;
382 }
```

383

384

385

386

387

388

389

390

391

392

393

394

395

C++ Example of Simplifying a Complicated Pointer Expression

```
396 quantityDiscount = rates->discounts->factors->net;
397 for ( rateIndex = 0; rateIndex < numRates; rateIndex++ ) {
398     netRate[ rateIndex ] = baseRate[ rateIndex ] * quantityDiscount;
399 }
```

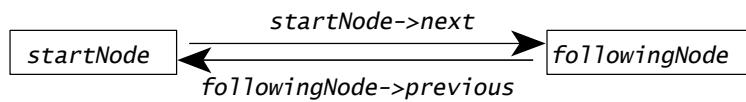
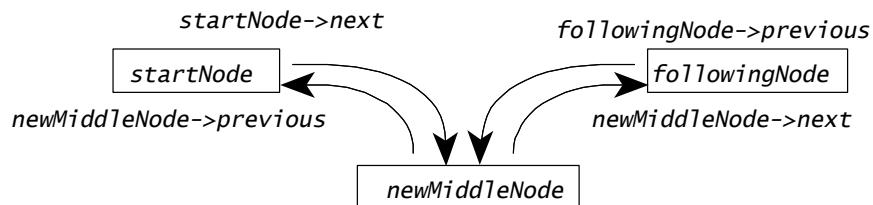
With this simplification, not only do you get a gain in readability, but you might also get a boost in performance from simplifying the pointer operation inside the loop. As usual, you'd have to measure the performance benefit before you bet any folding money on it.

396
397
398
399

CROSS-REFERENCE Diagrams such as this can become part of the external documentation of your program. For details on good documentation practices, see Chapter 32, "Self-Documenting Code."

Draw a picture

Code descriptions of pointers can get confusing. It usually helps to draw a picture. For example, a picture of the linked-list insertion problem might look like the one shown in Figure 13-2.

Initial Linkage**Desired Linkage****F13xx02****Figure 13-2**

An example of a picture that helps think through the steps involved in relinking pointers.

Free pointers in linked lists in the right order

A common problem in working with dynamically allocated linked lists is freeing the first pointer in the list first and then not being able to get to the next pointer in the list. To avoid this problem, make sure that you have a pointer to the next element in a list before you free the current one.

Allocate a reserve parachute of memory

If your program uses dynamic memory, you need to avoid the problem of suddenly running out of memory, leaving your user and your user's data lost in RAM space. One way to give your program a margin of error is to pre-allocate a memory parachute. Determine how much memory your program needs to save work, clean up, and exit gracefully. Allocate that amount of memory at the beginning of the program as a reserve parachute, and leave it alone. When you run out of memory, free the reserve parachute, clean up, and shut down.

Free pointers at the same scoping level as they were allocated

Keep allocation and deallocation of pointers symmetric. If you use a pointer within a single scope, call *new* to allocate and *delete* to deallocate the pointer within the same scope. If you allocate a pointer inside a routine, deallocate it inside a sister routine. If you allocate a pointer inside an object's constructor,

400
401
402
403
404405
406
407
408
409410
411
412
413
414
415
416
417418
419
420
421
422

423
424
425

426
427 **FURTHER READING** For an
428 excellent discussion of safe
429 approaches to handling
430 pointers in C, see *Writing
Solid Code* (Maguire 1993).

431
432
433
434

435
436
437
438
439

440
441
442
443
444
445
446
447

448
449

450
451
452
453

454
455
456
457

deallocate it inside the object's destructor. A routine that allocates memory and then expects its client code to deallocate the memory manually creates an inconsistency that is ripe for error.

Shred your garbage

Pointer errors are hard to debug because the point at which the memory the pointer points to becomes invalid is not deterministic. Sometimes the memory contents will look valid long after the pointer is freed. Other times, the memory will change right away.

You can force errors related to using deallocated pointers to be more consistent by overwriting memory blocks with junk data right before they're deallocated. As with many other operations, you can do this automatically if you use access routines. In C++, each time you delete a pointer, you could use code like this:

C++ Example of Forcing Deallocated Memory to Contain Junk Data

```
memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) );  
delete pointer;
```

Of course, this technique requires that you maintain a list of pointers that can be retrieved with the *MemoryBlockSize()* routine, which I'll discuss later.

Set pointers to NULL after deleting or freeing them

A common type of pointer error is the "dangling pointer," use of a pointer that has been *delete()*d or *free()*d. One reason pointer errors are hard to detect is that sometimes the error doesn't produce any symptoms. By setting pointers to NULL after freeing them, you don't change the fact that you can read data pointed to by a dangling pointer. But you do ensure that writing data to a dangling pointer produces an error. It will probably be an ugly, nasty, disaster of an error, but at least you'll find it instead of someone else finding it.

The code preceding the *delete* operation above could be augmented to handle this too:

C++ Example of Setting Pointers to NULL in a Replacement for *delete*

```
memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) );  
delete pointer;  
pointer = NULL;
```

Check for bad pointers before deleting a variable

One of the best ways to ruin a program is to *free()* or *delete()* a pointer after it has already been *free()*d or *delete()*d. Unfortunately, few languages detect this kind of problem.

458 Setting freed pointers to *NULL* also allows you to check whether a pointer is set
459 to *NULL* before you use it or attempt to delete it again; if you don't set freed
460 pointers to *NULL*, you won't have that option. That suggests another addition to
461 the pointer deletion code:

C++ Example of Setting Pointers to NULL in a Replacement for *delete*

```
462     ASSERT( pointer != NULL, "Attempting to delete NULL pointer." );  
463     memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) );  
464     delete pointer;  
465     pointer = NULL;
```

Keep track of pointer allocations

466 Keep a list of the pointers you have allocated. This allows you to check whether
467 a pointer is in the list before you dispose of it. Here's an example of how the
468 standard pointer deletion code could be modified to include that:

C++ Example of Checking Whether a Pointer has been Allocated

```
469     ASSERT( pointer != NULL, "Attempting to delete NULL pointer." );  
470     if ( IsPointerInList( pointer ) ) {  
471         memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) );  
472         RemovePointerFromList( pointer );  
473         delete pointer;  
474         pointer = NULL;  
475     }  
476     else {  
477         ASSERT( FALSE, "Attempting to delete unallocated pointer." );  
478     }
```

Write cover routines to centralize your strategy to avoiding pointer problems

479 As you can see from the preceding example, you can end up with quite a lot of
480 extra code each time a pointer is *new'd* or *delete'd*. Some of the techniques
481 described in this section are mutually exclusive or redundant, and you wouldn't
482 want to have multiple, conflicting strategies in use in the same code base. For
483 example, you don't need to create and check dog tag values if you're
484 maintaining your own list of valid pointers.

485 You can minimize programming overhead and reduce chance of errors by
486 creating cover routines for common pointer operations. In C++ you could use
487 these two routines:

- 488
- 489 • *SAFE_NEW*. This routine calls *new* to allocate the pointer, adds the new
490 pointer to a list of allocated pointers, and returns the newly allocated pointer
491 to the calling routine. It can also check for a NULL return from *new* (aka an
492

496 “out-of-memory” error) in this one place only, which simplifies error
497 processing in other parts of your program.

- 498
- 499
- 500
- 501
- 502
- 503
- *SAFE_DELETE*. This routine checks to see whether the pointer passed to it
is in the list of allocated pointers. If it is in the list, it sets the memory the
pointer pointed at to garbage values, removes the pointer from the list, calls
C++’s *delete* operator to deallocate the pointer, and sets the pointer to
NULL. If the pointer isn’t in the list, *SAFE_DELETE* displays a diagnostic
message and stops the program.

504 Here’s how the *SAFE_DELETE* routine would look, implemented here as a
505 macro:

C++ Example of Putting a Wrapper Around Pointer Deletion Code

```
506
507 #define SAFE_DELETE( pointer ) { \
508     ASSERT( pointer != NULL, "Attempting to delete NULL pointer." ); \
509     if ( IsPointerInList( pointer ) ) { \
510         memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) ); \
511         RemovePointerFromList( pointer ); \
512         delete pointer; \
513         pointer = NULL; \
514     } \
515     else { \
516         ASSERT( FALSE, "Attempting to delete unallocated pointer." ); \
517     } \
518 }
```

519 **CROSS-REFERENCE** For
520 details on planning to remove
code used for debugging, see
521 “Plan to Remove Debugging
Aids” in Section 8.6.

522

523

524

525

526

527

528

529

530

531

532

In C++, this routine will delete individual pointers, but you would also need to implement a similar *SAFE_DELETE_ARRAY* routine to delete arrays.

By centralizing memory handling in these two routines, you can also make *SAFE_NEW* and *SAFE_DELETE* behave differently in debug mode vs. production mode. For example when *SAFE_DELETE* detects an attempt to free a null pointer during development, it might stop the program, but during production it might simply log an error and continue processing.

You can easily adapt this scheme to *calloc()* and *free()* in C and to other languages that use pointers.

Use a nonpointer technique

Pointers are harder than average to understand, they’re error prone, and they tend to require machine-dependent, unportable code. If you can think of an alternative to using a pointer that works reasonably, save yourself a few headaches and use it instead.

533

534 **FURTHER READING** For
535 many more tips on using
pointers in C++, see *Effective
536 C++*, 2d Ed. (Meyers 1998)
and *More Effective C++*
537 (Meyers 1996).
538

539

540

541

542

543

544

545

546

547

548

549

550

551

552

553

554

555

556

557

558

559

560

561

562

563

564

565

566

567

568

569

C++ Pointer Pointers

C++ introduces some specific wrinkles related to using pointers and references. Here are some guidelines that apply to using pointers in C++.

Understand the difference between pointers and references

In C++, both pointers (*) and the references (&) refer indirectly to an object, and to the uninitiated the only difference appears to be a purely cosmetic distinction between referring to fields as *object->field* vs. *object.field*. The most significant differences are that a reference must always refer to an object, whereas a pointer can point to NULL; and what a reference refers to can't be changed after the reference is initialized.

Use pointers for “pass by reference” parameters and const references for “pass by value” parameters

C++ defaults to passing arguments to routines by value rather than by reference. When you pass an object to a routine by value, C++ creates a copy of the object, and when the object is passed back to the calling routine, a copy is created again. For large objects, that copying can eat up time and resources. Consequently, when passing objects to a routine, you usually want to avoid copying the object, which means you want to pass it by reference rather than by value.

Sometimes, however, you would like to have the *semantics* of pass by reference—that is, that the passed object should not be altered—with the *implementation* of pass by value—that is, passing the actual object rather than a copy.

In C++, the resolution to this issue is that you use pointers for pass by reference, and—odd as the terminology might sound—*const references* for pass by value! Here's an example:

C++ Example of Passing Parameters by Reference and by Value

```
void SomeRoutine(
    const LARGE_OBJECT &nonmodifiableObject,
    LARGE_OBJECT *modifiableObject
);
```

This approach provides the additional benefit of providing a syntactic differentiation within the called routine between objects that are supposed to be treated as modifiable and those that aren't. In a modifiable object, the references to members will use the *object->member* notation, whereas for nonmodifiable objects references to members will use *object.member* notation.

The limitation of this approach is difficulties propagating *const* references. If you control your own code base, it's good discipline to use *const* whenever possible

570 (Meyers 1998), and you should be able to declare pass-by-value parameters as
571 *const* references. For library code or other code that you don't control, you'll run
572 into problems using *const* routine parameters. The fallback position is still to use
573 references for read-only parameters but not declare them *const*. With that
574 approach, you won't realize the full benefits of the compiler checking for
575 attempts to modify non-modifiable arguments to a routine, but you'll at least
576 give yourself the visual distinction between *object->member* and *object.member*.

577 **Use auto_ptrs**

578 If you haven't developed the habit of using *auto_ptrs*, get into the habit!
579 *auto_ptrs* avoid many of the memory-leakage problems associated with regular
580 pointers by deleting memory automatically when the *auto_ptr* goes out of scope.
581 Scott Meyers' *More Effective C++*, Item #9 contains a good discussion of
582 *auto_ptr* (Meyers 1996).

583 **Get smart about smart pointers**

584 Smart pointers are a replacement for regular pointers or "dumb" pointers
585 (Meyers 1996). They operate similarly to regular pointers, but they provide more
586 control over resource management, copy operations, assignment operations,
587 object construction, and object destruction. The issues involved are specific to
588 C++. *More Effective C++*, Item #28, contains a complete discussion.

589 **C-Pointer Pointers**

590 Here are a few tips on using pointers that apply specifically to the C language.

591 **Use explicit pointer types rather than the default type**

592 C lets you use *char* or *void* pointers for any type of variable. As long as the
593 pointer points, the language doesn't really care what it points at. If you use
594 explicit types for your pointers, however, the compiler can give you warnings
595 about mismatched pointer types and inappropriate dereferences. If you don't, it
596 can't. Use the specific pointer type whenever you can.

597 The corollary to this rule is to use explicit type casting when you have to make a
598 type conversion. For example, in the fragment below, it's clear that a variable of
599 type *NODE_PTR* is being allocated:

600 **C Example of Explicit Type Casting**

```
601 NodePtr = (NODE_PTR) malloc( 1, sizeof( NODE ) );
```

602 **Avoid type casting**

603 Avoiding type casting doesn't have anything to do with going to acting school or
604 getting out of always playing "the heavy." It has to do with avoiding squeezing a
605 variable of one type into the space for a variable of another type. Type casting

606 turns off your compiler's ability to check for type mismatches and therefore
607 creates a hole in your defensive-programming armor. A program that requires
608 many type casts probably has some architectural gaps that need to be revisited.
609 Redesign if that's possible; otherwise, try to avoid type casts as much as you can.

610 ***Follow the asterisk rule for parameter passing***

611 You can pass an argument back from a routine in C only if you have an asterisk
612 (*) in front of the argument in the assignment statement. Many C programmers
613 have difficulty determining when C allows a value to be passed back to a calling
614 routine. It's easy to remember that, as long as you have an asterisk in front of the
615 parameter when you assign it a value, the value is passed back to the calling
616 routine. Regardless of how many asterisks you stack up in the declaration, you
617 must have at least one in the assignment statement if you want to pass back a
618 value. For example, in the following fragment, the value assigned to *parameter*
619 isn't passed back to the calling routine because the assignment statement doesn't
620 use an asterisk:

C Example of Parameter Passing That Won't Work

```
621 void TryToPassBackAValue( int *parameter ) {  
622     parameter = SOME_VALUE;  
623 }  
624
```

625 In the next fragment, the value assigned to *parameter* is passed back because
626 *parameter* has an asterisk in front of it:

C Example of Parameter Passing That Will Work

```
627 void TryToPassBackAValue( int *parameter ) {  
628     *parameter = SOME_VALUE;  
629 }  
630
```

Use sizeof() to determine the size of a variable in a memory allocation

631 It's easier to use *sizeof()* than to look up the size in a manual, and *sizeof()* works
632 for structures you create yourself, which aren't in the manual. *sizeof()* doesn't
633 carry a performance penalty since it's calculated at compile time. It's portable—
634 recompiling in a different environment automatically changes the value
635 calculated by *sizeof()*. And it requires little maintenance since you can change
636 types you have defined and allocations will be adjusted automatically.
637

638

639 **CROSS-REFERENCE** For
640 details on the differences
641 between global data and class
642 data, see “Class Data
643 Mistaken For Global Data” in
Section 5.3.

644

645

646

647 **KEY POINT**

648

649

650

651

652

653

654

655

656

657

658

659

660

661

662

663

664

665

666 theAnswer is a global
667 GetOtherAnswer() changes
668 averageAnswer is wrong.

669

670

671

672

13.3 Global Data

Global variables are accessible anywhere in a program. The term is also sometimes used sloppily to refer to variables with a broader scope than local variables—such as class variables that are accessible anywhere within a single class. But accessibility anywhere within a single class does not by itself mean that a variable is global.

Most experienced programmers have concluded that using global data is riskier than using local data. Most experienced programmers have also concluded that access to data from several routines is pretty doggone useful.

Even if global variables don’t always produce errors, however, they’re hardly ever the best way to program. The rest of this section fully explores the issues involved.

Common Problems with Global Data

If you use global variables indiscriminately or you feel that not being able to use them is restrictive, you probably haven’t caught on to the full value of information hiding and modularity yet. Modularity, information hiding, and the associated use of well-designed classes might not be revealed truths, but they go a long way toward making large programs understandable and maintainable. Once you get the message, you’ll want to write routines and classes with as little connection as possible to global variables and the outside world.

People cite numerous problems in using global data, but the problems boil down to a small number of major issues.

Inadvertent changes to global data

You might change the value of a global variable in one place and mistakenly think that it has remained unchanged somewhere else. Such a problem is known as a side effect. For example, in the following code fragment, *TheAnswer* is a global variable:

Visual Basic Example of a Side-Effect Problem

```
theAnswer = GetTheAnswer()  
otherAnswer = GetOtherAnswer()  
averageAnswer = (theAnswer + otherAnswer) / 2
```

You might assume that the call to *GetOtherAnswer()* doesn’t change the value of *theAnswer*; if it does, the average in the third line will be wrong. And in fact, *GetOtherAnswer()* does change the value of *theAnswer*, so the program has an error to be fixed.

673
674
675
676
677

CODING HORROR

678
679
680
681
682
683
684
685

Bizarre and exciting aliasing problems with global data

“Aliasing” refers to calling the same variable by two or more different names. This happens when a global variable is passed to a routine and then used by the routine both as a global variable and as a parameter. Here’s a routine that uses a global variable:

Visual Basic Example of a Routine That’s Ripe for an Aliasing Problem

```
Sub WriteGlobal( ByRef inputVar As Integer )
    inputVar = 0
    globalVar = inputVar + 5
    MsgBox( "Input Variable: " & Str( inputVar ) )
    MsgBox( "Global Variable: " & Str( globalVar ) )
End Sub
```

Here’s the code that calls the routine with the global variable as an argument:

Visual Basic Example of Calling the Routine with an Argument, Which Exposes Aliasing Problem

```
WriteGlobal( globalVar )
```

Since *inputVar* is initialized to 0 and *WriteGlobal()* adds 5 to *inputVar* to get *globalVar*, you’d expect *globalVar* to be 5 more than *inputVar*. But here’s the surprising result:

The Result of the Aliasing Problem in Visual Basic

```
Input Variable: 5
Global Variable: 5
```

The subtlety here is that *globalVar* and *inputVar* are actually the same variable! Since *globalVar* is passed into *WriteGlobal()* by the calling routine, it’s referenced or “aliased” by two different names. The effect of the *MsgBox()* lines is thus quite different from the one intended: They display the same variable twice, even though they refer to two different names.

Re-entrant code problems with global data

Code that can be entered by more than one thread of control is becoming increasingly common. Such code is used in programs for Microsoft Windows, the Apple Macintosh, and Linux and also in recursive routines. Re-entrant code creates the possibility that global data will be shared not only among routines, but among different copies of the same program. In such an environment, you have to make sure that global data keeps its meaning even when multiple copies of a program are running. This is a significant problem, and you can avoid it by using techniques suggested later in this section.

KEY POINT

701
702
703
704
705
706
707
708

709 ***Code reuse hindered by global data***

710 In order to use code from one program in another program, you have to be able
711 to pull it out of the first program and plug it into the second. Ideally, you'd be
712 able to lift out a single routine or class, plug it into another program, and
713 continue merrily on your way.

714 Global data complicates the picture. If the class you want to reuse reads or writes
715 global data, you can't just plug it into the new program. You have to modify the
716 new program or the old class so that they're compatible. If you take the high
717 road, you'll modify the old class so that it doesn't use global data. If you do that,
718 the next time you need to reuse the class you'll be able to plug it in with no extra
719 fuss. If you take the low road, you'll modify the new program to create the
720 global data that the old class needs to use. This is like a virus; not only does the
721 global data affect the original program, but it also spreads to new programs that
722 use any of the old program's classes.

723 ***Uncertain initialization-order issues with global data***

724 The order in which data is initialized among different "translation units" (files) is
725 not defined in some languages, notably, C++. If the initialization of a global
726 variable in one file uses a global variable that was initialized in a different file,
727 all bets are off on the value of the second variable unless you take explicit steps
728 to ensure the two variables are initialized in the right sequence.

729 This problem is solvable with a workaround that Scott Meyers describes in
730 *Effective C++*, Item #47 (Meyers 1998). But the trickiness of the solution is
731 representative of the extra complexity that using global data introduces.

732 ***Modularity and intellectual manageability damaged by global data***

733 The essence of creating programs that are larger than a few hundred lines of code
734 is managing complexity. The only way you can intellectually manage a large
735 program is to break it into pieces so that you only have to think about one part at
736 a time. Modularization is the most powerful tool at your disposal for breaking a
737 program into pieces.

738 Global data pokes holes in your ability to modularize. If you use global data, can
739 you concentrate on one routine at a time? No. You have to concentrate on one
740 routine and every other routine that uses the same global data. Although global
741 data doesn't completely destroy a program's modularity, it weakens it, and that's
742 reason enough to try to find better solutions to your problems.

743 **Reasons to Use Global Data**

744 Data purists sometimes argue that programmers should never use global data, but
745 most programs use "global data" when the term is broadly construed. Data in a

746 database is global data, as is data in configuration files such as the Windows
747 registry. Named constants are global data, just not global variables.

748 Used with discipline, global variables are useful in several situations:

749 ***Preservation of global values***

750 Sometimes you have data that applies conceptually to your whole program. This
751 might be a variable that reflects the state of a program—for example, interactive
752 vs. command-line mode, or normal vs. error-recovery mode. Or it might be
753 information that's needed throughout a program—for example, a data table that
754 every routine in the program uses.

755 **CROSS-REFERENCE** For
756 more details on named
757 constants, see Section 12.7,
758 “Named Constants.”

759 Although C++, Java, Visual Basic, and most modern languages support named
760 constants, some languages such as Python, Perl, Awk, and Unix shell script still
761 don't. You can use global variables as substitutes for named constants when your
762 language doesn't support them. For example, you can replace the literal values *1*
763 and *0* with the global variables *TRUE* and *FALSE* set to *1* and *0*, or replace *66* as
764 the number of lines per page with *LINES_PER_PAGE = 66*. It's easier to change
765 code later when this approach is used, and the code tends to be easier to read.
This disciplined use of global data is a prime example of the distinction between
programming *in* vs. programming *into* a language, which is discussed more in
Section 34.4, “Program Into Your Language, Not In It.”

766 ***Emulation of enumerated types***
767 You can also use global variables to emulate enumerated types in languages such
768 as Python that don't support enumerated types directly.

769 ***Streamlining use of extremely common data***
770 Sometimes you have so many references to a variable that it appears in the
771 parameter list of every routine you write. Rather than including it in every
772 parameter list, you can make it a global variable. In cases in which a variable
773 seems to be accessed everywhere, however, it rarely is. Usually it's accessed by
774 a limited set of routines you can package into a class with the data they work on.
775 More on this later.

776 ***Eliminating tramp data***
777 Sometimes you pass data to a routine or class merely so that it can be passed to
778 another routine or class. For example, you might have an error-processing object
779 that's used in each routine. When the routine in the middle of the call chain
780 doesn't use the object, the object is called “tramp data.” Use of global variables
781 can eliminate tramp data.

782 **Use Global Data Only as a Last Resort**

783 Before you resort to using global data, consider a few alternatives.

784 ***Begin by making each variable local and make variables global only as you
785 need to***

786 Make all variables local to individual routines initially. If you find they're
787 needed elsewhere, make them private or protected class variables before you go
788 so far as to make them global. If you finally find that you have to make them
789 global, do it, but only when you're sure you have to. If you start by making a
790 variable global, you'll never make it local, whereas if you start by making it
791 local, you might never need to make it global.

792 ***Distinguish between global and class variables***

793 Some variables are truly global in that they are accessed throughout a whole
794 program. Others are really class variables, used heavily only within a certain set
795 of routines. It's OK to access a class variable any way you want to within the set
796 of routines that use it heavily. If other routines need to use it, provide the
797 variable's value by means of an access routine. Don't access class values
798 directly—as if they were global variables—even if your programming language
799 allows you to. This advice is tantamount to saying "Modularize! Modularize!
800 Modularize!"

801 ***Use access routines***

802 Creating access routines is the workhorse approach to getting around problems
803 with global data. More on that in the next section.

804 **Using Access Routines Instead of Global Data**

805 **KEY POINT**

806 Anything you can do with global data, you can do better with access routines.
807 The use of access routines is a core technique for implementing abstract data
808 types and achieving information hiding. Even if you don't want to use a full-
809 blown abstract data type, you can still use access routines to centralize control
over your data and to protect yourself against changes.

810 **Advantages of Access Routines**

811 Here are several advantages of using access routines:

- 812 ● You get centralized control over the data. If you discover a more appropriate
813 implementation of the structure later, you don't have to change the code
814 everywhere the data is referenced. Changes don't ripple through your whole
815 program. They stay inside the access routines.

816 **CROSS-REFERENCE** For
817 more details on barricading,
818 see Section 8.5, “Barricade
819 Your Program to Contain the
820 Damage Caused by Errors.”

821
822

823 **CROSS-REFERENCE** For
824 details on information hiding,
825 see “Hide Secrets
826 (Information Hiding)” in
827 Section 5.3.

828

829
830
831
832
833
834
835
836

837
838
839
840
841
842

- You can ensure that all references to the variable are barricaded. If you allow yourself to push elements onto the stack with statements like `stack.array[stack.top] = newElement`, you can easily forget to check for stack overflow and make a serious mistake. If you use access routines, for example, `PushStack(newElement)`—you can write the check for stack overflow into the `PushStack()` routine; the check will be done automatically every time the routine is called, and you can forget about it.
- You get the general benefits of information hiding automatically. Access routines are an example of information hiding, even if you don’t design them for that reason. You can change the interior of an access routine without changing the rest of the program. Access routines allow you to redecorate the interior of your house and leave the exterior unchanged so that your friends still recognize it.
- Access routines are easy to convert to an abstract data type. One advantage of access routines is that you can create a level of abstraction that’s harder to do when you’re working with global data directly. For example, instead of writing code that says `if lineCount > MAX_LINES`, an access routine allows you to write code that says `if PageFull()`. This small change documents the intent of the `if lineCount` test, and it does so *in the code*. It’s a small gain in readability, but consistent attention to such details makes the difference between beautifully crafted software and code that’s just hacked together.

How to Use Access Routines

Here’s the short version of the theory and practice of access routines: Hide data in a class. Declare that data using the `static` keyword or its equivalent to ensure there is only a single instance of the data. Write routines that let you look at the data and change it. Require code outside the class to use the access routines rather than working directly with the data.

For example, if you have a global status variable `g_globalStatus` that describes your program’s overall status, you can create two access routines: `globalStatus.Get()` and `globalStatus.set()`, each of which does what it sounds like it does. Those routines access a variable hidden within the class that replaces `g_globalStatus`. The rest of the program can get all the benefit of the formerly-global variable by accessing `globalStatus.Get()` and `globalStatus.Set()`.

849 **CROSS-REFERENCE** Rest
850 ricting access to global
851 variables even when your
852 language doesn't directly
support that is an example of
853 programming *into* a language
854 vs. programming *in* a
language. For more details,
see Section 34.4, "Program
855 Into Your Language, Not In
856 It."

857
858
859

860
861
862
863
864
865
866

867
868
869
870
871
872
873

874 **CROSS-REFERENCE** For
875 details on planning for
876 differences between
877 developmental and
878 production versions of a
879 program, see "Plan to
880 Remove Debugging Aids" in
881 Section 8.6 and Section 8.7,
882 "Determining How Much
Defensive Programming to
Leave in Production Code."

883
884

If your language doesn't support classes, you can still create access routines to manipulate the global data but you'll have to enforce restrictions on the use of the global data through coding standards in lieu of built-in programming language enforcement.

Here are a few detailed guidelines for using access routines to hide global variables when your language doesn't have built-in support:

Require all code to go through the access routines for the data

A good convention is to require all global data to begin with the `g_` prefix, and to further require that no code access a variable with the `g_` prefix except that variable's access routines. All other code reaches the data through the access-routines.

Don't just throw all your global data into the same barrel

If you throw all your global data into a big pile and write access routines for it, you eliminate the problems of global data but you miss out on some of the advantages of information hiding and abstract data types. As long as you're writing access routines, take a moment to think about which class each global variable belongs in and then package the data and its access routines with the other data and routines in that class.

Use locking to control access to global variables

Similar to concurrency control in a multi-user database environment, locking requires that before the value of a global variable can be used or updated, the variable must be "checked out." After the variable is used, it's checked back in. During the time it's in use (checked out), if some other part of the program tries to check it out, the lock/unlock routine displays an error message or fires an assertion.

This description of locking ignores many of the subtleties of writing code to fully support concurrency. For that reason, simplified locking schemes like this one are most useful during the development stage. Unless the scheme is very well thought out, it probably won't be reliable enough to be put into production. When the program is put into production, the code is modified to do something safer and more graceful than displaying error messages. For example, it might log an error message to a file when it detects multiple parts of the program trying to lock the same global variable.

This sort of development-time safeguard is fairly easy to implement when you use access routines for global data but would be awkward to implement if you were using global data directly.

885 ***Build a level of abstraction into your access routines***
 886 Build access routines at the level of the problem domain rather than at the level
 887 of the implementation details. That approach buys you improved readability as
 888 well as insurance against changes in the implementation details.

889 Compare the following pairs of statements:

Direct Use of Global Data	Use of Global Data Through Access Routines
<code>node = node.next</code>	<code>account = NextAccount(account)</code>
<code>node = node.next</code>	<code>employee = NextEmployee(employee)</code>
<code>node = node.next</code>	<code>rateLevel = NextRateLevel(rateLevel)</code>
<code>event = eventQueue[queueFront]</code>	<code>event = HighestPriorityEvent()</code>
<code>event = eventQueue[queueBack]</code>	<code>event = LowestPriorityEvent()</code>

890 In the first three examples, the point is that an abstract access routine tells you a
 891 lot more than a generic structure. If you use the structure directly, you do too
 892 much at once: You show both what the structure itself is doing (moving to the
 893 next link in a linked list) and what's being done with respect to the entity it
 894 represents (getting an account, next employee, or rate level). This is a big burden
 895 to put on a simple data-structure assignment. Hiding the information behind
 896 abstract access routines lets the code speak for itself and makes the code read at
 897 the level of the problem domain, rather than at the level of implementation
 898 details.

899 ***Keep all accesses to the data at the same level of abstraction***
 900 If you use an access routine to do one thing to a structure, you should use an
 901 access routine to do everything else to it too. If you read from the structure with
 902 an access routine, write to it with an access routine. If you call *InitStack()* to
 903 initialize a stack and *PushStack()* to push an item onto the stack, you've created
 904 a consistent view of the data. If you pop the stack by writing `value = array[`
 905 `stack.top]`, you've created an inconsistent view of the data. The inconsistency
 906 makes it harder for others to understand the code. Create a *PopStack()* routine
 907 instead of writing `value = array[stack.top]`.

908 **CROSS-REFERENCE** Using
 909 access routines for an event
 910 queue suggests the need to
 911 create a class. For details, see
 912 Chapter 6, "Working
 913 Classes."
 914
 915
 916 In the example pairs of statements in the table above, the two event-queue
 operations occurred in parallel. Inserting an event into the queue would be
 trickier than either of the two operations in the table, requiring several lines of
 code to find the place to insert the event, adjust existing events to make room for
 the new event, and adjust the front or back of the queue. Removing an event
 from the queue would be just as complicated. During coding, the complex
 operations would be put into routines and the others would be left as direct data
 manipulations. This would create an ugly, nonparallel use of the structure.
 Compare the following pairs of statements:

Non-Parallel Use of Complex Data	Parallel Use of Complex Data
<code>event = EventQueue[queueFront]</code>	<code>event = HighestPriorityEvent()</code>
<code>event = EventQueue[queueBack]</code>	<code>event = LowestPriorityEvent()</code>
<code>AddEvent(event)</code>	<code>AddEvent(event)</code>
<code>eventCount = eventCount - 1</code>	<code>RemoveEvent(event)</code>

917
 918 Although you might think that these guidelines apply only to large programs,
 919 access routines have shown themselves to be a productive way of avoiding the
 920 problems of global data. As a bonus, they make the code more readable and add
 flexibility.

921 How to Reduce the Risks of Using Global Data

922 In most instances, global data is really class data for a class that hasn't been
 923 designed or implemented very well. In a few instances, data really does need to
 924 be global, but accesses to it can be wrapped with access routines to minimize
 925 potential problems. In a tiny number of remaining instances, you really do need
 926 to use global data. In those cases, you might think of following the guidelines in
 927 this section as getting shots so that you can drink the water when you travel to a
 928 foreign country: They're kind of painful, but they improve the odds of staying
 929 healthy.

930 **CROSS-REFERENCE** For
 931 details on naming
 932 conventions for global
 933 variables, see "Identify global
 934 variables" in Section 11.4.

Develop a naming convention that makes global variables obvious

You can avoid some mistakes just by making it obvious that you're working with global data. If you're using global variables for more than one purpose (for example, as variables and as substitutes for named constants), make sure your naming convention differentiates among the types of uses.

Create a well-annotated list of all your global variables

Once your naming convention indicates that a variable is global, it's helpful to indicate what the variable does. A list of global variables is one of the most useful tools that someone working with your program can have.

Don't use global variables to contain intermediate results

If you need to compute a new value for a global variable, assign the global variable the final value at the end of the computation rather than using it to hold the result of intermediate calculations.

Don't pretend you're not using global data by putting all your data into a monster object and passing it everywhere

Putting everything into one huge object might satisfy the letter of the law by avoiding global variables. But it's pure overhead, producing none of the benefits of true encapsulation. If you use global data, do it openly. Don't try to disguise it with obese objects.

949 CC2E.COM/1385

Additional Resources

950 Maguire, Steve. *Writing Solid Code*. Redmond, WA: Microsoft Press, 1993.951 Chapter 3 contains an excellent discussion of the hazards of pointer use and
952 numerous specific tips for avoiding problems with pointers.953 Meyers, Scott. *Effective C++*, 2d Ed, Reading, Mass.: Addison Wesley, 1998;954 Meyers, Scott, *More Effective C++*, Reading, Mass.: Addison Wesley, 1996. As
955 the titles suggest, these books contain numerous specific tips for improving C++
956 programs, including guidelines for using pointers safely and effectively. *More*
957 *Effective C++* in particular contains an excellent discussion of C++'s memory
958 management issues.

959 CC2E.COM/1392

CHECKLIST: Considerations In Using Unusual Data Types

Structures

- Have you used structures instead of naked variables to organize and manipulate groups of related data?
- Have you considered creating a class as an alternative to using a structure?

Global Data

- Are all variables local or class-scope unless they absolutely need to be global?
- Do variable naming conventions differentiate among local, class, and global data?
- Are all global variables documented?
- Is the code free of pseudoglobal data—mammoth objects containing a mishmash of data that's passed to every routine?
- Are access routines used instead of global data?
- Are access routines and data organized into classes?
- Do access routines provide a level of abstraction beyond the underlying data-type implementations?
- Are all related access routines at the same level of abstraction?

Pointers

- Are pointer operations isolated in routines?
- Are pointer references valid, or could the pointer be dangling?
- Does the code check pointers for validity before using them?
- Is the variable that the pointer references checked for validity before it's used?

- 983 Are pointers set to NULL after they're freed?
984 Does the code use all the pointer variables needed for the sake of
985 readability?
986 Are pointers in linked lists freed in the right order?
987 Does the program allocate a reserve parachute of memory so that it can shut
988 down gracefully if it runs out of memory?
989 Are pointers used only as a last resort, when no other method is available?
-

991

Key Points

- 992
 - 993 • Structures can help make programs less complicated, easier to understand,
994 and easier to maintain.
 - 995 • Whenever you consider using a structure, consider whether a class would
996 work better.
 - 997 • Pointers are error prone. Protect yourself by using access routines or classes
998 and defensive-programming practices.
 - 999 • Avoid global variables, not just because they're dangerous, but because you
1000 can replace them with something better.
 - 1001 • If you can't avoid global variables, work with them through access routines.
1002 Access routines give you everything that global variables give you, and
 more.

14

2 Organizing Straight-Line 3 Code

4 CC2E.COM/1465

5 Contents

6 14.1 Statements That Must Be in a Specific Order

7 14.2 Statements Whose Order Doesn't Matter

8 Related Topics

9 General control topics: Chapter 19

10 Code with conditionals: Chapter 15

11 Code with loops: Chapter 16

12 Scope of variables and objects: Section 10.4, "Scope"

13 THIS CHAPTER TURNS FROM a data-centered view of programming to a
14 statement-centered view. It introduces the simplest kind of control flow—putting
statements and blocks of statements in sequential order.

15 Although organizing straight-line code is a relatively simple task, some
16 organizational subtleties influence code quality, correctness, readability, and
17 maintainability.

18 14.1 Statements That Must Be in a Specific 19 Order

20 The easiest sequential statements to order are those in which the order counts.
21 Here's an example:

22 Java Example of Statements in Which Order Counts

```
23 data = ReadData();  
24 results = CalculateResultsFromData( data );  
25 PrintResults( results );
```

26 Unless something mysterious is happening with this code fragment, the
27 statement must be executed in the order shown. The data must be read before the
28 results can be calculated, and the results must be calculated before they can be
29 printed.

30 The underlying concept in this example is that of dependencies. The third
31 statement depends on the second, the second on the first. In this example, the
32 fact that one statement depends on another is obvious from the routine names. In
33 the code fragment below, the dependencies are less obvious:

34 Java Example of Statements in Which Order Counts, but Not Obviously

```
35 revenue.ComputeMonthly();  
36 revenue.ComputeQuarterly();  
37 revenue.ComputeAnnual();
```

38 In this case, the quarterly revenue calculation assumes that the monthly revenues
39 have already been calculated. A familiarity with accounting—or even common
40 sense—might tell you that quarterly revenues have to be calculated before annual
41 revenues. There is a dependency, but it's not obvious merely from reading the
42 code. In the code fragment below, the dependencies aren't obvious—they're
43 literally hidden:

44 Visual Basic Example of Statements in Which Order Dependencies Are 45 Hidden

```
46 ComputeMarketingExpense  
47 ComputeSalesExpense  
48 ComputeTravelExpense  
49 ComputePersonnelExpense  
50 DisplayExpenseSummary
```

51 Suppose that *ComputeMarketingExpense()* initializes the class member variables
52 that all the other routines put their data into. In such a case, it needs to be called
53 before the other routines. How could you know that from reading this code?
54 Because the routine calls don't have any parameters, you might be able to guess
55 that each of these routines accesses class data. But you can't know for sure from
56 reading this code.

57 KEY POINT

58 When statements have dependencies that require you to put them in a certain
59 order, take steps to make the dependencies clear. Here are some simple
guidelines for ordering statements:

60 *Organize code so that dependencies are obvious*

61 In the Visual Basic example presented above, *ComputeMarketingExpense()*
62 shouldn't initialize the class member variables. The routine names suggest that
63 *ComputeMarketingExpense()* is similar to *ComputeSalesExpense()*,
64 *ComputeTravelExpense()*, and the other routines except that it works with

65 marketing data rather than with sales data or other data. Having
66 *ComputeMarketingExpense()* initialize the member variable is an arbitrary
67 practice you should avoid. Why should initialization be done in that routine
68 instead of one of the other two? Unless you can think of a good reason, you
69 should write another routine, *InitializeExpenseData()* to initialize the member
70 variable. The routine's name is a clear indication that it should be called before
71 the other expense routines.

72 **Name routines so that dependencies are obvious**
73 In the example above, *ComputeMarketingExpense()* is misnamed because it does
74 more than compute marketing expenses; it also initializes member data. If you're
75 opposed to creating an additional routine to initialize the data, at least give
76 *ComputeMarketingExpense()* a name that describes all the functions it performs.
77 In this case, *ComputeMarketingExpenseAndInitializeMemberData()* would be an
78 adequate name. You might say it's a terrible name because it's so long, but the
79 name describes what the routine does and is not terrible. The routine itself is
80 terrible!

81 **CROSS-REFERENCE** For
82 details on using routines and
83 their parameters, see Chapter
84 5, "High-Level Design in
85 Construction."

Use routine parameters to make dependencies obvious

In the example above, since no data is passed between routines, you don't know whether any of the routines use the same data. By rewriting the code so that data is passed between the routines, you set up a clue that the execution order is important. Here's how the code would look:

Visual Basic Example of Data That Suggests an Order Dependency

```
86 InitializeExpenseData( expenseData )
87 ComputeMarketingExpense( expenseData )
88 ComputeSalesExpense( expenseData )
89 ComputeTravelExpense( expenseData )
90 ComputePersonnelExpense( expenseData )
91 DisplayExpenseSummary( expenseData )
```

Because all the routines use *expenseData*, you have a hint that they might be working on the same data and that the order of the statements might be important.

Visual Basic Example of Data and Routine Calls That Suggest an Order Dependency

```
96 expenseData = InitializeExpenseData( expenseData )
97 expenseData = ComputeMarketingExpense( expenseData )
98 expenseData = ComputeSalesExpense( expenseData )
99 expenseData = ComputeTravelExpense( expenseData )
100 expenseData = ComputePersonnelExpense( expenseData )
101
102 DisplayExpenseSummary( expenseData )
```

104
105
106

In this particular example, a better approach might be to convert the routines to functions that take *expenseData* as inputs and return updated *expenseData* as outputs, which makes it even clearer that there are order dependencies.

107 Data can also indicate that execution order isn't important. Here's an example:

108 Visual Basic Example of Data That Doesn't Indicate an Order 109 Dependency

110 ComputeMarketingExpense(marketingData)
111 ComputeSalesExpense(salesData)
112 ComputeTravelExpense(travelData)
113 ComputePersonnelExpense(personnelData)
114 DisplayExpenseSummary(marketingData, salesData, travelData, personnelData)
115 Since the routines in the first four lines don't have any data in common, the code
116 implies that the order in which they're called doesn't matter. Because the routine
117 in the fifth line uses data from each of the first four routines, you can assume that
118 it needs to be executed after the first four routines.

119 **Document unclear dependencies with comments**

120 **KEY POINT**
121 Try first to write code without order dependencies. Try second to write code that
122 makes dependencies obvious. If you're still concerned that an order dependency
123 isn't explicit enough, document it. Documenting unclear dependencies is one
124 aspect of documenting coding assumptions, which is critical to writing
125 maintainable, modifiable code. In the Visual Basic example, comments along
these lines would be helpful:

126 Visual Basic Example of Statements in Which Order Dependencies Are 127 Hidden but Clarified with Comments

128 ' Compute expense data. Each of the routines accesses the
129 ' member data expenseData. DisplayExpenseSummary
130 ' should be called last because it depends on data calculated
131 ' by the other routines.
132 expenseData = InitializeExpenseData(expenseData)
133 expenseData = ComputeMarketingExpense(expenseData)
134 expenseData = ComputeSalesExpense(expenseData)
135 expenseData = ComputeTravelExpense(expenseData)
136 expenseData = ComputePersonnelExpense(expenseData)
137 DisplayExpenseSummary(expenseData)

138 The code in this example doesn't use the techniques for making order
139 dependencies obvious. It's better to rely on such techniques rather than on
140 comments, but if you're maintaining tightly controlled code or you can't
141 improve the code itself for some other reason, use documentation to compensate
142 for code weaknesses.

143 ***Check for dependencies with assertions or error-handling code***
144 If the code is critical enough, you might use status variables and error-handling
145 code or assertions to document critical sequential dependencies. For example, in
146 the class's constructor, you might initialize a class member variable
147 *isExpenseDataInitialized* to *FALSE*. Then in *InitializeExpenseData()*, you can
148 set *isExpenseDataInitialized* to *TRUE*. Each function that depends on
149 *expenseData* being initialized can then check whether *isExpenseDataInitialized*
150 has been set to *TRUE* before performing additional operations on *expenseData*.
151 Depending on how extensive the dependencies are, you might also need
152 variables like *isMarketingExpenseComputed*, *isSalesExpenseComputed*, and so
153 on.

154 This technique creates new variables, new initialization code, and new error-
155 checking code, all of which create additional possibilities for error. The benefits
156 of this technique should be weighed against the additional complexity and
157 increased chance of secondary errors that this technique creates.

158 **14.2 Statements Whose Order Doesn't 159 Matter**

160 You might encounter cases in which it seems as if the order of a few statements
161 or a few blocks of code doesn't matter at all. One statement doesn't depend on,
162 or logically follow, another statement. But ordering affects readability,
163 performance, and maintainability, and in the absence of execution-order
164 dependencies, you can use secondary criteria to determine the order of
165 statements or blocks of code. The guiding principle is the Principle of Proximity:
166 *Keep related actions together.*

167 **Making Code Read from Top to Bottom**

168 As a general principle, make the program read from top to bottom rather than
169 jumping around. Experts agree that top-to-bottom order contributes most to
170 readability. Simply making the control flow from top to bottom at run time isn't
171 enough. If someone who is reading your code has to search the whole program to
172 find needed information, you should reorganize the code. Here's an example:

173 **C++ Example of Bad Code That Jumps Around**

```
174 MARKETING_DATA *marketingData = new MARKETING_DATA;  
175 SALES_DATA *salesData = new SALES_DATA;  
176 TRAVEL_DATA *travelData = new TRAVEL_DATA;  
177  
178 travelData.ComputeQuarterly();
```

```
179 salesData.ComputeQuarterly();  
180 marketingData.ComputeQuarterly();  
181  
182 salesData.ComputeAnnual();  
183 marketingData.ComputeAnnual();  
184 travelData.ComputeAnnual();  
185  
186 salesData.Print();  
187 delete salesData;  
188 travelData.Print();  
189 delete travelData;  
190 marketingData.Print();  
191 delete marketingData;
```

Suppose that you want to determine how *marketingData* is calculated. You have to start at the last line and track all references to *marketingData* back to the first line. *marketingData* is used in only a few other places, but you have to keep in mind how *marketingData* is used everywhere between the first and last references to it. In other words, you have to look at and think about every line of code in this fragment to figure out how *marketingData* is calculated. And of course this example is simpler than code you see in life-size systems. Here's the same code with better organization:

C++ Example of Good, Sequential Code That Reads from Top to Bottom

```
200 MARKETING_DATA *marketingData = new MARKETING_DATA;  
201 marketingData.ComputeQuarterly();  
202 marketingData.ComputeAnnual();  
203 marketingData.Print();  
204 delete marketingData;  
205  
206 SALES_DATA *salesData = new SALES_DATA;  
207 salesData.ComputeQuarterly();  
208 salesData.ComputeAnnual();  
209 salesData.Print();  
210 delete salesData;  
211  
212 TRAVEL_DATA *travelData = new TRAVEL_DATA;  
213 travelData.ComputeQuarterly();  
214 travelData.ComputeAnnual();  
215 travelData.Print();  
216 delete travelData;
```

217
218 **CROSS-REFERENCE** A more technical definition of “live” variables is given in “Measuring the Live Time of a Variable” in Section 10.4.
219
220
221
222

This code is better in several ways. References to each object are kept close together; they're “localized.” The number of lines of code in which the objects are “live” is small. And perhaps most important, the code now looks as if it could be broken into separate routines for marketing, sales, and travel data. The first code fragment gave no hint that such a decomposition was possible.

223

224 **CROSS-REFERENCE** If
225 you follow the Pseudocode
226 Programming Process, your
code will automatically be
grouped into related
227 statements. For details on the
228 process, see Chapter 9, “The
229 Pseudocode Programming
230 Process.”

231

232

233

234

235

236

237

238

Grouping Related Statements

Put related statements together. They can be related because they operate on the same data, perform similar tasks, or depend on each other’s being performed in order.

An easy way to test whether related statements are grouped well is to print out a listing of your routine and then draw boxes around the related statements. If the statements are ordered well, you’ll get a picture like that shown in Figure 14-1, in which the boxes don’t overlap.

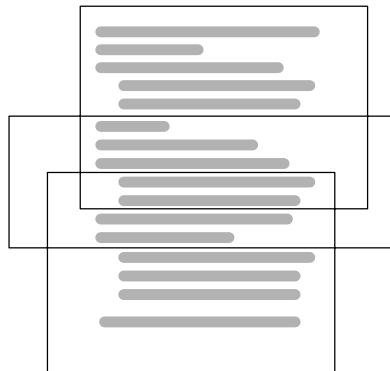


F14xx01

Figure 14-1

If the code is well organized into groups, boxes drawn around related sections don’t overlap. They might be nested.

If statements aren’t ordered well, you’ll get a picture something like that shown in Figure 14-2, in which the boxes do overlap. If you find that your boxes overlap, reorganize your code so that related statements are grouped better.



239

240

241

242

243

244

245

246

CC2E.COM/1472

247

248

249

250

251

252

253

254

255

256

257

258

F14xx02

Figure 14-2

If the code is organized poorly, boxes drawn around related sections overlap.

Once you've grouped related statements, you might find that they're strongly related and have no meaningful relationship to the statements that precede or follow them. In such a case, you might want to put the strongly related statements into their own routine.

Checklist: Organizing Straight-Line Code

- Does the code make dependencies among statements obvious?
 - Do the names of routines make dependencies obvious?
 - Do parameters to routines make dependencies obvious?
 - Do comments describe any dependencies that would otherwise be unclear?
 - Have housekeeping variables been used to check for sequential dependencies in critical sections of code?
 - Does the code read from top to bottom?
 - Are related statements grouped together?
 - Have relatively independent groups of statements been moved into their own routines?
-

Key Points

- The strongest principle for organizing straight-line code is order dependencies.

- 262 • Dependencies should be made obvious through the use of good routine
263 names, parameter lists, comments, and—if the code is critical enough—
264 housekeeping variables.
265 • If code doesn't have order dependencies, keep related statements as close
266 together as possible.

15

2 Using Conditionals

3 CC2E.COM/1538

4 Contents

5 15.1 *if* Statements
6 15.2 *case* Statements

7 Related Topics

8 Taming Deep Nesting: Section 19.4

9 General control issues: Chapter 19

10 Code with loops: Chapter 16

11 Straight-line code: Chapter 14

12 Relationship between control structures and data types: Section 10.7

13 A CONDITIONAL IS A STATEMENT that controls the execution of other
14 statements; execution of the other statements is “conditioned” on statements such
15 as *if*, *else*, *case*, and *switch*. Although it makes sense logically to refer to loop
16 controls such as *while* and *for* as conditionals too, by convention they’ve been
 treated separately. Chapter 16, on loops, will examine *while* and *for* statements.

17 15.1 *if* Statements

18 Depending on the language you’re using, you might be able to use any of several
19 kinds of *if* statements. The simplest is the plain *if* or *if-then* statement. The *if-*
20 *then-else* is a little more complex, and chains of *if-then-else-if* are the most
21 complex.

22 Plain *if-then* Statements

23 Follow these guidelines when writing *if* statements:

KEY POINT

24 ***Write the nominal path through the code first; then write the unusual cases***
25 Write your code so that the normal path through the code is clear. Make sure that
26 the rare cases don't obscure the normal path of execution. This is important for
27 both readability and performance.

28 ***Make sure that you branch correctly on equality***
29 Using > instead of >= or < instead of <= is analogous to making an off-by-one
30 error in accessing an array or computing a loop index. In a loop, think through
31 the endpoints to avoid an off-by-one error. In a conditional statement, think
32 through the equals case to avoid an off-by-one error.

33 ***Put the normal case after the if rather than after the else***
34 Put the case you normally expect to process first. This is in line with the general
35 principle of putting code that results from a decision as close as possible to the
36 decision. Here's a code example that does a lot of error processing, haphazardly
37 checking for errors along the way:

Visual Basic Example of Code That Processes a Lot of Errors

Haphazardly

```
40                   OpenFile( inputFile, status )
41                   If ( status = Status_Error ) Then
42                     errorType = FileOpenError
43                   Else
44                     nominal case    ReadFile( inputFile, fileData, status )
45                        If ( status = Status_Success ) Then
46                        nominal case    SummarizeFileData( fileData, summaryData, status )
47                        If ( status = Status_Error ) Then
48                        error case      errorType = ErrorType_DataSummaryError
49                        Else
50                        nominal case    PrintSummary( summaryData )
51                        SaveSummaryData( summaryData, status )
52                        If ( status = Status_Error ) Then
53                        error case      errorType = ErrorType_SummarySaveError
54                        Else
55                        nominal case    UpdateAllAccounts()
56                        EraseUndoFile()
57                        errorType = ErrorType_None
58                        End If
59                        End If
60                        Else
61                        errorType = ErrorType_FileReadError
62                        End If
63                        End If
```

64 This code is hard to follow because the nominal cases and the error cases are all
65 mixed together. It's hard to find the path that is normally taken through the code.

66 In addition, because the error conditions are sometimes processed in the *if* clause
 67 rather than the *else* clause, it's hard to figure out which *if* test the normal case
 68 goes with. In the rewritten code below, the normal path is consistently coded
 69 first, and all the error cases are coded last. This makes it easier to find and read
 70 the nominal case.

71 Visual Basic Example of Code That Processes a Lot of Errors 72 Systematically

```
73 OpenFile( inputFile, status )
74 If status = Status_Success Then
75     nominal case
76         ReadFile( inputFile, fileData, status )
77             If status = Status_Success Then
78                 nominal case
79                     SummarizeFileData( fileData, summaryData, status )
80                         If status = Status_Success Then
81                             nominal case
82                                 PrintSummary( summaryData )
83                                     SaveSummaryData( summaryData, status )
84                                         If status = Status_Success Then
85                                             nominal case
86                                                 UpdateAllAccounts()
87                                                 EraseUndoFile()
88                                                 errorType = ErrorType_None
89                                         Else
90                                             errorType = ErrorType_SummarySaveError
91                                         End If
92                                         Else
93                                             errorType = ErrorType_DataSummaryError
94                                         End If
95                                         Else
96                                             errorType = ErrorType_FileReadError
97                                         End If
98                                         Else
99                                             errorType = ErrorType_FileOpenError
100                                         End If
101 
```

In the revised example, you can read the main flow of the *if* tests to find the normal case. The revision puts the focus on reading the main flow rather than on wading through the exceptional cases. The code is easier to read overall. The stack of error conditions at the bottom of the nest is a sign of well-written error-processing code.

102 *Follow the if clause with a meaningful statement*

103 Sometimes you see code like the next example, in which the *if* clause is null.

104 CODING HORROR

Java Example of a Null if Clause

```
105 if ( SomeTest )
106     ;
107 else {
```

```
108  
109  
110  
111 CROSS-REFERENCE One key to writing an effective if statement is writing the right boolean expression to control it. For details on using  
112 boolean expressions  
113 effectively, see Section 19.1,  
114 "Boolean Expressions."  
115  
116
```

```
117  
118  
119  
120
```

HARD DATA

```
121  
122  
123  
124
```

```
125  
126  
127  
128  
129  
130
```

```
131  
132  
133  
134  
135  
136  
137  
138  
139  
140
```

```
141  
142  
143  
144
```

```
// do something  
...  
}
```

Most experienced programmers would avoid code like this if only to avoid the work of coding the extra null line and the *else* line. It looks silly and is easily improved by negating the predicate in the *if* statement, moving the code from the *else* clause to the *if* clause, and eliminating the *else* clause. Here's how the code would look after such a change:

Java Example of a Converted Null *if* Clause

```
if ( ! SomeTest ) {  
    // do something  
    ...  
}
```

Consider the else clause

If you think you need a plain *if* statement, consider whether you don't actually need an *if-then-else* statement. A classic General Motors analysis found that 50 to 80 percent of *if* statements should have had an *else* clause (Elshoff 1976).

One option is to code the *else* clause—with a null statement if necessary—to show that the *else* case has been considered. Coding null *elses* just to show that that case has been considered might be overkill, but at the very least, take the *else* case into account. When you have an *if* test without an *else*, unless the reason is obvious, use comments to explain why the *else* clause isn't necessary. Here's an example:

Java Example of a Helpful, Commented *else* Clause

```
// if color is valid  
if ( COLOR_MIN <= color && color <= COLOR_MAX ) {  
    // do something  
    ...  
}  
else {  
    // else color is invalid  
    // screen not written to -- safely ignore command  
}
```

Test the else clause for correctness

When testing your code, you might think that the main clause, the *if*, is all that needs to be tested. If it's possible to test the *else* clause, however, be sure to do that.

145
146
147
148
149

Check for reversal of the if and else clauses

A common mistake in programming *if-thens* is to flip-flop the code that's supposed to follow the *if* clause and the code that's supposed to follow the *else* clause or to get the logic of the *if* test backward. Check your code for this common error.

150

Chains of *if-then-else* Statements

151
152
153

In languages that don't support *case* statements—or that support them only partially—you will often find yourself writing chains of *if-then-else* tests. For example, the code to categorize a character might use a chain like this one:

154 **CROSS-REFERENCE** For
155 more details on simplifying
156 complicated expressions, see
157 Section 19.1, "Boolean
158 Expressions."

159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181

C++ Example of Using an *if-then-else* Chain to Categorize a Character

```
if ( inputCharacter < SPACE ) {  
    characterType = CharacterType_ControlCharacter;  
}  
  
else if (  
    inputCharacter == ' ' ||  
    inputCharacter == ',' ||  
    inputCharacter == '.' ||  
    inputCharacter == '!' ||  
    inputCharacter == '(' ||  
    inputCharacter == ')' ||  
    inputCharacter == ':' ||  
    inputCharacter == ';' ||  
    inputCharacter == '?' ||  
    inputCharacter == '-'  
) {  
    characterType = CharacterType_Punctuation;  
}  
  
else if ( '0' <= inputCharacter && inputCharacter <= '9' ) {  
    characterType = CharacterType_Digit;  
}  
  
else if (  
    ( 'a' <= inputCharacter && inputCharacter <= 'z' ) ||  
    ( 'A' <= inputCharacter && inputCharacter <= 'Z' )  
) {  
    characterType = CharacterType_Letter;  
}
```

Here are some guidelines to follow when writing such *if-then-else* chains:

182
183
184

Simplify complicated tests with boolean function calls

One reason the code above is hard to read is that the tests that categorize the character are complicated. To improve readability, you can replace them with

185 calls to boolean functions. Here's how the code above looks when the tests are
186 replaced with boolean functions:

C++ Example of an *if-then-else* Chain That Uses Boolean Function Calls

```
188 if ( IsControl( inputCharacter ) ) {  
189     characterType = CharacterType_ControlCharacter;  
190 }  
191 else if ( IsPunctuation( inputCharacter ) ) {  
192     characterType = CharacterType_Punctuation;  
193 }  
194 else if ( IsDigit( inputCharacter ) ) {  
195     characterType = CharacterType_Digit;  
196 }  
197 else if ( IsLetter( inputCharacter ) ) {  
198     characterType = CharacterType_Letter;  
199 }
```

200 ***Put the most common cases first***

201 By putting the most common cases first, you minimize the amount of exception-
202 case handling code someone has to read to find the usual cases. You improve
203 efficiency because you minimize the number of tests the code does to find the
204 most common cases. In the example above, letters would be more common than
205 punctuation but the test for punctuation is made first. Here's the code revised so
206 that it tests for letters first:

C++ Example of Testing the Most Common Case First

```
208 This test, the most common,  
209         is now done first.  
210  
211 if ( IsLetter( inputCharacter ) ) {  
212     characterType = CharacterType_Letter;  
213 }  
214 else if ( IsPunctuation( inputCharacter ) ) {  
215     characterType = CharacterType_Punctuation;  
216 }  
217 This test, the least common,  
218         is now done last  
219 else if ( IsDigit( inputCharacter ) ) {  
220     characterType = CharacterType_Digit;  
221 }  
222 else if ( IsControl( inputCharacter ) ) {  
223     characterType = CharacterType_ControlCharacter;  
224 }
```

220 ***Make sure that all cases are covered***

221 Code a final *else* clause with an error message or assertion to catch cases you
222 didn't plan for. This error message is intended for you rather than for the user, so
223 word it appropriately. Here's how you can modify the character-classification
224 example to perform an "other cases" test:

225 **CROSS-REFERENCE** This
 226 is also a good example of
 227 how you can use a chain of
 228 *if-then-else* tests instead of
 229 deeply nested code. For
 230 details on this technique, see
 231 Section 19.4, “Taming
 231 Dangerously Deep Nesting.”

232
 233
 234
 235
 236
 237
 238
 239
 240

241
 242
 243
 244
 245
 246
 247

248
 249
 250
 251
 252
 253
 254
 255
 256
 257
 258
 259
 260
 261

262

C++ Example of Using the Default Case to Trap Errors

```
if ( IsLetter( inputCharacter ) ) {
    characterType = CharacterType_Letter;
}
else if ( IsPunctuation( inputCharacter ) ) {
    characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
    characterType = CharacterType_Digit;
}
else if ( IsControl( inputCharacter ) ) {
    characterType = CharacterType_ControlCharacter;
}
else {
    DisplayInternalError( "Unexpected type of character detected." );
}
```

Replace *if-then-else* chains with other constructs if your language supports them

A few languages—Visual Basic and Ada, for example—provide *case* statements that support use of strings, enums, and logical functions. Use them. They are easier to code and easier to read than *if-then-else* chains. Here’s how the code for classifying character types would be written using a *case* statement in Visual Basic;

Visual Basic Example of Using a *case* Statement Instead of an *if-then-else* Chain

```
Select Case inputCharacter
Case "a" To "z"
    characterType = CharacterType_Letter
Case " ", ",", ".", "!", "(", ")", ":", ";", "?", "-"
    characterType = CharacterType_Punctuation
Case "0" To "9"
    characterType = CharacterType_Digit
Case FIRST_CONTROL_CHARACTER To LAST_CONTROL_CHARACTER
    characterType = CharacterType_Control
Case Else
    DisplayInternalError( "Unexpected type of character detected." )
End Select
```

15.2 *case* Statements

The *case* or *switch* statement is a construct that varies a great deal from language to language. C++ and Java support *case* only for ordinal types taken one value at

265 a time. Visual Basic supports *case* for ordinal types and has powerful shorthand
266 notations for expressing ranges and combinations of values. Many scripting
267 languages don't support *case* statements at all.

268 The following sections present guidelines for using *case* statements effectively.

269 **Choosing the Most Effective Ordering of Cases**

270 You can choose from among a variety of ways to organize the cases in a *case*
271 statement. If you have a small *case* statement with three options and three
272 corresponding lines of code, the order you use doesn't matter much. If you have
273 a long *case* statement—for example, a *case* statement in an event-driven
274 program—order is significant. Here are some ordering possibilities:

275 ***Order cases alphabetically or numerically***

276 If cases are equally important, putting them in A-B-C order improves readability.
277 A specific case is easy to pick out of the group.

278 ***Put the normal case first***

279 If you have one normal case and several exceptions, put the normal case first.
280 Indicate with comments that it's the normal case and that the others are unusual.

281 ***Order cases by frequency***

282 Put the most frequently executed cases first and the least frequently executed
283 last. This approach has two advantages. First, human readers can find the most
284 common cases easily. Readers scanning the list for a specific case are likely to be
285 interested in one of the most common cases. Putting the common ones at the top
286 of the code makes the search quicker.

287 In this instance, achieving better human readability also supports faster machine
288 execution. Each case represents a test that the machine performs at run time. If
289 you have 12 cases and the last one is the one that needs to be executed, the
290 machine executes the equivalent of 12 *if* tests before it finds the right one. By
291 putting the common cases first, you reduce the number of tests the machine must
292 perform and thus improve the efficiency of your code.

293 **Tips for Using *case* Statements**

294 Here are several tips for using *case* statements:

295 ***Keep the actions of each case simple***

296 Keep the code associated with each case short. Short code following each case
297 helps make the structure of the *case* statement clear. If the actions performed for
298 a case are complicated, write a routine and call the routine from the case rather
299 than putting the code into the case itself.

300
301
302
303
304

CODING HORROR

305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326

327 **CROSS-REFERENCE** In
328 contrast to this advice,
329 sometimes you can improve
330 readability by assigning a
331 complicated expression to a
332 well-named boolean variable
333 or function. For details, see
334 “Making Complicated
335 Expressions Simple” in
Section 19.1.

336
337
338

Don’t make up phony variables in order to be able to use the case statement

A *case* statement should be used for simple data that’s easily categorized. If your data isn’t simple, use chains of *if-then-elses* instead. Phony variables are confusing, and you should avoid them. Here’s an example of what not to do:

Java Example of Creating a Phony *case* Variable—Bad Practice

```
action = userCommand[ 0 ];
switch ( action ) {
    case 'c':
        Copy();
        break;
    case 'd':
        DeleteCharacter();
        break;
    case 'f':
        Format();
        break;
    case 'h':
        Help();
        break;
    ...
    default:
        HandleUserInputError( ErrorType.InvalidUserCommand );
}
```

The variable that controls the *case* statement is *action*. In this case, *action* is created by peeling off the first character of the *userCommand* string, a string that was entered by the user.

This troublemaking code is from the wrong side of town and invites problems. In general, when you manufacture a variable to use in a *case* statement, the real data might not map onto the *case* statement the way you want it to. In this example, if the user types “copy,” the *case* statement peels off the first “c” and correctly calls the *Copy()* routine. On the other hand, if the user types “cement overshoes,” “clambake,” or “cellulite,” the *case* statement also peels off the “c” and calls *Copy()*. The test for an erroneous command in the *case* statement’s *else* clause won’t work very well because it will miss only erroneous first letters rather than erroneous commands.

This code should use a chain of *if-then-else-if* tests to check the whole string rather than making up a phony variable. A virtuous rewrite of the code looks like this:

Java Example of Using *if-then-elses* Instead of a Phony *case* Variable—Good Practice

```
339  
340  
341     if ( UserCommand.equals( COMMAND_STRING_COPY ) ) {  
342         Copy();  
343     }  
344     else if ( UserCommand.equals( COMMAND_STRING_DELETE ) ) {  
345         DeleteCharacter();  
346     }  
347     else if ( UserCommand.equals( COMMAND_STRING_FORMAT ) ) {  
348         Format();  
349     }  
350     else if ( UserCommand.equals( COMMAND_STRING_HELP ) ) {  
351         Help();  
352     }  
353     ...  
354     else {  
355         HandleUserInputError( ErrorType_InvalidCommandInput );  
356     }
```

357 *Use the default clause only to detect legitimate defaults*

358 You might sometimes have only one case remaining and decide to code that case
359 as the default clause. Though sometimes tempting, that's dumb. You lose the
360 automatic documentation provided by *case*-statement labels, and you lose the
361 ability to detect errors with the default clause.

362 Such *case* statements break down under modification. If you use a legitimate
363 default, adding a new case is trivial—you just add the case and the
364 corresponding code. If you use a phony default, the modification is more
365 difficult. You have to add the new case, possibly making it the new default, and
366 then change the case previously used as the default so that it's a legitimate case.
367 Use a legitimate default in the first place.

368 *Use the default clause to detect errors*

369 If the default clause in a *case* statement isn't being used for other processing and
370 isn't supposed to occur, put a diagnostic message in it. An example follows.

**371 *Java Example of Using the Default Case to Detect Errors—Good*
372 *Practice***

```
373     switch ( commandShortcutLetter ) {  
374         case 'a':  
375             PrintAnnualReport();  
376             break;  
377         case 'p':  
378             // no action required, but case was considered  
379             break;
```

```

380     case 'q':
381         PrintQuarterlyReport();
382         break;
383     case 's':
384         PrintSummaryReport();
385         break;
386     default:
387         DisplayInternalError( "Internal Error 905: Call customer support." );
388 }
```

Messages like this are useful in both debugging and production code. Most users prefer a message like “Internal Error: Please call customer support” to a system crash—or worse, subtly incorrect results that look right until the user’s boss checks them.

393 If the default clause is used for some purpose other than error detection, the
 394 implication is that every case selector is correct. Double-check to be sure that
 395 every value that could possibly enter the *case* statement would be legitimate. If
 396 you come up with some that wouldn’t be legitimate, rewrite the statements so
 397 that the default clause will check for errors.

398 ***In C++ and Java, avoid dropping through the end of a case statement***
 399 C-like languages (C, C++, and Java) don’t automatically break out of each case.
 400 Instead, you have to code the end of each case explicitly. If you don’t code the
 401 end of a case, the program drops through the end and executes the code for the
 402 next case. This can lead to some particularly egregious coding practices,
 403 including the following horrible example:

404 **CROSS-REFERENCE** This
 405 code’s formatting makes it
 406 look better than it is. For
 407 details on how to use
 408 formatting to make good
 409 code look good and bad code
 410 look bad, see “Endline
 411 Layout” in “Endline Layout”
 412 in Section 31.3 and the rest
 413 of Chapter 31.

414
 415
 416
 417
 418
 419
 420
 421

C++ Example of Abusing the **case** Statement

```

switch ( InputVar )
{
    case 'A': if ( test )
        {
            // statement 1
            // statement 2
        case 'B':   // statement 3
            // statement 4
            ...
        }
        ...
    break;
    ...
}
```

This practice is bad because it intermingles control constructs. Nested control constructs are hard enough to understand; overlapping constructs are all but impossible. Modifications of case ‘A’ or case ‘B’ will be harder than brain

422 surgery, and it's likely that the cases will need to be cleaned up before any
423 modifications will work. You might as well do it right the first time. In general,
424 it's a good idea to avoid dropping through the end of a *case* statement.

425 ***In C++, clearly and unmistakably identify flow-throughs at the end of a***
426 ***case statement***

427 If you intentionally write code to drop through the end of a case, comment the
428 place at which it happens clearly and explain why it needs to be coded that way.

C++ Example of Documenting Falling Through the End of a *case Statement*

```
431 switch ( errorDocumentationLevel ) {  
432     case DocumentationLevel_Full:  
433         DisplayErrorDetails( errorNumber );  
434         // FALLTHROUGH -- Full documentation also prints summary comments  
435  
436     case DocumentationLevel_Summary:  
437         DisplayErrorSummary( errorNumber );  
438         // FALLTHROUGH -- Summary documentation also prints error number  
439  
440     case DocumentationLevel_NumberOnly:  
441         DisplayErrorNumber( errorNumber );  
442         break;  
443  
444     default:  
445         DisplayInternalError( "Internal Error 905: Call customer support." );  
446 }
```

447 This technique is useful about as often as you find someone who would rather
448 have a used Pontiac Aztek than a new Corvette. Generally, code that falls
449 through from one case to another is an invitation to make mistakes as the code is
450 modified and should be avoided.

CC2E.COM/1545

451

CHECKLIST: Conditionals

452 ***if-then Statements***

- 453 Is the nominal path through the code clear?
- 454 Do *if-then* tests branch correctly on equality?
- 455 Is the *else* clause present and documented?
- 456 Is the *else* clause correct?
- 457 Are the *if* and *else* clauses used correctly—not reversed?
- 458 Does the normal case follow the *if* rather than the *else*?

459 *if-then-else-if* Chains

- 460 Are complicated tests encapsulated in boolean function calls?
- 461 Are the most common cases tested first?
- 462 Are all cases covered?
- 463 Is the *if-then-else-if* chain the best implementation—better than a *case* statement?
- 464

465 *case* Statements

- 466 Are cases ordered meaningfully?
- 467 Are the actions for each case simple—calling other routines if necessary?
- 468 Does the *case* statement test a real variable, not a phony one that's made up solely to use and abuse the *case* statement?
- 469
- 470 Is the use of the default clause legitimate?
- 471 Is the default clause used to detect and report unexpected cases?
- 472 In C, C++, or Java, does the end of each case have a *break*?
- 473
-

474 Key Points

- 475 • For simple *if-elses*, pay attention to the order of the *if* and *else* clauses, especially if they process a lot of errors. Make sure the nominal case is clear.
- 476
- 477 • For *if-then-else* chains and *case* statements, choose an order that maximizes readability.
- 478
- 479 • Use the default clause in a *case* statement or the last *else* in a chain of *if-then-elses* to trap errors.
- 480
- 481 • All control constructs are not created equal. Choose the control construct that's most appropriate for each section of code.
- 482

16

2 Controlling Loops

3 CC2E.COM/1609

4 16.1 Selecting the Kind of Loop

5 16.2 Controlling the Loop

6 16.3 Creating Loops Easily—from the Inside Out

7 16.4 Correspondence Between Loops and Arrays

8 **Related Topics**

9 Taming Deep Nesting: Section 19.4

10 General control issues: Chapter 19

11 Code with conditionals: Chapter 15

12 Straight-line code: Chapter 14

13 Relationship between control structures and data types: Section 10.7

14 “LOOP” IS AN INFORMAL TERM that refers to any kind of iterative control
15 structure—any structure that causes a program to repeatedly execute a block of
16 code. Common loop types are *for*, *while*, and *do-while* in C++ and Java and
17 *For-Next*, *While-Wend*, and *Do-Loop-While* in Visual Basic. Using loops is one
18 of the most complex aspects of programming; knowing how and when to use
19 each kind of loop is a decisive factor in constructing high-quality software.20

16.1 Selecting the Kind of Loop

21 In most languages, you’ll use a few kinds of loops.

- 22
- 23 • The counted loop is performed a specific number of times, perhaps one time
for each employee.
 - 24 • The continuously evaluated loop doesn’t know ahead of time how many
25 times it will be executed and tests whether it has finished on each iteration.
26 For example, it runs while money remains, until the user selects quit, or
27 until it encounters an error.

- 28 • The endless loop executes forever once it has started. It's the kind you find
29 in embedded systems such as pacemakers, microwave ovens, and cruise
30 controls.
31 • The iterator loop that performs its action once for each element in a
32 container class

33 The kinds of loops are differentiated first by flexibility—whether the loop
34 executes a specified number of times or whether it tests for completion on each
35 iteration.

36 The kinds of loops are also differentiated by the location of the test for com-
37 pletion. You can put the test at the beginning, the middle, or the end of the loop.
38 This characteristic tells you whether the loop executes at least once. If the loop
39 is tested at the beginning, its body isn't necessarily executed. If the loop is tested
40 at the end, its body is executed at least once. If the loop is tested in the middle,
41 the part of the loop that precedes the test is executed at least once, but the part of
42 the loop that follows the test isn't necessarily executed at all.

43 Flexibility and the location of the test determine the kind of loop to choose as a
44 control structure. Table 16-1 shows the kinds of loops in several languages and
45 describes each loop's flexibility and test location.

46 **Table 16-1. The Kinds of Loops**

Language	Kind of Loop	Flexibility	Test Location
Visual Basic	<i>For-Next</i>	rigid	beginning
	<i>While-Wend</i>	flexible	beginning
	<i>Do-Loop-While</i>	flexible	beginning or end
	<i>For-Each</i>	rigid	beginning
C, C++, C#, Java	<i>for</i>	flexible	beginning
	<i>while</i>	flexible	beginning
	<i>do-while</i>	flexible	end
	<i>foreach</i> *	rigid	beginning

47 * Available only in C# at the time of this writing.

48 **When to Use a *while* Loop**

49 Novice programmers sometimes think that a *while* loop is continuously
50 evaluated and that it terminates the instant the *while* condition becomes false,
51 regardless of which statement in the loop is being executed (Curtis et al. 1986).
52 Although it's not quite that flexible, a *while* loop is a flexible loop choice. If you
53 don't know ahead of time exactly how many times you'll want the loop to
54 iterate, use a *while* loop. Contrary to what some novices think, the test for the
55 loop exit is performed only once each time through the loop, and the main issue

56 with respect to *while* loops is deciding whether to test at the beginning or the
57 end of the loop.

58 **Loop with Test at the Beginning**

59 For a loop that tests at the beginning, you can use a *while* loop in C++, C, Java,
60 Visual Basic, and most other languages. You can emulate a *while* loop in other
61 languages.

62 **Loop with Test at the End**

63 You might occasionally have a situation in which you want a flexible loop but
64 the loop needs to execute at least one time. In such a case, you can use a *while*
65 loop that is tested at its end. You can use *do-while* in C++, C, and Java, *Do-*
66 *Loop-While* in Visual Basic, or you can emulate end-tested loops in other
67 languages.

68 **When to Use a loop-with-exit Loop**

69 A loop-with-exit loop is a loop in which the exit condition appears in the middle
70 of the loop rather than at the beginning or at the end. The *loop-with-exit* loop is
71 available explicitly in Visual Basic, and you can emulate it with the structured
72 constructs *while* and *break* in C++, C, and Java or with *gotos* in other languages.

73 **Normal loop-with-exit Loops**

74 A loop-with-exit loop usually consists of the loop beginning, the loop body
75 including an exit condition, and the loop end, as in this Visual Basic example:

76 **Visual Basic Example of a Generic loop-with-exit Loop**

```
77                   Do  
78                    Statements  
79                    ...  
80                    If ( some exit condition ) Then Exit Do  
81                   More statements  
82                    ...  
83                   Loop
```

84 The typical use of a loop-with-exit loop is for the case in which testing at the
85 beginning or at the end of the loop requires coding a loop-and-a-half. Here's a
86 C++ example of a case that warrants a loop-with-exit loop but doesn't use one:

85 **C++ Example of Duplicated Code That Will Break Down Under** 86 **Maintenance (A Place to Use a loop-with- exit Loop)**

```
87                   // Compute scores and ratings.  
88                   score = 0;  
89                   These lines appear here...  
90                   GetNextRating( &ratingIncrement );  
91                   rating = rating + ratingIncrement;  
                  while ( ( score < targetScore ) && ( ratingIncrement != 0 ) ) {
```

```
92  
93  
94     ...and are repeated here.  
95  
96  
97  
98  
99  
100  
101  
102
```

103 **CROSS-REFERENCE** The *FOREVER* macro used at the
104 top of this loop is equivalent
105 to *for(;;)* and is described
106 later in this chapter.

```
107  
108  
109  
110  
111 This is the loop-exit condition.  
112  
113  
114  
115  
116  
117  
118
```

```
GetNextScore( &scoreIncrement );  
score = score + scoreIncrement;  
GetNextRating( &ratingIncrement );  
rating = rating + ratingIncrement;  
}
```

The two lines of code at the top of this example are repeated in the last two lines of code of the *while* loop. During modification, you can easily forget to keep the two sets of lines parallel. Another programmer modifying the code probably won't even realize that the two sets of lines are supposed to be modified in parallel. Either way, the result will be errors arising from incomplete modifications. Here's how you can rewrite the code more clearly:

C++ Example of a loop-with-exit Loop That's Easier to Maintain

```
// Compute scores and ratings. The loop uses a FOREVER macro  
// and a break statement to emulate a loop-with-exit loop.  
score = 0;  
FOREVER {  
    GetNextRating( &ratingIncrement );  
    rating = rating + ratingIncrement;  
  
    if ( !( ( score < targetScore ) && ( ratingIncrement != 0 ) ) ) {  
        break;  
    }  
  
    GetNextScore( &scoreIncrement );  
    score = score + scoreIncrement;  
}
```

Here's how the same code is written in Visual Basic:

Visual Basic Example of a /loop-with-exit Loop

```
' Compute scores and ratings  
score = 0  
Do  
    GetNextRating( ratingIncrement )  
    rating = rating + ratingIncrement  
  
    If ( not ( score < targetScore and ratingIncrement <> 0 ) ) Then Exit Do  
  
    GetNextScore( ScoreIncrement )  
    score = score + scoreIncrement  
Loop
```

Here are a couple of fine points to consider when you use this kind of loop:

132 **CROSS-REFERENCE** Details on exit conditions are
133 presented later in this
134 chapter. For details on using
135 comments with loops, see
136 “Commenting Control
137 Structures” in Section 32.5.

138 **HARD DATA**

139
140
141
142
143
144
145
146

147
148
149
150

151
152
153

154 **CODING HORROR**

155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171

- Put all the exit conditions in one place. Spreading them around practically guarantees that one exit condition or another will be overlooked during debugging, modification, or testing.
- Use comments for clarification. If you use the loop-with-exit loop technique in a language that doesn’t support it directly use comments to make what you’re doing clear.

The loop-with-exit loop is a one-entry, one-exit, structured control construct, and it is the preferred kind of loop control (Software Productivity Consortium 1989). It has been shown to be easier to understand than other kinds of loops. A study of student programmers compared this kind of loop with those that exited at either the top or the bottom (Soloway, Bonar, and Ehrlich 1983). Students scored 25 percent higher on a test of comprehension when loop-with-exit loops were used, and the authors of the study concluded that the loop-with-exit structure more closely models the way people think about iterative control than other loop structures do.

In common practice, the loop-with-exit loop isn’t widely used yet. The jury is still locked in a smoky room arguing about whether it’s a good practice for production code. Until the jury is in, the loop-with-exit is a good technique to have in your programmer’s toolbox—as long as you use it carefully.

Abnormal loop-with-exit Loops

Another kind of loop-with-exit loop that’s used to avoid a loop-and-a-half is shown here:

C++ Example of Entering the Middle of a Loop with a *goto*—Bad Practice

```
156 goto Start;  
157 while ( expression ) {  
158     // do something  
159     ...  
160  
161     Start:  
162  
163     // do something else  
164     ...  
165 }
```

At first glance, this seems to be similar to the previous loop-with-exit examples. It’s used in simulations in which *// do something* doesn’t need to be executed at the first pass through the loop but *// do something else* does. It’s a one-in, one-out control construct: The only way into the loop is through the *goto* at the top; the only way out of the loop is through the *while* test. This approach has two problems: It uses a *goto*, and it’s unusual enough to be confusing.

172
173
174

In C++, you can accomplish the same effect without using a *goto*, as demonstrated in the following example. If the language you're using doesn't support a *break* or *leave* command, you can emulate one with a *goto*.

175
176
177 *The blocks before and after*
178 *the break have been*
179 *switched.*
180
181
182
183
184
185
186

C++ Example of Code Rewritten Without a *goto*—Better Practice

```
FOREVER {  
    // do something else  
    ...  
  
    if ( !( expression ) ) {  
        break;  
    }  
  
    // do something  
    ...  
}
```

187

188 **FURTHER READING** For
189 more good guidelines on
190 using for loops, see *Writing
Solid Code* (Maguire 1993).

191
192
193
194
195
196

When to Use a *for* Loop

A *for* loop is a good choice when you need a loop that executes a specified number of times. You can use *for* in C++, C, Java, Visual Basic, and most other languages.

Use *for* loops for simple activities that don't require internal loop controls. Use them when the loop control involves simple increments or simple decrements. The point of a *for* loop is that you set it up at the top of the loop and then forget about it. You don't have to do anything inside the loop to control it. If you have a condition under which execution has to jump out of a loop, use a *while* loop instead.

197
198
199

Likewise, don't explicitly change the index value of a *for* loop to force it to terminate. Use a *while* loop instead. The *for* loop is for simple uses. Most complicated looping tasks are better handled by a *while* loop.

200

When to Use a *foreach* Loop

The *foreach* loop or its equivalent (*foreach* in C#, *For-Each* in Visual Basic, *For-In* in Python), is useful for performing an operation on each member of an array or other container. It has the advantage of eliminating loop-housekeeping arithmetic, and therefore eliminating any chance of errors in the loop-housekeeping arithmetic. Here's an example of this kind of loop:

206
207

C# Example of a *foreach* Loop

```
int [] fibonacciSequence = new int [] { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

```
208  
209  
210  
211  
212     int oddFibonacciNumbers = 0;  
213     int evenFibonacciNumbers = 0;  
214  
215     // count the number of odd and even numbers in a Fibonacci sequence  
216     foreach ( int fibonacciNumber in fibonacciSequence ) {  
217         if ( fibonacciNumber % 2 ) == 0 ) {  
218             evenFibonacciNumbers++;  
219         }  
220         else {  
221             oddFibonacciNumbers++;  
222         }  
223     }  
224  
225     Console.WriteLine( "Found {0} odd numbers and {1} even numbers.",  
226                     oddFibonacciNumbers, evenFibonacciNumbers );  
227
```

223 16.2 Controlling the Loop

```
224  
225  
226  
227  
228  
229
```

What can go wrong with a loop? Any answer would have to include at the very least incorrect or omitted loop initialization, omitted initialization of accumulators or other variables related to the loop, improper nesting, incorrect termination of the loop, forgetting to increment a loop variable or incrementing the variable incorrectly, and indexing an array element from a loop index incorrectly.

230 KEY POINT

```
231  
232  
233  
234  
235  
236
```

237 **CROSS-REFERENCE** If you use the *FOREVER-break*
238 technique described earlier,
239 the exit condition is inside
240 the black box. Even if you
241 use only one exit condition,
242 you lose the benefit of
243 treating the loop as a black
244 box.
245

```
246  
247  
248  
249  
250
```

You can forestall these problems by observing two practices. First, minimize the number of factors that affect the loop. Simplify! Simplify! Simplify! Second, treat the inside of the loop as if it were a routine—keep as much of the control as possible outside the loop. Explicitly state the conditions under which the body of the loop is to be executed. Don’t make the reader look inside the loop to understand the loop control. Think of a loop as a black box: The surrounding program knows the control conditions but not the contents.

C++ Example of Treating a Loop as a Black Box

```
while ( !inputFile.EndOfFile() && moreDataAvailable ) {  
    [REDACTED]  
}
```

What are the conditions under which this loop terminates? Clearly, all you know is that either *inputFile.EndOfFile()* becomes true or *MoreDataAvailable* becomes false.

251 Entering the Loop

252 Here are several guidelines for entering a loop:

253 *Enter the loop from one location only*

254 A variety of loop-control structures allows you to test at the beginning, middle,
255 or end of a loop. These structures are rich enough to allow you to enter the loop
256 from the top every time. You don't need to enter at multiple locations.

257 *Put initialization code directly before the loop*

258 The Principle of Proximity advocates putting related statements together. If
259 related statements are strewn across a routine, it's easy to overlook them during
260 modification and to make the modifications incorrectly. If related statements are
261 kept together, it's easier to avoid errors during modification.

262 Keep loop-initialization statements with the loop they're related to. If you don't,
263 you're more likely to cause errors when you generalize the loop into a bigger
264 loop and forget to modify the initialization code. The same kind of error can
265 occur when you move or copy the loop code into a different routine without
266 moving or copying its initialization code. Putting initializations away from the
267 loop—in the data-declaration section or in a housekeeping section at the top of
268 the routine that contains the loop—invites initialization troubles.

269 *In C++, use the FOREVER macro for infinite loops and event loops*

270 You might have a loop that runs without terminating—for example, a loop in
271 firmware such as a pacemaker or a microwave oven. Or you might have a loop
272 that terminates only in response to an event—an “event loop.” You could code
273 an infinite loop in several ways, but the following macro is the standard way to
274 code one in C++:

275 C++ Example of an Infinite Loop

```
276 #define FOREVER    for (;;) {  
277     ...  
278     FOREVER {  
279         ...  
280     }  
281 }
```

Here's the infinite loop.

281 This technique is the standard way to implement infinite loops and event loops.
282 Faking an infinite loop with a statement like *for i := 1 to 9999* is making a poor
283 substitution because using loop limits muddies the intent of the loop—maybe
284 9999 is a legitimate value. Such a fake infinite loop can also break down under
285 maintenance.

286 ***In C++ and Java, use for(;;) or while(true) for infinite loops***
287 As an alternative to the *FOREVER* macro, the *for(;;)* and *while(true)* idioms
288 are also considered standard ways of writing infinite loops in C++ and Java.

289 ***In C++, prefer for loops when they're appropriate***
290 The C++ *for* loop is one of the language's powerful constructs. Not only is it
291 flexible, but it packages loop-control code in one place, which makes for
292 readable loops. One mistake programmers commonly make when modifying
293 software is changing the loop-initialization code at the top of a loop but
294 forgetting to change related code at the bottom. In a C++ *for* loop, all the
295 relevant code is together at the top of the loop, which makes correct
296 modifications easier. If you can use the *for* loop appropriately in C++ instead of
297 another kind of loop, do it.

298 ***Don't use a for loop when a while loop is more appropriate***
299 A common abuse of C++'s flexible *for* loop is haphazardly cramming the
300 contents of a *while* loop into a *for* loop header. The following example shows a
301 *while* loop crammed into a *for* loop header.

CODING HORROR

302 ***C++ Example of a while Loop Abusively Crammed into a for Loop***
303 **Header**
304

```
// read all the records from a file
for ( inputFile.MoveToStart(), recordCount = 0; !inputFile.EndOfFile();
      recordCount++ ) {
    inputFile.GetRecord();
}
```

305 The advantage of C++'s *for* loop over *for* loops in other languages is that it's
306 more flexible about the kinds of initialization and termination information it can
307 use. The weakness inherent in such flexibility is that you can put statements into
308 the loop header that have nothing to do with controlling the loop.

313 Reserve the *for* loop header for loop-control statements—statements that
314 initialize the loop, terminate it, or move it toward termination. In the example
315 above, the *inputFile.GetRecord()* statement in the body of the loop moves the
316 loop toward termination, but the *recordCount* statements don't; they're
317 housekeeping statements that don't control the loop's progress. Putting the
318 *recordCount* statements in the loop header and leaving the
319 *inputFile.GetRecord()* statement out is misleading; it creates the false
320 impression that *recordCount* controls the loop.

321 If you want to use the *for* loop rather than the *while* loop in this case, put the
322 loop-control statements in the loop header and leave everything else out. Here's
323 the right way to use the loop header:

324

```
recordCount = 0;
for ( inputFile.MoveToStart(); !inputFile.EndOfFile(); inputFile.GetRecord() ) {
    recordCount++;
}
```

The contents of the loop header in this example are all related to control of the loop. The *inputFile.MoveToStart()* statement initializes the loop; the *!inputFile.EndOfFile()* statement tests whether the loop has finished; and the *inputFile.GetRecord()* statement moves the loop toward termination. The statements that affect *recordCount* don't directly move the loop toward termination and are appropriately not included in the loop header. The *while* loop is probably still more appropriate for this job, but at least this code uses the loop header logically. For the record, here's how the code looks when it uses a *while* loop:

338

C++ Example of Appropriate Use of a *while* Loop

```
// read all the records from a file
inputFile.MoveToStart();
recordCount = 0;
while ( !inputFile.EndOfFile() ) {
    inputFile.GetRecord( &inputRec[ recordCount ], MAX_CHARS );
    recordCount++;
}
```

346

Processing the Middle of the Loop

347

Here are several guidelines for handling the middle of a loop:

348

Use { and } to enclose the statements in a loop
Use code brackets every time. They don't cost anything in space or speed at run time, they help readability, and they help prevent errors as the code is modified. They're a good defensive programming practice.

352

Avoid empty loops
In C++ and Java, it's possible to create an empty loop, one in which the work the loop is doing is coded on the same line as the test that checks whether the work is finished. Here's an example:

356

C++ Example of an Empty Loop

```
while ( ( inputChar = cin.get() ) != '\n' ) {
    ;
}
```

In this example, the loop is empty because the *while* expression includes two things: the work of the loop—*inputChar = cin.get()*—and a test for whether the

362 loop should terminate—*inputChar != <;\$QS>\n;>\$QS>*. The loop would be
363 clearer if it were recoded so that the work it does is evident to the reader. Here's
364 how the revised loop would look:

365 C++ Example of an Empty Loop Converted to an Occupied Loop

```
366      do {  
367         inputChar = cin.get();  
368      } while ( inputChar != '\n' );
```

369 The new code takes up three full lines rather than one line and a semicolon,
370 which is appropriate since it does the work of three lines rather than that of one
371 line and a semicolon.

372 *Keep loop-housekeeping chores at either the beginning or the end of the* 373 *loop*

374 “Loop housekeeping” chores are expressions like $i = i + 1$, expressions whose
375 main purpose isn’t to do the work of the loop but to control the loop. Here’s an
376 example in which the housekeeping is done at the end of the loop:

377 C++ Example of Housekeeping Statements at the End of a Loop

```
378      stringIndex = 1;  
379      totalLength = 0;  
380      while ( !inputFile.EndOfFile() ) {  
381         // do the work of the loop  
382         inputFile >> inputString;  
383         strList[ stringIndex ] = inputString;  
384         ...  
385           
386         // prepare for next pass through the loop--housekeeping  
387         stringIndex++;  
388         totalLength = totalLength + inputString.length();  
389      }
```

390 *Here are the housekeeping*
391 *statements.*

392
393 **CROSS-REFERENCE** For
394 more on optimization, see
395 Chapters 25 and 26.

396
397
398
399

400

401

402

403

404

405

406

407

408

409

410

411

412

413

414

CODING HORROR

415

416

417

418

419

420 *Here's the monkeying.*

421

422

423

424

425

426

427

428

429

430

431

432

433

434

435

436

437

438

Exiting the Loop

Here are several guidelines for handling the end of a loop:

Assure yourself that the loop ends

This is fundamental. Mentally simulate the execution of the loop until you are confident that, in all circumstances, it ends. Think through the nominal cases, the endpoints, and each of the exceptional cases.

Make loop-termination conditions obvious

If you use a *for* loop and don't fool around with the loop index and don't use a *goto* or *break* to get out of the loop, the termination condition will be obvious. Likewise, if you use a *while* or *repeat-until* loop and put all the control in the *while* or *repeat-until* clause, the termination condition will be obvious. The key is putting the control in one place.

Don't monkey with the loop index of a for loop to make the loop terminate

Some programmers jimmy the value of a *for* loop index to make the loop terminate early. Here's an example:

Java Example of Monkeying with a Loop Index

```
for ( int i = 0; i < 100; i++ ) {  
    // some code  
    ...  
    if ( ... ) {  
        i = 100;  
    }  
  
    // more code  
    ...  
}
```

The intent in this example is to terminate the loop under some condition by setting *i* to *100*, a value that's larger than the end of the *for* loop's range of *0* through *99*. Virtually all good programmers avoid this practice; it's the sign of an amateur. When you set up a *for* loop, the loop counter is off limits. Use a *while* loop to provide more control over the loop's exit conditions.

Avoid code that depends on the loop index's final value

It's bad form to use the value of the loop index after the loop. The terminal value of the loop index varies from language to language and implementation to implementation. The value is different when the loop terminates normally and when it terminates abnormally. Even if you happen to know what the final value is without stopping to think about it, the next person to read the code will probably have to think about it. It's better form and more self-documenting if you assign the final value to a variable at the appropriate point inside the loop.

439

Here's an example of code that misuses the index's final value:

```
440
441
442
443
444
445
446
447
448 Here's the misuse of the loop
449     index's terminal value.
450
451
452
453
454
455
456
457
458
459
```

C++ Example of Code That Misuses a Loop Index's Terminal Value

```
for ( recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
    if ( entry[ recordCount ] == testValue ) {
        break;
    }
}
// lots of code
...
if ( recordCount < MAX_RECORDS ) {
    return( true );
}
else {
    return( false );
}
```

In this fragment, the second test for *recordCount < MaxRecords* makes it appear that the loop is supposed to loop though all the values in *entry[]* and return *true* if it finds the one equal to *TestValue*, *false* otherwise. It's hard to remember whether the index gets incremented past the end of the loop, so it's easy to make an off-by-one error. You're better off writing code that doesn't depend on the index's final value. Here's how to rewrite the code:

C++ Example of Code That Doesn't Misuse a Loop Index's Terminal Value

```
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
```

```
found = false;
for ( recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
    if ( entry[ recordCount ] == testValue ) {
        found = true;
        break;
    }
}
// lots of code
...
return( found );
```

This second code fragment uses an extra variable, and keeps references to *recordCount* more localized. As is often the case when an extra boolean variable is used, the resulting code is clearer.

475 **Consider using safety counters**
476 If you have a program in which an error would be catastrophic, you can use
477 safety counters to ensure that all loops end. Here's a C++ loop that could
478 profitably use a safety counter:

479

```
480 do {  
481     node = node->Next;  
482     ...  
483 } while ( node->Next != NULL );
```

484 Here's the same code with the safety counters added:

485

```
486 safetyCounter = 0;  
487 do {  
488     node = node->Next;  
489     ...  
490     Here's the safety-counter  
491             code.  
492     safetyCounter++;  
493     if ( safetyCounter >= SAFETY_LIMIT ) {  
494         Assert( false, "Internal Error: Safety-Counter Violation." );  
495     }  
496     ...  
497 } while ( node->Next != NULL );
```

498 Safety counters are not a cure all. Introduced into the code one at a time, safety
499 counters might lead to additional errors. If they aren't used in every loop, you
500 could forget to maintain safety-counter code when you modify loops in parts of
501 the program that do use them. If safety counters are instituted as a project-wide
502 standard, however, you learn to expect them, and safety-counter code is no more
503 prone to produce errors later than any other code is.

502

Exiting Loops Early

503 Many languages provide a means of causing a loop to terminate in some way
504 other than completing the *for* or *while* condition. In this discussion, *break* is a
505 generic term for *break* in C++, C, and Java, *Exit-Do* and *Exit-For* in Visual
506 Basic, and similar constructs, including those simulated with *gotos* in languages
507 that don't support *break* directly. The *break* statement (or equivalent) causes a
508 loop to terminate through the normal exit channel; the program resumes
509 execution at the first statement following the loop.

510

511 The *continue* statement is similar to *break* in that it's an auxiliary loop-control
512 statement. Rather than causing a loop exit, however, *continue* causes the
513 program to skip the loop body and continue executing at the beginning of the
514 next iteration of the loop. A *continue* statement is shorthand for an *if-then* clause
that would prevent the rest of the loop from being executed.

515

516 **Consider using break statements rather than boolean flags in a while loop**
517 In some cases, adding boolean flags to a *while* loop to emulate exits from the
518 body of the loop makes the loop hard to read. Sometimes you can remove
several levels of indentation inside a loop and simplify loop control just by using

519 a *break* instead of a series of *if* tests. Putting multiple *break* conditions into
520 separate statements and placing them near the code that produces the *break* can
521 reduce nesting and make the loop more readable.

522 ***Be wary of a loop with a lot of breaks scattered through it***

523 A loop's containing a lot of *breaks* can indicate unclear thinking about the
524 structure of the loop or its role in the surrounding code. A proliferation of *breaks*
525 raises the possibility that the loop could be more clearly expressed as a series of
526 loops rather than as one loop with many exits.

527 According to an article in Software Engineering Notes, the software error that
528 brought down the New York City phone systems for 9 hours on January 15,
529 1990 was due to an extra *break* statement (SEN 1990):

530 **C++ Example of Erroneous Use of a *break* Statement Within a *do*-
531 *switch-if* Block.**

```
532 do {  
533     ...  
534     switch  
535         ...  
536         if (0) {  
537             ...  
538             This break was intended for  
539                 the if, but broke out of the  
540                     switch instead.  
541             break;  
542             ...  
543         }  
544         ...  
545     } while ( ... );
```

Multiple *breaks* don't necessarily indicate an error, but their existence in a loop
is a warning sign, a canary in a coal mine that's not singing as loud as it should
be.

546 ***Use continue for tests at the top of a loop***

547 A good use of *continue* is for moving execution past the body of the loop after
548 testing a condition at the top. For example, if the loop reads records, discards
549 records of one kind, and processes records of another kind, you could put a test
550 like this one at the top of the loop:

551 **Pseudocode Example of a Relatively Safe Use of *continue***

```
552 while ( not eof( file ) ) do  
553     read( record, file )  
554     if ( record.Type <> targetType ) then  
555         continue  
556  
557         -- process record of targetType
```

```
558 ...
559 end while
```

Using *continue* in this way lets you avoid an *if* test that would effectively indent the entire body of the loop. If, on the other hand, the *continue* occurs toward the middle or end of the loop, use an *if* instead.

563 **Use labeled break if your language supports it**

564 Java supports use of labeled *breaks* to prevent the kind of problem experienced
565 with the New York City telephone outage. A labeled *break* can be used to exit
566 for a for loop, an if statement, or any block of code enclosed in braces (Arnold,
567 Gosling, and Holmes 2000).

568 Here's a possible solution to the New York City telephone code problem, with
569 the programming language changed from C++ to Java to show the labeled
570 break:

571 **Java Example of a Better Use of a labeled break Statement Within a do- 572 switch-if Block.**

```
573 do {
574 ...
575     switch
576         ...
577             CALL_CENTER_DOWN:
578                 if () {
579                     ...
580                     The target of the labeled
581                     break is unambiguous.
582                         break CALL_CENTER_DOWN;
583                         ...
584                 }
585             ...
586         } while ( ... );
```

585 **Use break and continue only with caution**

586 Use of *break* eliminates the possibility of treating a loop as a black box.
587 Limiting yourself to only one statement to control a loop's exit condition is a
588 powerful way to simplify your loops. Using a *break* forces the person reading
589 your code to look inside the loop for an understanding of the loop control. That
590 makes the loop more difficult to understand.

591 Use *break* only after you have considered the alternatives. To paraphrase the
592 nineteenth-century Danish philosopher Søren Kierkegaard, you don't know with
593 certainty whether *continue* and *break* are virtuous or evil constructs. Some
594 computer scientists argue that they are a legitimate technique in structured
595 programming; some argue that they are not. Because you don't know in general
596 whether *continue* and *break* are right or wrong, use them, but only with a fear

597
598

that you might be wrong. It really is a simple proposition: If you can't defend a *break* or a *continue*, don't use it.

599

Checking Endpoints

600
601
602
603
604
605

A single loop usually has three cases of interest: the first case, an arbitrarily selected middle case, and the last case. When you create a loop, mentally run through the first, middle, and last cases to make sure that the loop doesn't have any off-by-one errors. If you have any special cases that are different from the first or last case, check those too. If the loop contains complex computations, get out your calculator and manually check the calculations.

606 **KEY POINT**

607
608
609

Willingness to perform this kind of check is a key difference between efficient and inefficient programmers. Efficient programmers do the work of mental simulations and hand calculations because they know that such measures help them find errors.

610
611
612
613
614
615
616
617

Inefficient programmers tend to experiment randomly until they find a combination that seems to work. If a loop isn't working the way it's supposed to, the inefficient programmer changes the `<` sign to a `<=` sign. If that fails, the inefficient programmer changes the loop index by adding or subtracting 1. Eventually the programmer using this approach might stumble onto the right combination or simply replace the original error with a more subtle one. Even if this random process results in a correct program, it doesn't result in the programmer's knowing why the program is correct.

618
619
620
621
622

You can expect several benefits from mental simulations and hand calculations. The mental discipline results in fewer errors during initial coding, in more rapid detection of errors during debugging, and in a better overall understanding of the program. The mental exercise means that you understand how your code works rather than guessing about it.

623

Using Loop Variables

624

Here are some guidelines for using loop variables:

625 **CROSS-REFERENCE** For
626 details on naming loop
627 variables, see "Naming Loop
Indexes" in Section 11.2.
628
629

Use ordinal or enumerated types for limits on both arrays and loops

Generally, loop counters should be integer values. Floating-point values don't increment well. For example, you could add 1.0 to 26,742,897.0 and get 26,742,897.0 instead of 26,742,898.0. If this incremented value were a loop counter, you'd have an infinite loop.

630 **KEY POINT**

631
632

Use meaningful variable names to make nested loops readable

Arrays are often indexed with the same variables that are used for loop indexes. If you have a one-dimensional array, you might be able to get away with using *i*,

633
634
635
636
j, or *k* to index it. But if you have an array with two or more dimensions, you
should use meaningful index names to clarify what you're doing. Meaningful
array-index names clarify both the purpose of the loop and the part of the array
you intend to access.

637
638
Here's code that doesn't put this principle to work, using the meaningless names
i, *j*, and *k* instead:

CODING HORROR

639
640
641
642
643
644
645
646
647
648
649
650

```
for ( int i = 0; i < numPayCodes; i++ ) {  
    for ( int j = 0; j < 12; j++ ) {  
        for ( int k = 0; k < numDivisions; k++ ) {  
            sum = sum + transaction[ j ][ i ][ k ];  
        }  
    }  
}
```

What do you think the array indexes in *transaction* mean? Do *i*, *j*, and *k* tell
you anything about the contents of *transaction*? If you had the declaration of
transaction, could you easily determine whether the indexes were in the right
order? Here's the same loop with more readable loop variable names:

Java Example of Good Loop Variable Names

651
652
653
654
655
656
657
658
659
660
661
662
663
664

```
for ( int payCodeIdx = 0; payCodeIdx < numPayCodes; payCodeIdx++ ) {  
    for (int month = 0; month < 12; month++ ) {  
        for ( int divisionIdx = 0; divisionIdx < numDivisions; divisionIdx++ ) {  
            sum = sum + transaction[ month ][ payCodeIdx ][ divisionIdx ];  
        }  
    }  
}
```

What do you think the array indexes in *transaction* mean this time? In this case,
the answer is easier to come by because the variable names *payCodeIdx*, *month*,
and *divisionIdx* tell you a lot more than *i*, *j*, and *k* did. The computer can read the
two versions of the loop equally easily. People can read the second version more
easily than the first, however, and the second version is better since your
primary audience is made up of humans, not computers.

665
666
667
Use meaningful names to avoid loop-index cross talk

Habitual use of *i*, *j*, and *k* can give rise to index cross talk—using the same index
name for two different purposes. Here's an example:

C++ Example of Index Cross Talk

668
669 *i* is used first here...
670 for (i = 0; i < numPayCodes; i++) {
671 // lots of code
 ...

```
672     for ( j = 0; j < 12; j++ ) {  
673         // lots of code  
674         ...  
675         ...and again here.  
676         for ( i = 0; i < numDivisions; i++ ) {  
677             sum = sum + transaction[ j ][ i ][ k ];  
678         }  
679     }  
680 }
```

The use of *i* is so habitual that it's used twice in the same nesting structure. The second *for* loop controlled by *i* conflicts with the first, and that's index cross talk. Using more meaningful names than *i*, *j*, and *k* would have prevented the problem. In general, if the body of a loop has more than a couple of lines, if it might grow, or if it's in a group of nested loops, avoid *i*, *j*, and *k*.

685 ***Limit the scope of loop-index variables to the loop itself***
686 Loop-index cross-talk and other uses of loop indexes outside their loops is such
687 a significant problem that the designers of Ada decided to make *for* loop indexes
688 invalid outside their loops; trying to use one outside its *for* loop generates an
689 error at compile time.

690 C++ and Java implement the same idea to some extent—they allow loop indexes
691 to be declared within a loop, but they don't require it. In the example on page
692 000, the *recordCount* variable could be declared inside the *for* statement, which
693 would limit its scope to the *for* loop, like this:

694 **C++ Example of Declaring a Loop-Index variable Within a *for* loop**

```
695 for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {  
696     // looping code that uses recordCount  
697 }
```

698 In principle, this technique should allow creation of code that redeclares
699 *recordCount* in multiple loops without any risk of misusing the two different
700 *recordCounts*. That usage would give rise to code that looks like this:

701 **C++ Example of Declaring Loop-Indexes Within *for* loops and reusing 702 them safely—Maybe!**

```
703 for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {  
704     // looping code that uses recordCount  
705 }  
706  
707     // intervening code  
708  
709     for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {  
710         // additional looping code that uses a different recordCount  
711     }
```

This technique is helpful for documenting the purpose of the *recordCount* variable, however don't rely on your compiler to enforce *recordCount*'s scope. Section 6.3.3.1 of *The C++ Programming Language* (Stroustrup 1997) says that *recordCount* should have a scope limited to its loop. When I checked this functionality with three different C++ compilers, however, I got three different results:

- The first compiler flagged *recordCount* in the second *for* loop for multiple variable declarations and generated an error.
- The second compiler accepted *recordCount* in the second *for* loop but allowed it to be used outside the first *for* loop.
- The third compiler allowed both usages of *recordCount* and did not allow either one to be used outside the *for* loop in which it was declared.

As is often the case with more esoteric language features, compiler implementations can vary.

How Long Should a Loop Be?

Loop length can be measured in lines of code or depth of nesting. Here are some guidelines:

Make your loops short enough to view all at once

If you usually look at loops on 66-line paper, that puts a 66-line restriction on you. If your monitor displays 50 lines, that puts a 50-line restriction on you. Experts have suggested a loop-length limit of one printed page, or 66 lines. When you begin to appreciate the principle of writing simple code, however, you'll rarely write loops longer than 15 or 20 lines.

Limit nesting to three levels

Studies have shown that the ability of programmers to comprehend a loop deteriorates significantly beyond three levels of nesting (Yourdon 1986a). If you're going beyond that number of levels, make the loop shorter (conceptually) by breaking part of it into a routine or simplifying the control structure.

Move loop innards of long loops into routines

If the loop is well designed, the code on the inside of a loop can often be moved into one or more routines that are called from within the loop.

Make long loops especially clear

Length adds complexity. If you write a short loop, you can use riskier control structures such as *break* and *continue*, multiple exits, complicated termination conditions, and so on. If you write a longer loop and feel any concern for your

747
748

reader, you'll give the loop a single exit and make the exit condition unmistakably clear.

749

16.3 Creating Loops Easily—from the Inside Out

750

751
752

If you sometimes have trouble coding a complex loop—which most programmers do—you can use a simple technique to get it right the first time.

753
754
755
756
757
758

Here's the general process. Start with one case. Code that case with literals. Then indent it, put a loop around it, and replace the literals with loop indexes or computed expressions. Put another loop around that, if necessary, and replace more literals. Continue the process as long as you have to. When you finish, add all the necessary initializations. Since you start at the simple case and work outward to generalize it, you might think of this as coding from the inside out.

759 **CROSS-REFERENCE** This
760 process is similar to the
761 process described in Chapter
762 9, "The Pseudocode
763 Programming Process."

Suppose you're writing a program for an insurance company. It has life-insurance rates that vary according to a person's age and sex. Your job is to write a routine that computes the total life-insurance premium for a group. You need a loop that takes the rate for each person in a list and adds it to a total. Here's how you'd do it.

764
765
766

First, in comments, write the steps the body of the loop needs to perform. It's easier to write down what needs to be done when you're not thinking about details of syntax, loop indexes, array indexes, and so on.

767
768
769
770
771
772
773

Step 1: Creating a Loop from the Inside Out (Pseudocode Example)

```
-- get rate from table  
-- add rate to total
```

Second, convert the comments in the body of the loop to code, as much as you can without actually writing the whole loop. In this case, get the rate for one person and add it to the overall total. Use concrete, specific data rather than abstractions.

774
775 table doesn't have any
776 indexes yet.
777
778
779
780

Step 2: Creating a Loop from the Inside Out (Pseudocode Example)

```
rate = table[ ]  
totalRate = totalRate + rate
```

The example assumes that *table* is an array that holds the rate data. You don't have to worry about the array indexes at first. *rate* is the variable that holds the rate data selected from the rate table. Likewise, *totalRate* a variable that holds the total of the rates.

781 Next, put in indexes for the *table* array.

782 Step 3: Creating a Loop from the Inside Out (Pseudocode Example)

```
783 rate = table[ census.Age ][ census.Gender ]  
784 totalRate = totalRate + rate
```

785 The array is accessed by age and sex, so *census.Age* and *census.Gender* are used
786 to index the array. The example assumes that *census* is a structure that holds
787 information about people in the group to be rated.

788 The next step is to build a loop around the existing statements. Since the loop is
789 supposed to compute the rates for each person in a group, the loop should be
790 indexed by person.

791 Step 4: Creating a Loop from the Inside Out (Pseudocode Example)

```
792 For person = firstPerson to lastPerson  
793     rate = table[ census.Age, census.Gender ]  
794     totalRate = totalRate + rate  
795 End For
```

796 All you have to do here is put the *for* loop around the existing code and then
797 indent the existing code and put it inside a *begin-end* pair. Finally, check to
798 make sure that the variables that depend on the *person* loop index have been
799 generalized. In this case, the *census* variable varies with *person*, so it should be
800 generalized appropriately.

801 Step 5: Creating a Loop from the Inside Out (Pseudocode Example)

```
802 For person = firstPerson to lastPerson  
803     rate = table[ census[ person ].Age, census[ person ].Gender ]  
804     totalRate = totalRate + rate  
805 End For
```

806 Finally, write any initializations that are needed. In this case, the *totalRate*
807 variable needs to be initialized. The final code appears next.

808 Final Step: Creating a Loop from the Inside Out (Pseudocode Example)

```
809 totalRate = 0  
810 For person = firstPerson to lastPerson  
811     rate = table[ census[ person ].Age, census[ person ].Gender ]  
812     totalRate = totalRate + rate  
813 End For
```

814 If you had to put another loop around the *person* loop, you would proceed in the
815 same way. You don't need to follow the steps rigidly. The idea is to start with
816 something concrete, worry about only one thing at a time, and build up the loop
817 from simple components. Take small, understandable steps as you make the
818 loop more general and complex. That way, you minimize the amount of code

819
820

you have to concentrate on at any one time and therefore minimize the chance of error.

821
822

16.4 Correspondence Between Loops and Arrays

823 **CROSS-REFERENCE** For
824 further discussion of the
825 correspondence between
826 loops and arrays, see Section
10.7, “Relationship Between
Data Types and Control
827 Structures.”

828
829
830
831
832
833
834
835
836
837

Loops and arrays are often related. In many instances, a loop is created to perform an array manipulation, and loop counters correspond one-to-one with array indexes. For example, the Java *for* loop indexes below correspond to the array indexes:

Java Example of an Array Multiplication

```
for ( int row = 0; row < maxRows; row++ ) {  
    for ( int column = 0; column < maxCols; column++ ) {  
        product[ row ][ column ] = a[ row ][ column ] * b[ row ][ column ];  
    }  
}
```

In Java, a loop is necessary for this array operation. But it's worth noting that looping structures and arrays aren't inherently connected. Some languages, especially APL and Fortran 90 and later, provide powerful array operations that eliminate the need for loops like the one above. Here's an APL code fragment that performs the same operation:

APL Example of an Array Multiplication

```
Product <- a x b
```

The APL is simpler and less error prone. It uses only 3 operands, whereas the Java fragment uses 17. It doesn't have loop variables, array indexes, or control structures to code incorrectly.

One point of this example is that you do some programming to solve a problem and some to solve it in a particular language. The language you use to solve a problem substantially affects your solution.

838
839
840
841
842
843
844
845

CC2E.COM/1616
846

CHECKLIST: Loops

847
848
849
850
851

Loop Selection and Creation

- Is a *while* loop used instead of a *for* loop, if appropriate?
- Was the loop created from the inside out?

Entering the Loop

- Is the loop entered from the top?

- 852 Is initialization code directly before the loop?
- 853 If the loop is an infinite loop or an event loop, is it constructed cleanly
854 rather than using a kludge such as *for i = 1 to 9999?*
- 855 If the loop is a C++, C, or Java *for* loop, is the loop header reserved for
856 loop-control code?

857 **Inside the Loop**

- 858 Does the loop use *{* and *}* or their equivalent to prevent problems arising
859 from improper modifications?
- 860 Does the loop body have something in it? Is it nonempty?
- 861 Are housekeeping chores grouped, at either the beginning or the end of the
862 loop?
- 863 Does the loop perform one and only one function—as a well-defined routine
864 does?
- 865 Is the loop short enough to view all at once?
- 866 Is the loop nested to three levels or less?
- 867 Have long loop contents been moved into their own routine?
- 868 If the loop is long, is it especially clear?

869 **Loop Indexes**

- 870 If the loop is a *for* loop, does the code inside it avoid monkeying with the
871 loop index?
- 872 Is a variable used to save important loop-index values rather than using the
873 loop index outside the loop?
- 874 Is the loop index an ordinal type or an enumerated type—not floating point?
- 875 Does the loop index have a meaningful name?
- 876 Does the loop avoid index cross talk?

877 **Exiting the Loop**

- 878 Does the loop end under all possible conditions?
- 879 Does the loop use safety counters—if you’ve instituted a safety-counter
880 standard?
- 881 Is the loop’s termination condition obvious?
- 882 If *break* or *continue* are used, are they correct?

Key Points

- Loops are complicated. Keeping them simple helps readers of your code.
- Techniques for keeping loops simple include avoiding exotic kinds of loops, minimizing nesting, making entries and exits clear, and keeping housekeeping code in one place.
- Loop indexes are subjected to a great deal of abuse. Name them clearly and use them for only one purpose.
- Think the loop through carefully to verify that it operates normally under each case and terminates under all possible conditions.

17

2 Unusual Control Structures

3 CC2E.COM/1778

4 Contents

- 5 17.1 Multiple Returns from a Routine
- 6 17.2 Recursion
- 7 17.3 *goto*
- 17.4 Perspective on Unusual Control Structures

8 Related Topics

9 General control issues: Chapter 19

10 Straight-line code: Chapter 14

11 Code with conditionals: Chapter 15

12 Code with loops: Chapter 16

13 Exception handling: Section 8.4

14 SEVERAL CONTROL CONSTRUCTS exist in a hazy twilight zone somewhere
15 between being leading-edge and being discredited and disproved—often in both
16 places at the same time! These constructs aren't available in all languages but
17 can be useful when used with care in those languages that do offer them.

18 17.1 Multiple Returns from a Routine

19 Most languages support some means of exiting from a routine partway through
20 the routine. The *return* and *exit* statements are control constructs that enable a
21 program to exit from a routine at will. They cause the routine to terminate
22 through the normal exit channel, returning control to the calling routine. The
23 word *return* is used here as a generic term for *return* in C++ and Java, *Exit Sub*
24 and *Exit Function* in Visual Basic, and similar constructs. Here are guidelines for
25 using the *return* statement:

26 **KEY POINT**

27 ***Use a return when it enhances readability***
28 In certain routines, once you know the answer, you want to return it to the
calling routine immediately. If the routine is defined in such a way that it doesn't

29 require any further cleanup once it detects an error, not returning immediately
30 means that you have to write more code.

31 The following is a good example of a case in which returning from multiple
32 places in a routine makes sense:

C++ Example of a Good Multiple Return from a Routine

```
34 COMPARISON Compare ( int value1, int value2 ) {  
35     if ( value1 < value2 ) {  
36         return Comparison_LessThan;  
37     }  
38     else if ( value1 > value2 ) {  
39         return Comparison_GreaterThan;  
40     }  
41     else {  
42         return Comparison_Equal;  
43     }  
44 }
```

45 Other examples are less clear-cut, as the next section illustrates.

Use guard clauses (early returns or exits) to simplify complex error processing

46 Code that has to check for numerous error conditions before performing its
47 nominal actions can result in deeply indented code and can obscure the nominal
48 case, as shown here:

Visual Basic Code That Obscures the Nominal Case

```
51 If file.validName() Then  
52     If file.Open() Then  
53         If encryptionKey.valid() Then  
54             If file.Decrypt( encryptionKey ) Then  
55                 ' lots of code  
56                 ...  
57             End If  
58         End If  
59     End If  
60 End If
```

This is the code for the
nominal case.

61 Indenting the main body of the routine inside four *if* statements is aesthetically
62 ugly, especially if there's much code inside the innermost *if* statement. In such
63 cases, the flow of the code is sometimes clearer if the erroneous cases are
64 checked first, clearing the way for the nominal path through the code. Here's
65 how that might look:

67

Simple Visual Basic Code That Uses Early Exits to Clarify the Nominal Case

```
69      ' set up, bailing out if errors are found
70      If Not file.validName() Then Exit Sub
71      If Not file.Open() Then Exit Sub
72      If Not encryptionKey.valid() Then Exit Sub
73      If Not file.Decrypt( encryptionKey ) Then Exit Sub
74
75      ' lots of code
76      ...
77
```

The simple code above makes this technique look like a tidy solution, but production code often requires more extensive housekeeping or cleanup when an error condition is detected. Here is a more realistic example:

80

More Realistic Visual Basic Code That Uses Early Exits to Clarify the Nominal Case

```
82      ' set up, bailing out if errors are found
83      If Not file.validName() Then
84          errorStatus = FileError_InvalidFileName
85          Exit Sub
86      End If
87
88      If Not file.Open() Then
89          errorStatus = FileError_CantOpenFile
90          Exit Sub
91      End If
92
93      If Not encryptionKey.valid() Then
94          errorStatus = FileError_InvalidEncryptionKey
95          Exit Sub
96      End If
97
98      If Not file.Decrypt( encryptionKey ) Then
99          errorStatus = FileError_CantDecryptFile
100         Exit Sub
101     End If
102
103     ' lots of code
104     ...
105
```

This is the code for the nominal case.

With production-size code, the *Exit Sub* approach creates a noticeable amount of code before the nominal case is handled. The *Exit Sub* approach does avoid the deep nesting of the first example, however, and, if the code in the first example were expanded to show setting an *errorStatus* variable, the *Exit Sub* approach would do a better job of keeping related statements together. When all the dust

110 settles, the *Exit Sub* approach does appear more readable and maintainable, just
111 not by a very wide margin.

112 **Minimize the number of returns in each routine**

113 It's harder to understand a routine if, reading it at the bottom, you're unaware of
114 the possibility that it returned somewhere above. For that reason, use returns
115 judiciously—only when they improve readability.

116

17.2 Recursion

117 In recursion, a routine solves a small part of a problem itself, divides the problem
118 into smaller pieces, and then calls itself to solve each of the smaller pieces.
119 Recursion is usually called into play when a small part of the problem is easy to
120 solve and a large part is easy to decompose into smaller pieces.

121 **KEY POINT**
122 Recursion isn't useful very often, but when used judiciously it produces
123 exceptionally elegant solutions. Here's an example in which a sorting algorithm
makes excellent use of recursion:

Java Example of a Sorting Algorithm That Uses Recursion

```
125 void QuickSort( int firstIndex, int lastIndex, String [] names ) {  
126     if ( lastIndex > firstIndex ) {  
127         int midPoint = Partition( firstIndex, lastIndex, names );  
128         Here are the recursive calls.  
129         QuickSort( firstIndex, midPoint-1, names );  
130         QuickSort( midPoint+1, lastIndex, names )  
131     }  
132 }
```

In this case, the sorting algorithm chops an array in two and then calls itself to sort each half of the array. When it calls itself with a subarray that's too small to sort ($lastIndex \leq firstIndex$), it stops calling itself.

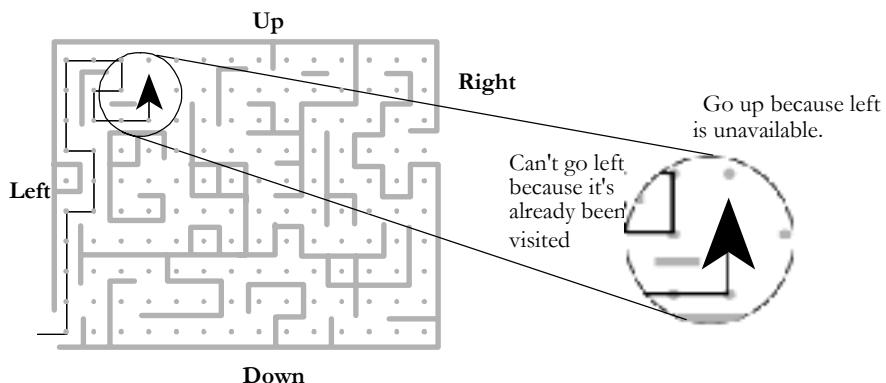
135 In general, recursion leads to small code and slow execution and chews up stack
136 space. For a small group of problems, recursion can produce simple, elegant
137 solutions. For a slightly larger group of problems, it can produce simple, elegant,
138 hard-to-understand solutions. For most problems, it produces massively
139 complicated solutions—in those cases, simple iteration is usually more
140 understandable. Use recursion selectively.

141

Example of Recursion

142 Suppose you have a data type that represents a maze. A maze is basically a grid,
143 and at each point on the grid you might be able to turn left, turn right, move up,
144 or move down. You'll often be able to move in more than one direction.

145 How do you write a program to find its way through the maze? If you use
 146 recursion, the answer is fairly straightforward. You start at the beginning and
 147 then try all possible paths until you find your way out of the maze. The first time
 148 you visit a point, you try to move left. If you can't move left, you try to go up or
 149 down, and if you can't go up or down, you try to go right. You don't have to
 150 worry about getting lost because you drop a few bread crumbs on each spot as
 151 you visit it, and you don't visit the same spot twice.



152

153

154

155

156

157

F17xx01**Figure 17-1**

Recursion can be a valuable tool in the battle against complexity—when used to attack suitable problems.

Here's how the recursive code looks:

C++ Example of Moving Through a Maze Recursively

```

159 bool FindPathThroughMaze( Maze maze, Point position ) {
160     // if the position has already been tried, don't try it again
161     if ( AlreadyTried( maze, position ) ) {
162         return false;
163     }
164
165     // if this position is the exit, declare success
166     if ( ThisIsTheExit( maze, position ) ) {
167         return true;
168     }
169
170     // remember that this position has been tried
171     RememberPosition( maze, position );
172
173     // check the paths to the left, up, down, and to the right; if
174     // any path is successful, stop looking
175     if ( MoveLeft( maze, position, &newPosition ) ) {

```

```
176     if ( FindPathThroughMaze( maze, newPosition ) ) {
177         return true;
178     }
179 }
180
181     if ( MoveUp( maze, position, &newPosition ) ) {
182         if ( FindPathThroughMaze( maze, newPosition ) ) {
183             return true;
184         }
185     }
186
187     if ( MoveDown( maze, position, &newPosition ) ) {
188         if ( FindPathThroughMaze( maze, newPosition ) ) {
189             return true;
190         }
191     }
192
193     if ( MoveRight( maze, position, &newPosition ) ) {
194         if ( FindPathThroughMaze( maze, newPosition ) ) {
195             return true;
196         }
197     }
198     return false;
199 }
```

The first line of code checks to see whether the position has already been tried. One key aim in writing a recursive routine is the prevention of infinite recursion. In this case, if you don't check for having tried a point, you might keep trying it infinitely.

The second statement checks to see whether the position is the exit from the maze. If *ThisIsTheExit()* returns *true*, the routine itself returns *true*.

The third statement remembers that the position has been visited. This prevents the infinite recursion that would result from a circular path.

The remaining lines in the routine try to find a path to the left, up, down, and to the right. The code stops the recursion if the routine ever returns *true*, that is, when the routine finds a path through the maze.

The logic used in this routine is fairly straightforward. Most people experience some initial discomfort using recursion because it's self-referential. In this case, however, an alternative solution would be much more complicated and recursion works well.

215 **Tips for Using Recursion**

216 Here are some tips for using recursion:

217 ***Make sure the recursion stops***

218 Check the routine to make sure that it includes a nonrecursive path. That usually
219 means that the routine has a test that stops further recursion when it's not
220 needed. In the maze example, the tests for *AlreadyTried()* and *ThisIsTheExit()*
221 ensure that the recursion stops.

222 ***Use safety counters to prevent infinite recursion***

223 If you're using recursion in a situation that doesn't allow a simple test such as
224 the one just described, use a safety counter to prevent infinite recursion. The
225 safety counter has to be a variable that's not re-created each time you call the
226 routine. Use a class member variable or pass the safety counter as a parameter.
227 Here's an example:

228 **Visual Basic Example of Using a Safety Counter to Prevent Infinite 229 Recursion**

230 *The recursive routine must be able to change the value of safetyCounter, so in Visual
231 Basic it's a ByRef parameter.*

```
232      Public Sub RecursiveProc( ByRef safetyCounter As Integer )
233         If ( safetyCounter > SAFETY_LIMIT ) Then
234             Exit Sub
235         End If
236         safetyCounter = safetyCounter + 1
237         ...
238         RecursiveProc( safetyCounter )
239         End Sub
```

238 In this case, if the routine exceeds the safety limit, it stops recursing.

239 If you don't want to pass the safety counter as an explicit parameter, you could
240 use a *static* variable in C++, Java, or Visual Basic, or the equivalent in other
241 languages.

242 ***Limit recursion to one routine***

243 Cyclic recursion (A calls B calls C calls A) is dangerous because it's hard to
244 detect. Mentally managing recursion in one routine is tough enough;
245 understanding recursion that spans routines is too much. If you have cyclic
246 recursion, you can usually redesign the routines so that the recursion is restricted
247 to a single routine. If you can't and you still think that recursion is the best
248 approach, use safety counters as a recursive insurance policy.

249 ***Keep an eye on the stack***

250 With recursion, you have no guarantees about how much stack space your
251 program uses and it's hard to predict in advance how the program will behave at

252 run time. You can take a couple of steps to control its run-time behavior,
253 however.

254 First, if you use a safety counter, one of the considerations in setting a limit for it
255 should be how much stack you're willing to allocate to the recursive routine. Set
256 the safety limit low enough to prevent a stack overflow.

257 Second, watch for allocation of local variables in recursive functions, especially
258 memory-intensive objects. In other words, use *new* to create objects on the heap
259 rather than letting the compiler create *auto* objects on the stack.

260 ***Don't use recursion for factorials or Fibonacci numbers***

261 One problem with computer-science textbooks is that they present silly examples
262 of recursion. The typical examples are computing a factorial or computing a
263 Fibonacci sequence. Recursion is a powerful tool, and it's really dumb to use it
264 in either of those cases. If a programmer who worked for me used recursion to
265 compute a factorial, I'd hire someone else. Here's the recursive version of the
266 factorial routine:

267 **CODING HORROR**

268 **Java Example of an Inappropriate Solution: Using Recursion to
269 Compute a Factorial**

```
270 int Factorial( int number ) {  
271     if ( number == 1 ) {  
272         return 1;  
273     }  
274     else {  
275         return number * Factorial( number - 1 );  
276     }  
277 }
```

278 In addition to being slow and making the use of run-time memory unpredictable,
279 the recursive version of this routine is harder to understand than the iterative
version. Here's the iterative version:

280 **Java Example of an Appropriate Solution: Using Iteration to Compute a
281 Factorial**

```
282 int Factorial( int number ) {  
283     int intermediateResult = 1;  
284     for ( int factor = 2; factor <= number; factor++ ) {  
285         intermediateResult = intermediateResult * factor;  
286     }  
287     return intermediateResult;  
288 }
```

289 You can draw three lessons from this example. First, computer-science textbooks
290 aren't doing the world any favors with their examples of recursion. Second, and

291 more important, recursion is a much more powerful tool than its confusing use in
292 computing factorials or Fibonacci numbers would suggest. Third, and most
293 important, you should consider alternatives to recursion before using it. You can
294 do anything with stacks and iteration that you can do with recursion. Sometimes
295 one approach works better; sometimes the other does. Consider both before you
296 choose either one.

297 **17.3 goto**

298 CC2E.COM/1785 You might think the debate related to *gos* is extinct, but a quick trip through
299 modern source-code repositories like *SourceForge.net* shows that the *goto* is still
300 alive and well and living deep in your company's server. Moreover, modern
301 equivalents of the *goto* debate still crop up in various guises including debates
302 about multiple returns, multiple loop exits, named loop exits, error processing,
303 and exception handling.

304 Here's a summary of the points on each side of the *goto* debate.

305 **The Argument Against gos**

306 The general argument against *gos* is that code without *gos* is higher-quality
307 code. The famous letter that sparked the original controversy was Edsger
308 Dijkstra's "Go To Statement Considered Harmful" in the March 1968
309 *Communications of the ACM*. Dijkstra observed that the quality of code was
310 inversely proportional to the number of *gos* the programmer used. In
311 subsequent work, Dijkstra has argued that code that doesn't contain *gos* can
312 more easily be proven correct.

313 Code containing *gos* is hard to format. Indentation should be used to show
314 logical structure, and *gos* have an effect on logical structure. Using indentation
315 to show the logical structure of a *goto* and its target, however, is difficult or
316 impossible.

317 Use of *gos* defeats compiler optimizations. Some optimizations depend on a
318 program's flow of control residing within a few statements. An unconditional
319 *goto* makes the flow harder to analyze and reduces the ability of the compiler to
320 optimize the code. Thus, even if introducing a *goto* produces an efficiency at the
321 source-language level, it may well reduce overall efficiency by thwarting
322 compiler optimizations.

323 Proponents of *gos* sometimes argue that they make code faster or smaller. But
324 code containing *gos* is rarely the fastest or smallest possible. Donald Knuth's
325 marvelous, classic article "Structured Programming with go to Statements" gives

326 several examples of cases in which using *gotos* makes for slower and larger code
327 (Knuth 1974).

328 In practice, the use of *gotos* leads to the violation of the principle that code
329 should flow strictly from top to bottom. Even if *gotos* aren't confusing when
330 used carefully, once *gotos* are introduced, they spread through the code like
331 termites through a rotting house. If any *gotos* are allowed, the bad creep in with
332 the good, so it's better not to allow any of them.

333 Overall, experience in the two decades that followed the publication of Dijkstra's
334 letter showed the folly of producing *goto-laden* code. In a survey of the
335 literature, Ben Shneiderman concluded that the evidence supports Dijkstra's
336 view that we're better off without the *goto* (1980), and many modern languages
337 including Java don't even have *gotos*.

338 **The Argument for *gotos***

339 The argument for the *goto* is characterized by an advocacy of its careful use in
340 specific circumstances rather than its indiscriminate use. Most arguments against
341 *gotos* speak against indiscriminate use. The *goto* controversy erupted when
342 Fortran was the most popular language. Fortran had no presentable loop
343 structures, and in the absence of good advice on programming loops with *gotos*,
344 programmers wrote a lot of spaghetti code. Such code was undoubtedly
345 correlated with the production of low-quality programs but has little to do with
346 the careful use of a *goto* to make up for a gap in a modern language's
347 capabilities.

348 A well-placed *goto* can eliminate the need for duplicate code. Duplicate code
349 leads to problems if the two sets of code are modified differently. Duplicate code
350 increases the size of source and executable files. The bad effects of the *goto* are
351 outweighed in such a case by the risks of duplicate code.

352 **CROSS-REFERENCE** For
353 details on using *gotos* in code
354 that allocates resources, see
355 "Error Processing and *gotos*"
356 in this section. See also the
357 discussion of exception
handling in Section 8.4,
"Exceptions."

358
359
360
361

The *goto* is useful in a routine that allocates resources, performs operations on
those resources, and then deallocates the resources. With a *goto*, you can clean
up in one section of code. The *goto* reduces the likelihood of your forgetting to
deallocate the resources in each place you detect an error.

In some cases, the *goto* can result in faster and smaller code. Knuth's 1974
article cited a few cases in which the *goto* produced a legitimate gain.

Good programming doesn't mean eliminating *gotos*. Methodical decomposition,
refinement, and selection of control structures automatically lead to *goto-free*
programs in most cases. Achieving *goto-less* code is not the aim, but the
outcome, and putting the focus on avoiding *gotos* isn't helpful.

362 **The evidence suggests**
363 **only that deliberately**
364 **chaotic control structure**
365 **degrades [programmer]**
366 **performance. These**
367 **experiments provide**
368 **virtually no evidence for**
369 **the beneficial effect of**
370 **any specific method of**
371 **structuring control flow.**

371 — B. A. Sheil

372
373

374

375
376
377
378

379
380
381
382

383
384
385

386
387

388
389
390
391
392
393
394
395
396

397
398

Decades' worth of research with *gotos* failed to demonstrate their harmfulness. In a survey of the literature, B. A. Sheil concluded that unrealistic test conditions, poor data analysis, and inconclusive results failed to support the claim of Shneiderman and others that the number of bugs in code was proportional to the number of *gotos* (1981). Sheil didn't go so far as to conclude that using *gotos* is a good idea—rather that experimental evidence against them was not conclusive.

Finally, the *goto* has been incorporated into many modern languages including Visual Basic, C++ and the Ada language—the most carefully engineered programming language in history. Ada was developed long after the arguments on both sides of the *goto* debate had been fully developed, and after considering all sides of the issue, Ada's engineers decided to include the *goto*.

The Phony *goto* Debate

A primary feature of most *goto* discussions is a shallow approach to the question. The arguer on the “*gotos* are evil” side presents a trivial code fragment that uses *gotos* and then shows how easy it is to rewrite the fragment without *gotos*. This proves mainly that it's easy to write trivial code without *gotos*.

The arguer on the “I can't live without *gotos*” side usually presents a case in which eliminating a *goto* results in an extra comparison or the duplication of a line of code. This proves mainly that there's a case in which using a *goto* results in one less comparison—not a significant gain on today's computers.

Most textbooks don't help. They provide a trivial example of rewriting some code without a *goto* as if that covered the subject. Here's a disguised example of a trivial piece of code from such a textbook:

C++ Example of Code That's Supposed to Be Easy to Rewrite Without *gotos*

```
388 do {  
389     GetData( inputFile, data );  
390     if ( eof( inputFile ) ) {  
391         goto LOOP_EXIT;  
392     }  
393     DoSomething( data );  
394 } while ( data != -1 );  
395  
396 LOOP_EXIT:  
397 
```

The book quickly replaces this code with *gotoless* code:

C++ Example of Supposedly Equivalent Code, Rewritten Without *gotos*

```
398 GetData( inputFile, data );  
399 
```

```
399     while ( ( !eof( inputFile ) ) && ( ( data != -1 ) ) ) do {  
400         DoSomething( data );  
401         GetData( inputFile, data )  
402     }
```

This so-called “trivial” example contains an error. In the case in which *data* equals *-1* entering the loop, the translated code detects the *-1* and exits the loop before executing *DoSomething()*. The original code executes *DoSomething()* before the *-1* is detected. The programming book trying to show how easy it is to code without *gotos* translated its own example incorrectly. But the author of that book shouldn’t feel too bad; other books make similar mistakes. Even the pros have difficulty achieving *gotoless nirvana*.

410 Here’s a faithful translation of the code with no *gotos*:

C++ Example of Truly Equivalent Code, Rewritten Without *gotos*

```
412     do {  
413         GetData( inputFile, data );  
414         if ( !eof( inputFile ) ) {  
415             DoSomething( data );  
416         }  
417     } while ( ( data != -1 ) && ( !eof( InputFile ) ) );
```

Even with a correct translation of the code, the example is still phony because it shows a trivial use of the *goto*. Such cases are not the ones for which thoughtful programmers choose a *goto* as their preferred form of control.

421 It would be hard at this late date to add anything worthwhile to the theoretical
422 *goto* debate. What’s not usually addressed, however, is the situation in which a
423 programmer fully aware of the *gotoless* alternatives chooses to use a *goto* to
424 enhance readability and maintainability.

425 The following sections present cases in which some experienced programmers
426 have argued for using *gotos*. The discussions provide examples of code with
427 *gotos* and code rewritten without *gotos* and evaluate the trade-offs between the
428 versions.

429 Error Processing and *gotos*

430 Writing highly interactive code calls for paying a lot of attention to error
431 processing and cleaning up resources when errors occur. Here’s a code example
432 that purges a group of files. The routine first gets a group of files to be purged,
433 and then it finds each file, opens it, overwrites it, and erases it. The routine
434 checks for errors at each step:

Visual Basic Code with *gos* That Processes Errors and Cleans Up Resources

```
435      ' This routine purges a group of files.
436      Sub PurgeFiles( ByRef errorState As Error_Code )
437          Dim fileIndex As Integer
438          Dim fileToPurge As Data_File
439          Dim fileList As File_List
440          Dim numFilesToPurge As Integer
441
442          MakePurgeFileList( fileList, numFilesToPurge )
443
444          errorState = FileStatus_Success
445          fileIndex = 0
446          While ( fileIndex < numFilesToPurge )
447              fileIndex = fileIndex + 1
448              If Not ( FindFile( fileList( fileIndex ), fileToPurge ) ) Then
449                  errorState = FileStatus_FileFindError
450                  Here's a GoTo.
451                  GoTo END_PROC
452                  End If
453
454                  If Not OpenFile( fileToPurge ) Then
455                      errorState = FileStatus_FileOpenError
456                      Here's a GoTo.
457                      GoTo END_PROC
458                      End If
459
460                      If Not OverwriteFile( fileToPurge ) Then
461                          errorState = FileStatus_FileOverwriteError
462                          Here's a GoTo.
463                          GoTo END_PROC
464                          End If
465
466                          if Erase( fileToPurge ) Then
467                              errorState = FileStatus_FileEraseError
468                              Here's a GoTo.
469                              GoTo END_PROC
470
471
472      Here's the GoTo label.
473      END_PROC:
474          DeletePurgeFileList( fileList, numFilesToPurge )
475      End Sub
```

This routine is typical of circumstances in which experienced programmers decide to use a *goto*. Similar cases come up when a routine needs to allocate and clean up resources like database connections, memory, or temporary files. The alternative to *gos* in those cases is usually duplicating code to clean up the resources. In such cases, a programmer might balance the evil of the *goto* against

480 the headache of duplicate-code maintenance and decide that the *goto* is the lesser
481 evil.

482 You can rewrite the routine above in a couple of ways that avoid *gos*, and both
483 ways involve trade-offs. Here are the possible rewrite strategies:

484 **Rewrite with nested if statements**

485 To rewrite with nested *if* statements, nest the *if* statements so that each is
486 executed only if the previous test succeeds. This is the standard, textbook
487 programming approach to eliminating *gos*. Here's a rewrite of the routine
488 using the standard approach:

489 **CROSS-REFERENCE** C++
490 programmers might point out
491 that this routine could easily
492 be rewritten with *break* and
493 no *gos*. For details, see
494 "Exiting Loops Early" in
495 Section 16.2.

496

497

498

499

500

501 *The While test has been*
502 *changed to add a test for*
503 *errorState.*

504

505

506

507

508

509

510

511

512

513

514

515

516

517

518 *This line is 13 lines away from*
519 *the If statement that invokes*

520 *it.*

521

522

Visual Basic Code That Avoids GoTos by Using Nested ifs

```
' This routine purges a group of files.
Sub PurgeFiles( ByRef errorState As Error_Code )
    Dim fileIndex As Integer
    Dim fileToPurge As Data_File
    Dim fileList As File_List
    Dim numFilesToPurge As Integer

    MakePurgeFileList( fileList, numFilesToPurge )

    errorState = FileStatus_Success
    fileIndex = 0
    While ( fileIndex < numFilesToPurge And errorState = FileStatus_Success )
        fileIndex = fileIndex + 1

        If FindFile( fileList( fileIndex ), fileToPurge ) Then
            If OpenFile( fileToPurge ) Then
                If OverwriteFile( fileToPurge ) Then
                    If Not Erase( fileToPurge ) Then
                        errorState = FileStatus_FileEraseError
                    End If
                Else ' couldn't overwrite file
                    errorState = FileStatus_FileOverwriteError
                End If
            Else ' couldn't open file
                errorState = FileStatus_FileOpenError
            End If
        Else ' couldn't find file
            errorState = FileStatus_FileFindError
        End If
    Wend
    DeletePurgeFileList( fileList, numFilesToPurge )
End Sub
```

523
524
525

526 **CROSS-REFERENCE** For
527 more details on indentation
528 and other coding layout
529 issues, see Chapter 31,
530 “Layout and Style.” For
531 details on nesting levels, see
Section 19.4, “Taming
Dangerously Deep Nesting.”

532
533
534
535
536

537
538
539
540

541
542
543
544
545
546
547
548
549
550
551
552
553 *The While test has been*
554 *changed to add a test for*
555 *errorState.*
556
557
558
559
560
561 *The status variable is tested.*
562
563

For people used to programming without *gotos*, this code might be easier to read than the *goto* version, and If you use it, you won’t have to face an inquisition from the *goto* goon squad.

The main disadvantage of this nested-*If* approach is that the nesting level is deep. Very deep. To understand the code, you have to keep the whole set of nested *ifs* in your mind at once. Moreover, the distance between the error-processing code and the code that invokes it is too great: The code that sets *errorState* to *FileStatus_FileFindError*, for example, is 13 lines from the *If* statement that invokes it.

With the *goto* version, no statement is more than 4 lines from the condition that invokes it. And you don’t have to keep the whole structure in your mind at once. You can essentially ignore any preceding conditions that were successful and focus on the next operation. In this case, the *goto* version is more readable and more maintainable than the nested-*If* version.

Rewrite with a status variable

To rewrite with a status variable (also called a state variable), create a variable that indicates whether the routine is in an error state. In this case, the routine already uses the *errorState* status variable, so you can use that.

Visual Basic Code That Avoids *gotos* by Using a Status Variable

```
' This routine purges a group of files.
Sub PurgeFiles( ByRef errorState As Error_Code )
    Dim fileIndex As Integer
    Dim fileToPurge As Data_File
    Dim fileList As File_List
    Dim numFilesToPurge As Integer

    MakePurgeFileList( fileList, numFilesToPurge )

    errorState = FileStatus_Success
    fileIndex = 0

    While ( fileIndex < numFilesToPurge ) And ( errorState = FileStatus_Success )
        fileIndex = fileIndex + 1

        If Not FindFile( fileList( fileIndex ), fileToPurge ) Then
            errorState = FileStatus_FileFindError
        End If

        If ( errorState = FileStatus_Success ) Then
            If Not OpenFile( fileToPurge ) Then
                errorState = FileStatus_FileOpenError
            End If
        End If
    End While
End Sub
```

```
564             End If
565         End If
566
567     The status variable is tested.
568         If ( errorState = FileStatus_Success ) Then
569             If Not OverwriteFile( fileToPurge ) Then
570                 errorState = FileStatus_FileOverwriteError
571             End If
572         End If
573
574     The status variable is tested.
575         If ( errorState = FileStatus_Success ) Then
576             If Not Erase( fileToPurge ) Then
577                 errorState = FileStatus_FileEraseError
578             End If
579         Wend
580     DeletePurgeFileList( fileList, numFilesToPurge )
581 End Sub
```

The advantage of the status-variable approach is that it avoids the deeply nested *if-then-else* structures of the first rewrite and is thus easier to understand. It also places the action following the *if-then-else* test closer to the test than the nested-*if* approach did and completely avoids *else* clauses.

Understanding the nested-*if* version requires some mental gymnastics. The status-variable version is easier to understand because it closely models the way people think about the problem. You find the file. If everything is OK, you open the file. If everything is still OK, you overwrite the file. If everything is still OK,...

The disadvantage of this approach is that using status variables isn't as common a practice as it should be. Document their use fully, or some programmers might not understand what you're up to. In this example, the use of well-named enumerated types helps significantly.

Rewrite with try-finally

Some languages, including Visual Basic and Java, provide a *try-finally* statement that can be used to clean up resources under error conditions.

To rewrite using the *try-finally* approach, enclose the code that would otherwise need to check for errors inside a *try* block, and place the cleanup code inside a *finally* block. The *try* block specifies the scope of the exception handling, and the *finally* block performs any resource cleanup. The *finally* block will always be called regardless of whether an exception is thrown and regardless of whether the *PurgeFiles()* routine *Catches* any exception that's thrown.

603

Visual Basic Code That Avoids *gos* by Using Try-Finally

```
' This routine purges a group of files. Exceptions are passed to the caller.
Sub PurgeFiles()
    Dim fileIndex As Integer
    Dim fileToPurge As Data_File
    Dim fileList As File_List
    Dim numFilesToPurge As Integer
    MakePurgeFileList( fileList, numFilesToPurge )

    Try
        fileIndex = 0
        While ( fileIndex < numFilesToPurge )
            fileIndex = fileIndex + 1
            FindFile( fileList( fileIndex ), fileToPurge )
            OpenFile( fileToPurge )
            OverwriteFile( fileToPurge )
            Erase( fileToPurge )
        Wend
    Finally
        DeletePurgeFileList( fileList, numFilesToPurge )
    End Try
End Sub
```

624 This approach assumes that all function calls throw exceptions for failures rather
625 than returning error codes.

626 The advantage of the *try-finally* approach is it achieves the visual simplicity of
627 the *goto* approach without the use of *gos*. It also avoids the deeply nested *if-*
628 *then-else* structures.

629 The limitation of the try-finally approach is that it must be implemented
630 consistently throughout a code base. If the code above was part of a code base
631 that used both error codes and exceptions, the code would be required to set an
632 error code for each possible error, and that requirement would make the code
633 above about as complicated as the other approaches. In that context, the *try-*
634 *finally* structure wouldn't be decisively more attractive than the other
635 approaches.

636 A final limitation of this approach is that the *try-finally* statement is not available
637 in all languages.

638 **Comparison of the Approaches**

639 Each of the four methods has something to be said for it. The *goto* approach
640 avoids deep nesting and unnecessary tests but of course has *gos*. The nested-*if*
641 approach avoids *gos* but is deeply nested and gives an exaggerated picture of
642 the logical complexity of the routine. The status-variable approach avoids *gos*

643 and deep nesting but introduces extra tests. The *try-finally* approach avoids both
644 *gotos* and deep nesting, but isn't available in all languages.

645 The *try-finally* approach is the most straightforward in languages that provide
646 *try-finally* and in code bases that haven't already standardized on another
647 approach. If *try-finally* isn't an option, the status-variable approach is slightly
648 preferable to the first two because it's more readable and it models the problem
649 better, but that doesn't make it the best approach in all circumstances.

650 Any of these techniques works well when applied consistently to all the code in a
651 project. Consider all the trade-offs, and then make a project-wide decision about
652 which method to favor.

653 **gotos and Sharing Code in an else Clause**

654 One challenging situation in which some programmers would use a *goto* is the
655 case in which you have two conditional tests and an *else* clause and want to
656 execute code in one of the conditions and in the *else* clause. Here's an example
657 of a case that could drive someone to *goto*:

658 **CODING HORROR**

```
659 if ( statusOk ) {  
660     if ( dataAvailable ) {  
661         importantVariable = x;  
662         goto MID_LOOP;  
663     }  
664 }  
665 else {  
666     importantVariable = GetValue();  
667  
668     MID_LOOP:  
669  
670     // lots of code  
671     ...  
672 }
```

673 This is a good example because it's logically tortuous—it's nearly impossible to
674 read as it stands, and it's hard to rewrite correctly without a *goto*. If you think
675 you can easily rewrite it without *gotos*, ask someone to review your code!
676 Several expert programmers have rewritten it incorrectly.

677 You can rewrite the code in several ways. You can duplicate code, put the
678 common code into a routine and call it from two places, or retest the conditions.
679 In most languages, the rewrite will be a tiny bit larger and slower than the
680 original, but it will be extremely close. Unless the code is in a really hot loop,
681 rewrite it without thinking about efficiency.

682 The best rewrite would be to put the *// lots of code* part into its own routine. Then
683 you can call the routine from the places you would otherwise have used as
684 origins or destinations of *gos* and preserve the original structure of the
685 conditional. Here's how it looks:

C++ Example of Sharing Code in an *else* Clause by Putting Common Code into a Routine

```
688 if ( statusOk ) {  
689     if ( dataAvailable ) {  
690         importantVariable = x;  
691         DoLotsOfCode( importantVariable );  
692     }  
693 }  
694 else {  
695     importantVariable = GetValue();  
696     DoLotsOfCode( importantVariable );  
697 }
```

698 Normally, writing a new routine is the best approach. Sometimes, however, it's
699 not practical to put duplicated code into its own routine. In this case you can
700 work around the impractical solution by restructuring the conditional so that you
701 keep the code in the same routine rather than putting it into a new routine. Here's
702 how it looks:

C++ Example of Sharing Code in an *else* Clause Without a *goto*

```
703 if ( ( statusOk && dataAvailable ) || !statusOk ) {  
704     if ( statusOk && dataAvailable ) {  
705         importantVariable = x;  
706     }  
707     else {  
708         importantVariable = GetValue();  
709     }  
710  
711     // lots of code  
712     ...  
713 }  
714 }
```

715 **CROSS-REFERENCE** Anot
716 her approach to this problem
717 is to use a decision table. For
718 details, see Chapter 18,
719 “Table-Driven Methods.”

This is a faithful and mechanical translation of the logic in the *goto* version. It tests *statusOK* two extra times and *dataAvailable* one, but the code is equivalent. If retesting the conditionals bothers you, notice that the value of *statusOK* doesn't need to be tested twice in the first *if* test. You can also drop the test for *dataAvailable* in the second *if* test.

720

KEY POINT

721 Use of *gos* is a matter of religion. My dogma is that in modern languages, you
722 can easily replace nine out of ten *gos* with equivalent sequential constructs. In
723 these simple cases, you should replace *gos* out of habit. In the hard cases, you
724 can still exorcise the *goto* in nine out of ten cases: You can break the code into
725 smaller routines, use nested *ifs*, test and retest a status variable, or restructure a
726 conditional. Eliminating the *goto* is harder in these cases, but it's good mental
727 exercise and the techniques discussed in this section give you the tools to do it.

728 In the remaining one case out of 100 in which a *goto* is a legitimate solution to
729 the problem, document it clearly and use it. If you have your rain boots on, it's
730 not worth walking around the block to avoid a mud puddle. But keep your mind
731 open to *gotoless* approaches suggested by other programmers. They might see
732 something you don't.

733 Here's a summary of guidelines for using *gos*:

- 734 • Use *gos* to emulate structured control constructs in languages that don't
735 support them directly. When you do, emulate them exactly. Don't abuse the
736 extra flexibility the *goto* gives you.
- 737 • Don't use the *goto* when an equivalent built-in construct is available.
- 738 • CROSS-REFERENCE For
739 details on improving
740 efficiency, see Chapter 25,
741 "Code-Tuning Strategies,"
742 and Chapter 26, "Code-
Tuning Techniques."
- 743 • Limit yourself to one *goto* label per routine unless you're emulating
744 structured constructs.
- 745 • Limit yourself to *gos* that go forward, not backward, unless you're
746 emulating structured constructs.
- 747 • Make sure all *goto* labels are used. Unused labels might be an indication of
748 missing code, namely the code that goes to the labels. If the labels aren't
749 used, delete them.
- 750 • Make sure a *goto* doesn't create unreachable code.
- 751 • If you're a manager, adopt the perspective that a battle over a single *goto*
752 isn't worth the loss of the war. If the programmer is aware of the alternatives
753 and is willing to argue, the *goto* is probably OK.

754 **17.4 Perspective on Unusual Control**

755 **Structures**

756 At one time or another, someone thought that each of the following control
757 structures was a good idea:

- 758 • Unrestricted use of *gos*tos
- 759 • Ability to compute a *goto* target dynamically, and jump to the computed
760 location
- 761 • Ability to use *goto* to jump from the middle of one routine into the middle of
762 another routine
- 763 • Ability to call a routine with a line number or label that allowed execution to
764 begin somewhere in the middle of the routine
- 765 • Ability to have the program generate code on the fly, then execute the code
766 it just wrote

767 At one time, each of these ideas was regarded as acceptable or even desirable,
768 even though now they all look hopelessly quaint, outdated or dangerous. The
769 field of software development has advanced largely through *restricting* what
770 programmers can do with their code. Consequently, I view unconventional
771 control structures with strong skepticism. I suspect that the majority of constructs
772 in this chapter will eventually find their way onto the programmer's scrap heap
773 along with computed *goto* labels, variable routine entry points, self-modifying
774 code, and other structures that favored flexibility and convenience over structure
775 and ability to manage complexity.

CC2E.COM/1792

776 **Additional Resources**

777 **Returns**

778 Fowler, Martin. *Refactoring: Improving the Design of Existing Code*, Reading,
779 Mass.: Addison Wesley, 1999. In the description of the refactoring called
780 “Replace Nested Conditional with Guard Clauses,” Fowler suggests using
781 multiple *return* statements from a routine to reduce nesting in a set of *if*
782 statements. Fowler argues that multiple *returns* are an appropriate means of
783 achieving greater clarity, and that no harm arises from having multiple returns
784 from a routine.

gotos

These articles contain the whole *goto* debate. It erupts from time to time in most workplaces, textbooks, and magazines, but you won't hear anything that wasn't fully explored 20 years ago.

CC2E.COM/1799 Dijkstra, Edsger. "Go To Statement Considered Harmful." *Communications of the ACM* 11, no. 3 (March 1968): 147–48, also available from www.cs.utexas.edu/users/EWD/. This is the famous letter in which Dijkstra put the match to the paper and ignited one of the longest-running controversies in software development.

Wulf, W. A. "A Case Against the GOTO." *Proceedings of the 25th National ACM Conference*, August 1972: 791–97. This paper was another argument against the indiscriminate use of *gotos*. Wulf argued that if programming languages provided adequate control structures, *gotos* would become largely unnecessary. Since 1972, when the paper was written, languages such as C++, Java, and Visual Basic have proven Wulf correct.

Knuth, Donald. "Structured Programming with go to Statements," 1974. In *Classics in Software Engineering*, edited by Edward Yourdon. Englewood Cliffs, N. J.: Yourdon Press, 1979. This long paper isn't entirely about *gotos*, but it includes a horde of code examples that are made more efficient by eliminating *gotos* and another horde of code examples that are made more efficient by adding *gotos*.

Rubin, Frank. "'GOTO Considered Harmful' Considered Harmful." *Communications of the ACM* 30, no. 3 (March 1987): 195–96. In this rather hotheaded letter to the editor, Rubin asserts that *gotoless* programming has cost businesses "hundreds of millions of dollars." He then offers a short code fragment that uses a *goto* and argues that it's superior to *gotoless* alternatives.

The response that Rubin's letter generated was more interesting than the letter itself. For five months, *Communications of the ACM* published letters that offered different versions of Rubin's original seven-line program. The letters were evenly divided between those defending *gotos* and those castigating them. Readers suggested roughly 17 different rewrites, and the rewritten code fully covered the spectrum of approaches to avoiding *gotos*. The editor of *CACM* noted that the letter had generated more response by far than any other issue ever considered in the pages of *CACM*.

For the follow-up letters, see

Communications of the ACM 30, no. 5 (May 1987): 351–55.

821 *Communications of the ACM* 30, no. 6 (June 1987): 475–78.

822 *Communications of the ACM* 30, no. 7 (July 1987): 632–34.

823 *Communications of the ACM* 30, no. 8 (August 1987): 659–62.

824 *Communications of the ACM* 30, no. 12 (December 1987): 997, 1085.

825 CC2E.COM/1706 Clark, R. Lawrence, “A Linguistic Contribution of GOTO-less Programming,”
826 *Datamation*, December 1973. This classic paper humorously argues for
827 replacing the “go to” statement with the “come from” statement. It was also
828 reprinted in the April 1974 edition of *Communications of the ACM*.

CC2E.COM/1713
829 **CHECKLIST: Unusual Control Structures**

830 **return**

- Does each routine use *return* only when necessary?
- Do *returns* enhance readability?

833 **Recursion**

- Does the recursive routine include code to stop the recursion?
- Does the routine use a safety counter to guarantee that the routine stops?
- Is recursion limited to one routine?
- Is the routine’s depth of recursion within the limits imposed by the size of the program’s stack?
- Is recursion the best way to implement the routine? Is it better than simple iteration?

841 **goto**

- Are *gotos* used only as a last resort, and then only to make code more readable and maintainable?
 - If a *goto* is used for the sake of efficiency, has the gain in efficiency been measured and documented?
 - Are *gotos* limited to one label per routine?
 - Do all *gotos* go forward, not backward?
 - Are all *goto* labels used?
-

850

851

852

853

854

855

856

857

Key Points

- Multiple *returns* can enhance a routine's readability and maintainability, and they help prevent deeply nested logic. They should, nevertheless, be used carefully.
- Recursion provides elegant solutions to a small set of problems. Use it carefully, too.
- In a few cases, *gotos* are the best way to write code that's readable and maintainable. Such cases are rare. Use *gotos* only as a last resort.

18

Table-Driven Methods

3 CC2E.COM/1865

4 18.1 General Considerations in Using Table-Driven Methods

5 18.2 Direct Access Tables

6 18.3 Indexed Access Tables

7 18.4 Stair-Step Access Tables

8 18.5 Other Examples of Table Lookups

9 Related Topics

10 Information hiding: “Hide Secrets (Information Hiding)” in Section 5.3

11 Class design: Chapter 6

12 Using decision tables to replace complicated logic: in Section 19.1.

13 Substitute table lookups for complicated expressions: in Section 26.1

14 PROGRAMMERS OFTEN TALK ABOUT “table-driven” methods, but
15 textbooks never tell you what a “table-driven” method is. A table-driven method
16 is a scheme that allows you to look up information in a table rather than using
17 logic statements (*if* and *case*) to figure it out. Virtually anything you can select
18 with logic statements, you can select with tables instead. In simple cases, logic
19 statements are easier and more direct. As the logic chain becomes more complex,
20 tables become increasingly attractive.

21 If you’re already familiar with table-driven methods, this chapter might be just a
22 review. You might examine the “Flexible-Message-Format Example” in Section
23 18.2 for a good example of how an object-oriented design isn’t necessarily better
24 than any other kind of design just because it’s object oriented, and then move on
25 to the discussion of general control issues in Chapter 19.

26 18.1 General Considerations in Using Table- 27 Driven Methods

KEY POINT

28 Used in appropriate circumstances, table-driven code is simpler than complicated
29 logic, easier to modify, and more efficient. Suppose you wanted to classify
30 characters into letters, punctuation marks, and digits, you might use a
31 complicated chain of logic like this one:

32 **Java Example of Using Complicated Logic to Classify a Character**

```
33       if ( ( ( 'a' <= inputChar ) && ( inputChar <= 'z' ) ) ||  
34           ( ( 'A' <= inputChar ) && ( inputChar <= 'Z' ) ) ) {  
35           charType = CharacterType.Letter;  
36       }  
37       else if ( ( inputChar == ' ' ) || ( inputChar == ',' ) ||  
38           ( inputChar == '.' ) || ( inputChar == '!' ) || ( inputChar == '(' ) ||  
39           ( inputChar == ')' ) || ( inputChar == ':' ) || ( inputChar == ';' ) ||  
40           ( inputChar == '?' ) || ( inputChar == '-' ) ) {  
41           charType = CharacterType.Punctuation;  
42       }  
43       else if ( ( '0' <= inputChar ) && ( inputChar <= '9' ) ) {  
44           charType = CharacterType.Digit;  
45       }
```

46 If you used a lookup table instead, you'd store the type of each character in an
47 array that's accessed by type of character. The complicated code fragment above
48 would be replaced by this:

49 **Java Example of Using a Lookup Table to Classify a Character**

```
50       charType = charTypeTable[ inputChar ];
```

51 This fragment assumes that the *charTypeTable* array has been set up earlier. You
52 put your program's knowledge into its data rather than into its logic—in the table
53 instead of in the *if* tests.

54 **Two Issues in Using Table-Driven Methods**

55 When you use table-driven methods, you have to address two issues:

56 **KEY POINT**

57 First you have to address the question of how to look up entries in the table. You
58 can use some data to access a table directly. If you need to classify data by
59 month, for example, keying into a month table is straightforward. You can use an
array with indexes 1 through 12.

60 Other data is too awkward to be used to look up a table entry directly. If you
61 need to classify data by social security number, for example, you can't use the
62 social security number to key into the table directly unless you can afford to
63 store 999-99-9999 entries in your table. You're forced to use a more complicated
64 approach. Here's a list of ways to look up an entry in a table:

65
66
67
68

- Direct access
- Indexed access
- Stair-step access

Each of these kinds of accesses is described in more detail in later subsections.

69 **KEY POINT**

70
71
72
73
74
75

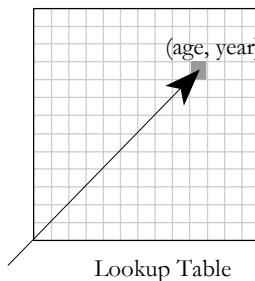
The second issue you have to address if you're using a table-driven method is what you should store in the table. In some cases, the result of a table lookup is data. If that's the case, you can store the data in the table. In other cases, the result of a table lookup is an action. In such a case, you can store a code that describes the action or, in some languages, you can store a reference to the routine that implements the action. In either of these cases, tables become more complicated.

76

18.2 Direct Access Tables

77
78
79
80

Like all lookup tables, direct-access tables replace more complicated logical control structures. They are "direct access" because you don't have to jump through any complicated hoops to find the information you want in the table. As Figure 18-1 suggests, you can pick out the entry you want directly.



81
82
83
84
85

F18xx01

Figure 18-1

As the name suggests, a direct access table allows you to access the table element you're interested in directly.

86
87
88
89

Days-in-Month Example

Suppose you need to determine the number of days per month (forgetting about leap year, for the sake of argument). A clumsy way to do it, of course, is to write a large *if* statement.

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

Visual Basic Example of a Clumsy Way to Determine the Number of Days in a Month

```
If ( month = 1 ) Then  
    days = 31  
ElseIf ( month = 2 ) Then  
    days = 28  
ElseIf ( month = 3 ) Then  
    days = 31  
ElseIf ( month = 4 ) Then  
    days = 30  
ElseIf ( month = 5 ) Then  
    days = 31  
ElseIf ( month = 6 ) Then  
    days = 30  
ElseIf ( month = 7 ) Then  
    days = 31  
ElseIf ( month = 8 ) Then  
    days = 31  
ElseIf ( month = 9 ) Then  
    days = 30  
ElseIf ( month = 10 ) Then  
    days = 31  
ElseIf ( month = 11 ) Then  
    days = 30  
ElseIf ( month = 12 ) Then  
    days = 31  
End If
```

An easier and more modifiable way to perform the same function is to put the data in a table. In Visual Basic, you'd first set up the table:

Visual Basic Example of an Elegant Way to Determine the Number of Days in a Month

```
' Initialize Table of "Days Per Month" Data  
Dim daysPerMonth() As Integer = _  
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

Now, instead of the long *if* statement shown above, you can just use a simple array access to find out the number of days in a month:

Visual Basic Example of an Elegant Way to Determine the Number of Days in a Month (continued)

```
days = daysPerMonth( month-1 )
```

If you wanted to account for leap year in the table-lookup version, the code would still be simple, assuming *LeapYearIndex()* has a value of either 0 or 1:

131
132
133
134
135136
137
138
139

140

141
142
143
144**CODING HORROR**145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171

Visual Basic Example of an Elegant Way to Determine the Number of Days in a Month (continued)

```
days = daysPerMonth( month-1, LeapYearIndex() )
```

In the *if*-statement version, the long string of *ifs* would grow even more complicated if leap year were considered.

Determining the number of days per month is a convenient example because you can use the *month* variable to look up an entry in the table. You can often use the data that would have controlled a lot of *if* statements to access a table directly.

Insurance-Rates Example

Suppose you're writing a program to compute medical-insurance rates, and you have rates that vary by age, gender, marital status, and whether a person smokes. If you had to write a logical control structure for the rates, you'd get something like this:

Java Example of a Clumsy Way to Determine an Insurance Rate

```
if ( gender == Gender.Female ) {
    if ( maritalStatus == MaritalStatus.Single ) {
        if ( smokingStatus == SmokingStatus.NonSmoking ) {
            if ( age < 18 ) {
                rate = 200.00;
            }
            else if ( age == 18 ) {
                rate = 250.00;
            }
            else if ( age == 19 ) {
                rate = 300.00;
            }
            ...
            else if ( 65 < age ) {
                rate = 450.00;
            }
        }
        else {
            if ( age < 18 ) {
                rate = 250.00;
            }
            else if ( age == 18 ) {
                rate = 300.00;
            }
            else if ( age == 19 ) {
                rate = 350.00;
            }
        }
    }
}
```

```
172         ...
173         else if ( 65 < age ) {
174             rate = 575.00;
175         }
176     }
177     else if ( maritalStatus == MaritalStatus.Married )
178     ...
179 }
```

The abbreviated version of the logic structure should be enough to give you an idea of how complicated this kind of thing can get. It doesn't show married females, any males, or most of the ages between 18 and 65. You can imagine how complicated it would get when you programmed the whole rate table.

You might say, "Yeah, but why did you do a test for each age? Why don't you just put the rates in arrays for each age?" That's a good question, and one obvious improvement would be to put the rates into separate arrays for each age.

A better solution, however, is to put the rates into arrays for all the factors, not just age. Here's how you would declare the array in Visual Basic:

189 **Visual Basic Example of Declaring Data to Set Up an Insurance-Rates 190 Table**

```
191       Public Enum SmokingStatus
192           SmokingStatus_First = 0
193           SmokingStatus_Smoking = 0
194           SmokingStatus_NonSmoking = 1
195           SmokingStatus_Last = 1
196       End Enum
197
198       Public Enum Gender
199           Gender_First = 0
200           Gender_Male = 0
201           Gender_Female = 1
202           Gender_Last = 1
203       End Enum
204
205       Public Enum MaritalStatus
206           MaritalStatus_First = 0
207           MaritalStatus_Single = 0
208           MaritalStatus_Married = 1
209           MaritalStatus_Last = 1
210       End Enum
211
212       Const MAX_AGE As Integer = 125
```

214
215
216 **CROSS-REFERENCE** One
217 advantage of a table-driven
218 approach is that you can put
219 the table's data in a file and
220 read it at run time. That
221 allows you to change
something like an insurance-
222 rates table without changing
the program itself. For more
on the idea, see Section 10.6,
223 "Binding Time."

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

```
Dim rateTable ( SmokingStatus_Last, Gender_Last, MaritalStatus_Last, _
    MAX_AGE ) As Double
```

Once you declare the array, you have to figure out some way of putting data into it. You can use assignment statements, read the data from a disk file, compute the data, or do whatever is appropriate. After you've set up the data, you've got it made when you need to calculate a rate. The complicated logic shown earlier is replaced with a simple statement like this one:

Visual Basic Example of an Elegant Way to Determine an Insurance Rate

```
rate = rateTable( smokingStatus, gender, maritalStatus, age )
```

This approach has the general advantages of replacing complicated logic with a table lookup. The table lookup is more readable and easier to change, takes up less space, and executes faster.

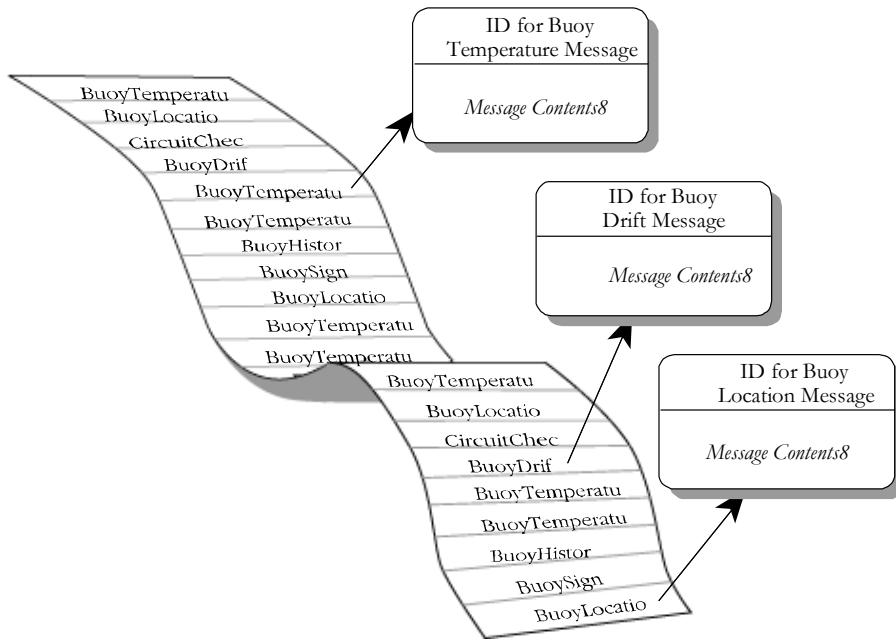
Flexible-Message-Format Example

You can use a table to describe logic that's too dynamic to represent in code. With the character-classification example, the days-in-the-month example, and the insurance-rates example, you at least knew that you could write a long string of *if* statements if you needed to. In some cases, however, the data is too complicated to describe with hard-coded *if* statements.

If you think you've got the idea of how direct-access tables work, you might want to skip the next example. It's a little more complicated than the earlier examples, though, and it further demonstrates the power of table-driven approaches.

Suppose you're writing a routine to print messages that are stored in a file. The file usually has about 500 messages, and each file has about 20 kinds of messages. The messages originally come from a buoy and give water temperature, the buoy's location, and so on.

Each of the messages has several fields, and each message starts with a header that has an ID to let you know which of the 20 or so kinds of messages you're dealing with. Figure 18-2 illustrates how the messages are stored.



244
245
246
247
248

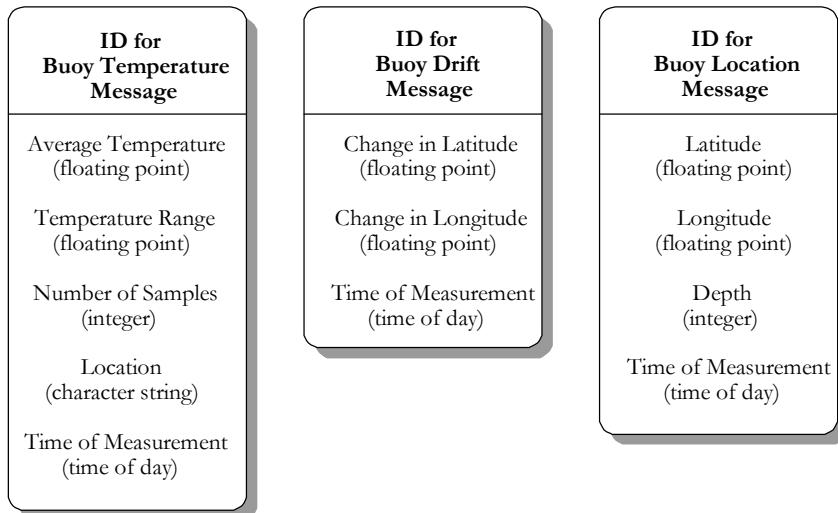
F18xx02

Figure 18-2

Messages are stored in no particular order, and each one is identified with a message ID.

249
250
251

The format of the messages is volatile, determined by your customer, and you don't have enough control over your customer to stabilize it. Figure 18-3 shows what a few of the messages look like in detail.



252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

F18xx03

Figure 18-3

Aside from the Message ID, each kind of message has its own format.

Logic-Based Approach

If you used a logic-based approach, you'd probably read each message, check the ID, and then call a routine that's designed to read, interpret, and print each kind of message. If you had 20 kinds of messages, you'd have 20 routines. You'd also have who-knows-how-many lower-level routines to support them—for example, you'd have a *PrintBuoyTemperatureMessage()* routine to print the buoy temperature message. An object-oriented approach wouldn't be much better: you'd typically use an abstract message object with a subclass for each message type.

Each time the format of any message changed, you'd have to change the logic in the routine or class responsible for that message. In the detailed message above, if the average-temperature field changed from a floating point to something else, you'd have to change the logic of *PrintBuoyTemperatureMessage()*. (If the buoy changed from a "floating point" to something else, you'd have to get a new buoy!)

In the logic-based approach, the message-reading routine consists of a loop to read each message, decode the ID, and then call one of 20 routines based on the message ID. Here's the pseudocode for the logic-based approach:

274 **CROSS-REFERENCE** This
275 low-level pseudocode is used
276 for a different purpose than
277 the pseudocode you use for
278 routine design. For details on
279 designing in pseudocode, see
280 Chapter 9, "The Pseudocode
281 Programming Process."

282
283
284
285
286
287

```
While more messages to read
  Read a message header
  Decode the message ID from the message header
  If the message header is type 1 then
    Print a type 1 message
  Else if the message header is type 2 then
    Print a type 2 message
  ...
  Else if the message header is type 19 then
    Print a type 19 message
  Else if the message header is type 20 then
    Print a type 20 message
```

The pseudocode is abbreviated because you can get the idea without seeing all 20 cases.

288

289
290
291

Object-Oriented Approach

If you were using a rote object-oriented approach, the logic would be hidden in the object inheritance structure, but the basic structure would be just as complicated:

292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310

```
While more messages to read
  Read a message header
  Decode the message ID from the message header
  If the message header is type 1 then
    Instantiate a type 1 message object
  Else if the message header is type 2 then
    Instantiate a type 2 message object
  ...
  Else if the message header is type 19 then
    Instantiate a type 19 message object
  Else if the message header is type 20 then
    Instantiate a type 20 message object
  End if
End While
```

Regardless of whether the logic is written directly or contained within specialized classes, each of the 20 kinds of messages will have its own routine for printing its message. Each routine could also be expressed in pseudocode. Here's the pseudocode for the routine to read and print the buoy temperature message.

311
312
313
314
315
316
317
318

```
Print "Buoy Temperature Message"
Read a floating-point value
Print "Average Temperature"
Print the floating-point value
Read a floating-point value
Print "Temperature Range"
```

```
319     Print the floating-point value  
320  
321     Read an integer value  
322     Print "Number of Samples"  
323     Print the integer value  
324  
325     Read a character string  
326     Print "Location"  
327     Print the character string  
328  
329     Read a time of day  
330     Print "Time of Measurement"  
331     Print the time of day  
332 This is the code for just one kind of message. Each of the other 19 kinds of  
333 messages would require similar code. And if a 21st kind of message was added,  
334 either a 21st routine or a 21st subclass would need to be added—either way a  
335 new message type would require the code to be changed.
```

336 **The Table-Driven Approach**

```
337 The table-driven approach is more economical than this one. The message-  
338 reading routine consists of a loop that reads each message header, decodes the  
339 ID, looks up the message description in the Message array, and then calls the  
340 same routine every time to decode the message.
```

```
341 With a table-driven approach, you can describe the format of each message in a  
342 table rather than hard-coding it in program logic. This makes it easier to code  
343 originally, generates less code, and makes it easier to maintain without changing  
344 code.
```

```
345 To use this approach, you start by listing the kinds of messages and the types of  
346 fields. In C++, you could define the types of all the possible fields this way:
```

347 **C++ Example of Defining Message Data Types**

```
348 enum FieldType {  
349     FieldType_FloatingPoint,  
350     FieldType_Integer,  
351     FieldType_String,  
352     FieldType_TimeOfDay,  
353     FieldType_Boolean,  
354     FieldType_BitField,  
355     FieldType_Last = FieldType_BitField  
356 };
```

```
357 Rather than hard-coding printing routines for each of the 20 kinds of messages,  
358 you can create a handful of routines that print each of the primary data types—  
359 floating point, integer, character string, and so on. You can describe the contents  
360 of each kind of message in a table (including the name of each field) and then
```

361 decode each message based on the description in the table. A table entry to
362 describe one kind of message might look like this:

363 Example of Defining a Message Table Entry

```
364 Message Begin
365     NumFields 5
366     MessageName "Buoy Temperature Message"
367     Field 1, FloatingPoint, "Average Temperature"
368     Field 2, FloatingPoint, "Temperature Range"
369     Field 3, Integer, "Number of Samples"
370     Field 4, String, "Location"
371     Field 5, TimeOfDay, "Time of Measurement"
372 Message End
```

373 This table could be hardcoded in the program (in which case each of the elements
374 shown would be assigned to variables), or it could be read from a file at program
375 startup time or later.

376 Once message definitions are read into the program, instead of having all the
377 information embedded in a program's logic you have it embedded in data. Data
378 tends to be more flexible than logic. Data is easy to change when a message
379 format changes. If you have to add a new kind of message, you can just add
380 another element to the data table.

381 Here's the pseudocode for the top-level loop in the table-driven approach:

382 *The first three lines here are*
383 *the same as in the logic-*
384 *based approach.*
385 While more messages to read
386 Read a message header
387 Decode the message ID from the message header
388 Look up the message description in the message-description table
389 Read the message fields and print them based on the message description
390 End While

391 Unlike the pseudocode for the logic-based approach, the pseudocode in this case
392 isn't abbreviated because the logic is so much less complicated. In the logic
393 below this level, you'll find one routine that's capable of interpreting a message
394 description from the message description table, reading message data, and
395 printing a message. That routine is more general than any of the logic-based
396 message-printing routines but not much more complicated, and it will be one
397 routine instead of 20:

```
398     While more fields to print
399         Get the field type from the message description
400         case ( field type )
401             of ( floating point )
402                 read a floating-point value
403                 print the field label
404                 print the floating-point value
```

```
402
403          of ( integer )
404              read an integer value
405              print the field label
406              print the integer value
407
408          of ( character string )
409              read a character string
410              print the field label
411              print the character string
412
413          of ( time of day )
414              read a time of day
415              print the field label
416              print the time of day
417
418          of ( boolean )
419              read a single flag
420              print the field label
421              print the single flag
422
423          of ( bit field )
424              read a bit field
425              print the field label
426              print the bit field
427      End Case
428  End While
429
430  Admittedly, this routine with its six cases is longer than the single routine needed
431  to print the buoy temperature message. But this is the only routine you need. You
432  don't need 19 other routines for the 19 other kinds of messages. This routine
433  handles the six field types and takes care of all the kinds of messages.
```

```
434
435  This routine also shows the most complicated way of implementing this kind of
436  table lookup because it uses a case statement. Another approach would be to
437  create an abstract class AbstractField and then create subclasses for each field
438  type. You won't need a case statement; you can call the member routine of the
439  appropriate type of object.
```

440 Here's how you would set up the object types in C++:

441 **C++ Example of Setting Up Object Types**

```
442 class AbstractField {
443     public:
444         virtual void ReadAndPrint( string, FileStatus & ) = 0;
445     }
446
447     class FloatingPointField : public AbstractField {
```

```
446     public:  
447         virtual void ReadAndPrint( string, FileStatus & ) {  
448             ...  
449         }  
450  
451         class IntegerField ...  
452         class StringField ...  
453         ...  
454     }
```

This code fragment declares a member routine for each class that has a string parameter and a *FileStatus* parameter.

The second step is to declare an array to hold the set of objects. The array is the lookup table, and here's how it looks:

C++ Example of Setting Up a Table to Hold an Object of Each Type

```
460     AbstractField* field[ Field_Last ];
```

The final step required to set up the table of objects is to assign the names of specific objects to the *Field* array. Here's how those assignments would look:

C++ Example of Setting Up a List of Objects

```
464     field[ Field_FloatingPoint ] = new FloatingPointField();  
465     field[ Field_Integer ] = new IntegerField();  
466     field[ Field_String ] = new StringField();  
467     field[ Field_TimeOfDay ] = new TimeOfDayField();  
468     field[ Field_Boolean ] = new BooleanField();  
469     field[ Field_BitField ] = new BitFieldField();
```

This code fragment assumes that *FloatingPointField* and the other identifiers on the right side of the assignment statements are names of objects of type *AbstractField*. Assigning the objects to array elements in the array means that you can call the right *ReadAndPrint()* routine by referencing an array element instead of by using a specific kind of object directly.

Once the table of routines is set up, you can handle a field in the message simply by accessing the table of objects and calling one of the member routines in the table. The code looks like this:

C++ Example of Looking Up Objects and Member Routines in a Table

```
479 This stuff is just housekeeping  
480 for each field in a message.  
481  
482     messageIdx = 1;  
483     while ( ( messageIdx <= numFieldsInMessage ) and ( fileStatus == OK ) ) {  
484         fieldType = fieldDescription[ messageIdx ].FieldType;  
485         fieldName = fieldDescription[ messageIdx ].FieldName;  
486         field[ fieldType ].ReadAndPrint( fieldName, fileStatus );
```

484 *This is the table lookup that
485 calls a routine depending on
486 the type of the field—just by
487 looking it up in a table of
objects.*

488
489
490

491 Remember the original 34 lines of table-lookup pseudocode containing the *case*
492 statement? If you replace the *case* statement with a table of objects, this is all the
493 code you'd need to provide the same functionality. Incredibly, it's also all the
 code needed to replace all 20 of the individual routines in the logic-based
 approach. Moreover, if the message descriptions are read from a file, new
 message types won't require code changes unless there's a new field type.

491
492
493

494 You can use this approach in any object-oriented language. It's less error prone,
495 more maintainable, and more efficient than lengthy *if* statements, *case*
496 statements, or copious subclasses.
497
498
499

The fact that a design uses inheritance and polymorphism doesn't make it a good
design. The "rote object-oriented design" example described earlier would
require as much code as a rote functional design—or more. That approach made
the solution space more complicated, rather than less. The key design insight in
this case is neither object-orientation nor functional orientation—but the use of a
well-thought-out lookup table.

500

Fudging Lookup Keys

501 In each of the three previous examples, you could use the data to key into the
502 table directly. That is, you could use *messageID* as a key without alteration, as
503 you could use *month* in the days-per-month example and *gender*, *maritalStatus*,
504 and *smokingStatus* in the insurance-rates example.

505
506
507
508
509
510

You'd always like to key into a table directly because it's simple and fast.
Sometimes, however, the data isn't cooperative. In the insurance-rates example,
Age wasn't well behaved. The original logic had one rate for people under 18,
individual rates for ages 18 through 65, and one rate for people over 65. This
meant that for ages 0 through 17 and 66 and over, you couldn't use the age to
key directly into a table that stored only one set of rates for several ages.

511
512

This leads to the topic of fudging table-lookup keys. You can fudge keys in
several ways:

513
514
515
516
517
518
519
520

Duplicate information to make the key work directly

One straightforward way to make *age* work as a key into the rates table is to
duplicate the under-18 rates for each of the ages 0 through 17 and then use the
age to key directly into the table. You can do the same thing for ages 66 and
over. The benefits of this approach are that the table structure itself is
straightforward and the table accesses are, straightforward. If you needed to add
age-specific rates for ages 17 and below, you could just change the table. The
drawbacks are that the duplication would waste space for redundant information

521 and increase the possibility of errors in the table—if only because the table
522 would contain redundant data.

523 ***Transform the key to make it work directly***

524 A second way to make *Age* work as a direct key is to apply a function to *Age* so
525 that it works well. In this case, the function would have to change all ages 0
526 through 17 to one key, say 17, and all ages above 66 to another key, say 66. This
527 particular range is well behaved enough that you could just use *min()* and *max()*
528 functions to make the transformation. For example, you could use the
529 expression

530 `max(min(66, Age), 17)`
531 to create a table key that ranges from 17 to 66.

532 Creating the transformation function requires that you recognize a pattern in the
533 data you want to use as a key, and that's not always as simple as using the *min()*
534 and *max()* routines. Suppose that in this example the rates were for five-year age
535 bands instead of one-year bands. Unless you wanted to duplicate all your data
536 five times, you'd have to come up with a function that divided *Age* by 5 properly
537 and used the *min()* and *max()* routines.

538 ***Isolate the key-transformation in its own routine***

539 Anytime you have to fudge data to make it work as a table key, put the operation
540 that changes the data to a key into its own routine. A routine eliminates the
541 possibility of using different transformations in different places. It makes
542 modifications easier when the transformation changes. A good name for the
543 routine, like *KeyFromAge()*, also clarifies and documents the purpose of the
544 mathematical machinations.

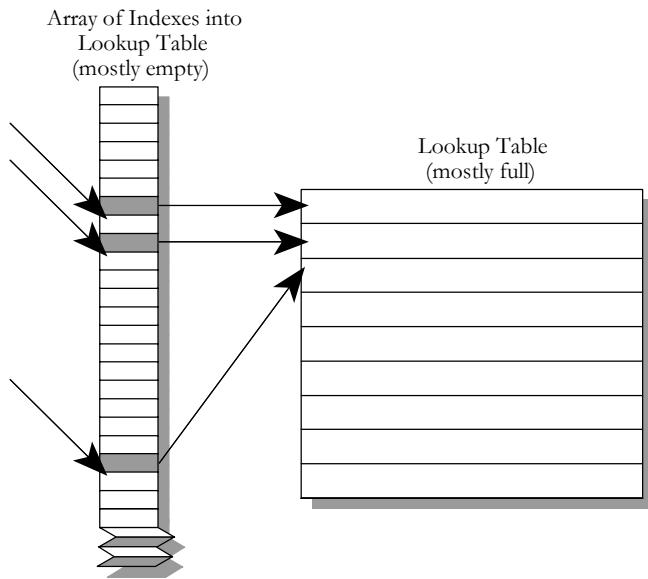
545 18.3 Indexed Access Tables

546 Sometimes a simple mathematical transformation isn't powerful enough to make
547 the jump from data like *Age* to a table key. Some such cases are suited to the use
548 of an indexed access scheme.

549 When you use indexes, you use the primary data to look up a key in an index
550 table and then you use the value from the index table to look up the main data
551 you're interested in.

552 Suppose you run a warehouse and have an inventory of about 100 items.
553 Suppose further that each item has a four-digit part number that ranges from
554 0000 through 9999. In this case, if you want to use the part number to key
555 directly into a table that describes some aspect of each item, you set up an index

556 array with 10,000 entries (from 0 through 9999). The array is empty except for
557 the 100 entries that correspond to part numbers of the 100 items in your ware-
558 house. As Figure 18-4 shows, those entries point to an item-description table that
559 has far fewer than 10,000 entries.



F18xx04

Figure 18-4

Rather than being accessed directly, an indexed access table is accessed via an intermediate index.

560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
Indexed access schemes offer two main advantages. First, if each of the entries in
the main lookup table is large, it takes a lot less space to create an index array
with a lot of wasted space than it does to create a main lookup table with a lot of
wasted space. For example, suppose that the main table takes 100 bytes per entry
and that the index array takes 2 bytes per entry. Suppose that the main table has
100 entries and that the data used to access it has 10,000 possible values. In such
a case, the choice is between having an index with 10,000 entries or a main data
member with 10,000 entries. If you use an index, your total memory use is
30,000 bytes. If you forgo the index structure and waste space in the main table,
your total memory use is 1,000,000 bytes.

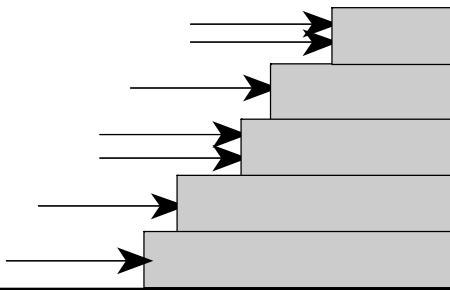
575
576
577
578
579
The second advantage, even if you don't save space by using an index, is that it's
sometimes cheaper to manipulate entries in an index than entries in a main table.
For example, if you have a table with employee names, hiring dates, and salaries,
you can create one index that accesses the table by employee name, another that
accesses the table by hiring date, and a third that accesses the table by salary.

580 A final advantage of an index-access scheme is the general table-lookup
581 advantage of maintainability. Data encoded in tables is easier to maintain than
582 data embedded in code. To maximize the flexibility, put the index-access code in
583 its own routine and call the routine when you need to get a table key from a part
584 number. When it's time to change the table, you might decide to switch the
585 index-accessing scheme or to switch to another table-lookup scheme altogether.
586 The access scheme will be easier to change if you don't spread index accesses
587 throughout your program.

588 18.4 Stair-Step Access Tables

589 Yet another kind of table access is the stair-step method. This access method
590 isn't as direct as an index structure, but it doesn't waste as much data space.

591 The general idea of stair-step structures, illustrated in Figure 18-5, is that entries
592 in a table are valid for ranges of data rather than for distinct data points.



593 F18xx05

594 **Figure 18-5**

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

597 For example, if you're writing a grading program, the "B" entry range might be
598 from 75 percent to 90 percent. Here's a range of grades you might have to
599 program someday:

$\geq 90.0\%$	A
< 90.0%	B
< 75.0%	C
< 65.0%	D
< 50.0%	F

600 This is an ugly range for a table lookup because you can't use a simple data-
601 transformation function to key into the letters *A* through *F*. An index scheme

603 would be awkward because the numbers are floating point. You might consider
604 converting the floating-point numbers to integers, and in this case that would be
605 a valid design option, but for the sake of illustration, this example will stick with
606 floating point.

607 To use the stair-step method, you put the upper end of each range into a table
608 and then write a loop to check a score against the upper end of each range. When
609 you find the point at which the score first exceeds the top of a range, you know
610 what the grade is. With the stair-step technique, you have to be careful to handle
611 the endpoints of the ranges properly. Here's the code in Visual Basic that assigns
612 grades to a group of students based on this example:

613 **Visual Basic Example of a Stair-Step Table Lookup**

```
614 ' set up data for grading table
615 Dim rangeLimit() As Double = { 50.0, 65.0, 75.0, 90.0, 100.0 }
616 Dim grade() As String =      { "F",   "D",   "C",   "B",   "A"  }
617 maxGradeLevel = grade.Length - 1
618 ...
619
620 ' assign a grade to a student based on the student's score
621 gradeLevel = 0
622 studentGrade = "A"
623 While ( ( studentGrade = "A" ) and ( gradeLevel < maxGradeLevel ) )
624     If ( studentScore < rangeLimit( gradeLevel ) ) Then
625         studentGrade = grade( gradeLevel )
626     End If
627     gradeLevel = gradeLevel + 1
628 Wend
```

629 Although this is a simple example, you can easily generalize it to handle multiple
630 students, multiple grading schemes (for example, different grades for different
631 point levels on different assignments), and changes in the grading scheme.

632 The advantage of this approach over other table-driven methods is that it works
633 well with irregular data. The grading example is simple in that, although grades
634 are assigned at irregular intervals, the numbers are “round,” ending with 5s and
635 0s. The stair-step approach is equally well suited to data that doesn’t end neatly
636 with 5s and 0s. You can use the stair-step approach in statistics work for proba-
637 bility distributions with numbers like this:

Probability	Insurance Claim Amount
0.458747	\$0.00
0.547651	\$254.32

0.627764	\$514.77
0.776883	\$747.82
0.893211	\$1,042.65
0.957665	\$5,887.55
0.976544	\$12,836.98
0.987889	\$27,234.12

...

638 Ugly numbers like these defy any attempt to come up with a function to neatly
639 transform them into table keys. The stair-step approach is the answer.

640 This approach also enjoys the general advantages of table-driven approaches. It
641 is flexible and modifiable. If the grading ranges in the grading example were to
642 change, the program could easily be adapted by modifying the entries in the
643 *RangeLimit* array. You could easily generalize the grade-assignment part of the
644 program so that it would accept a table of grades and corresponding cut-off
645 scores. The grade-assignment part of the program wouldn't have to use scores
646 expressed as percentages; it could use raw points rather than percentages, and the
647 program wouldn't have to change much.

648 Here are a few subtleties to consider as you use the stair-step technique:

649 ***Watch the endpoints***

650 Make sure you've covered the case at the top end of each stair-step range. Run
651 the stair-step search so that it finds items that map to any range other than the
652 uppermost range, and then have the rest fall into the uppermost range.
653 Sometimes this requires creating an artificial value for the top of the uppermost
654 range.

655 Be careful too about mistaking < for <=. Make sure that the loop terminates
656 properly with values that fall into the top ranges and that the range boundaries
657 are handled correctly.

658 ***Consider using a binary search rather than a sequential search***

659 In the grading example, the loop that assigns the grade searches sequentially
660 through the list of grading limits. If you had a larger list, the cost of the
661 sequential search might become prohibitive. If it does, you can replace it with a
662 quasi-binary search. It's a "quasi" binary search because the point of most binary
663 searches is to find a value. In this case, you don't expect to find the value; you
664 expect to find the right category for the value. The binary-search algorithm must
665 correctly determine where the value should go. Remember also to treat the
666 endpoint as a special case.

667 ***Consider using indexed access instead of the stair-step technique***
668 An index-access scheme such as the ones described in the preceding section
669 might be a good alternative to a stair-step technique. The searching required in
670 the stair-step method can add up, and if execution speed is a concern, you might
671 be willing to trade the space an extra index structure takes up for the time
672 advantage you get with a more direct access method.

673 Obviously, this alternative isn't a good choice in all cases. In the grading
674 example, you could probably use it; if you had only 100 discrete percentage
675 points, the memory cost of setting up an index array wouldn't be prohibitive. If,
676 on the other hand, you had the probability data mentioned above, you couldn't
677 set up an indexing scheme because you can't key into entries with numbers like
678 0.458747 and 0.547651.

679 In some cases, any of the several options might work. The point of design is
680 choosing one of the several good options for your case. Don't worry too much
681 about choosing the best one. As Butler Lampson, a distinguished engineer at
682 Microsoft, says, it's better to strive for a good solution and avoid disaster rather
683 than trying to find the best solution (Lampson 1984).

684 ***Put the stair-step table lookup into its own routine***
685 When you create a transformation function that changes a value like
686 *StudentGrade* into a table key, put it into its own routine.

687 18.5 Other Examples of Table Lookups

688 A few other examples of table lookups appear in other sections of the book.
689 They're used in the course of discussing other techniques, and the contexts don't
690 emphasize the table lookups per se. Here's where you'll find them:

- 691 • Looking up rates in an insurance table: Section 16.3, "Creating Loops
692 Easily—from the Inside Out"
- 693 • Using decision tables to replace complicated logic: "Use decision tables to
694 replace complicated conditions" in Section 19.1.
- 695 • Cost of memory paging during a table lookup: Section 25.3, "Kinds of Fat
696 and Molasses"
- 697 • Combinations of boolean values (A or B or C): "Substitute Table Lookups
698 for Complicated Expressions" in Section 26.1
- 699 • Precomputing values in a loan repayment table: Section 26.4, "Expressions."

700

701

702

703

704

705

706

707

708

709

710

CHECKLIST: Table-Driven Methods

- Have you considered table-driven methods as an alternative to complicated logic?
 - Have you considered table-driven methods as an alternative to complicated inheritance structures?
 - Have you considered storing the table's data externally and reading it at run time so that the data can be modified without changing code?
 - If the table cannot be accessed directly via a straightforward array index (as in the *Age* example), have you put the access-key calculation into a routine rather than duplicating the index calculation in the code?
-

711

Key Points

- Tables provide an alternative to complicated logic and inheritance structures. If you find that you're confused by a program's logic or inheritance tree, ask yourself whether you could simplify by using a lookup table.
- One key consideration in using a table is deciding how to access the table. You can access tables using direct access, indexed access, or stair-step access.
- Another key consideration in using a table is deciding what exactly to put into the table.

19

2 General Control Issues

3 CC2E.COM/1978

4 **Contents**

- 5 19.1 Boolean Expressions
- 6 19.2 Compound Statements (Blocks)
- 7 19.3 Null Statements
- 8 19.4 Taming Dangerously Deep Nesting
- 9 19.5 A Programming Foundation: Structured Programming
- 10 19.6 Control Structures and Complexity

11 **Related Topics**

12 Straight-line code: Chapter 14

13 Code with conditionals: Chapter 15

14 Code with loops: Chapter 16

15 Unusual control structures: Chapter 17

16 Complexity in software development: “Software’s Primary Technical Imperative: Managing Complexity” in Section 5.2.

17 NO DISCUSSION OF CONTROL WOULD BE COMPLETE unless it went
18 into several general issues that crop up when you think about control constructs.
19 Most of the information in this chapter is detailed and pragmatic. If you’re
20 reading for the theory of control structures rather than for the gritty details,
21 concentrate on the historical perspective on structured programming in Section
22 19.5 and on the relationships between control structures in Section 19.6.

23 **19.1 Boolean Expressions**

24 Except for the simplest control structure, the one that calls for the execution of
25 statements in sequence, all control structures depend on the evaluation of
26 boolean expressions.

27

Using *True* and *False* for Boolean Tests

28

Use the identifiers *True* and *False* in boolean expressions rather than using flags like *0* and *1*. Most modern languages have a boolean data type and provide predefined identifiers for true and false. They make it easy—they don't even allow you to assign values other than *True* or *False* to boolean variables. Languages that don't have a boolean data type require you to have more discipline to make boolean expressions readable. Here's an example of the problem:

29

30

31

32

33

34

CODING HORROR

35

Visual Basic Examples of Using Ambiguous Flags for Boolean Values

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

```
Dim printerError As Integer
Dim reportSelected As Integer
Dim summarySelected As Integer
...
If printerError = 0 Then InitializePrinter()
If printerError = 1 Then NotifyUserOfError()

If reportSelected = 1 Then PrintReport()
If summarySelected = 1 Then PrintSummary()

If printerError = 0 Then CleanupPrinter()
```

If using flags like *0* and *1* is common practice, what's wrong with it? It's not clear from reading the code whether the function calls are executed when the tests are true or when they're false. Nothing in the code fragment itself tells you whether *1* represents true and *0* false or whether the opposite is true. It's not even clear that the values *1* and *0* are being used to represent true and false. For example, in the *If reportSelected = 1* line, the *1* could easily represent the first report, a 2 the second, a 3 the third; nothing in the code tells you that *1* represents either true or false. It's also easy to write *0* when you mean *1* and vice versa.

56

57

58

59

Use terms named *True* and *False* for tests with boolean expressions. If your language doesn't support such terms directly, create them using preprocessor macros or global variables. The code example is rewritten below using Visual Basic's built-in *True* and *False*:

60

61

Good Visual Basic Examples of Using *True* and *False* for Tests Instead of Numeric Values

62

63

64

65

66

```
Dim printerError As Boolean
Dim reportSelected As ReportType
Dim summarySelected As Boolean
...
If ( printerError = False ) Then InitializePrinter()
```

```
67     If ( printerError = True ) Then NotifyUserOfError()  
68  
69     If ( reportSelected = ReportType_First ) Then PrintReport()  
70     If ( summarySelected = True ) Then PrintSummary()  
71  
72     If ( printerError = False ) Then CleanupPrinter()  
73 Use of the True and False constants makes the intent clearer. You don't have to  
74 remember what 1 and 0 represent, and you won't accidentally reverse them.  
75 Moreover, in the rewritten code, it's now clear that some of the 1s and 0s in the  
76 original Visual Basic example weren't being used as boolean flags. The If  
77 reportSelected = 1 line was not a boolean test at all; it tested whether the first  
78 report had been selected.
```

79 This approach tells the reader that you're making a boolean test; it's harder to
80 write *True* when you mean *False* than it is to write *1* when you mean *0*, and you
81 avoid spreading the magic numbers *0* and *1* throughout your code. Here are some
82 tips on defining *True* and *False* in boolean tests:

83 **Compare boolean values to True and False implicitly**

84 If your language supports boolean variables, you can write clearer tests by
85 treating the expressions as boolean expressions. For example, write

```
86     while ( not done ) ...  
87         while ( a = b ) ...  
88 rather than
```

```
89     while ( done = False ) ...  
90         while ( (a = b) = True ) ...
```

91 Using implicit comparisons reduces the number of terms that someone reading
92 your code has to keep in mind, and the resulting expressions read more like
93 conversational English. The example above could be rewritten with even better
94 style like this:

95 **Better Visual Basic Examples of Using *True* and *False* for Tests Instead 96 of Numeric Values**

```
97 Dim printerError As Boolean  
98 Dim reportSelected As ReportType  
99 Dim summarySelected As Boolean  
100 ...  
101 If ( Not printerError ) Then InitializePrinter()  
102 If ( printerError ) Then NotifyUserOfError()  
103  
104 If ( reportSelected = ReportType_First ) Then PrintReport()  
105 If ( summarySelected ) Then PrintSummary()
```

107 **If (Not printerError) Then CleanupPrinter()**
108 If your language doesn't support boolean variables and you have to emulate
109 them, you might not be able to use this technique because emulations of *True*
110 and *False* can't always be tested with statements like *while (not done)*.

111 **In C, use the 1==1 trick to define TRUE and FALSE**
112 In C, sometimes it's hard to remember whether *TRUE* equals 1 and *FALSE*
113 equals 0 or vice versa. You could remember that testing for *FALSE* is the same
114 as testing for a null terminator or another zero value. Otherwise, an easy way to
115 avoid the problem is to define *TRUE* and *FALSE* as follows:

116 **C Example of Easy-to-Remember Boolean Definitions**

```
117          #define TRUE  (1==1)  
118          #define FALSE (!TRUE)
```

CROSS-REFERENCE For details, see Section 12.5, "Boolean Variables."

120

121 **Making Complicated Expressions Simple**

122 You can take several steps to simplify complicated expressions.

123 **Break complicated tests into partial tests with new boolean variables**

Rather than creating a monstrous test with half a dozen terms, assign

intermediate values to terms that allow you to perform a simpler test.

139 **CROSS-REFERENCE** For
140 details on the technique of
141 using intermediate variables
142 to clarify a boolean test, see
143 “Use boolean variables to
144 document your program” in
Section 12.5.

145

146

147

148

149

150 *Intermediate variables are*
151 *introduced here to clarify the*
152 *test on the final line, below.*

153

154

155

156

157

158

159

160

161

162

163

164 **KEY POINT**

165

166

167

168

169

170

171 **CROSS-REFERENCE** For
172 details on using tables as
173 substitutes for complicated
174 logic, see Chapter 18, “Table-
Driven Methods.”

175

176

177

178

Visual Basic Example of a Complicated Test Moved Into a Boolean Function, With New Intermediate Variables To Make the Test Clearer

```
Function DocumentIsValid( _
    ByRef documentToCheck As Document, _
    lineCount As Integer, _
    inputError As Boolean _
) As Boolean

    Dim allDataRead As Boolean
    Dim legalLineCount As Boolean

    allDataRead = ( documentToCheck.AtEndOfStream ) And ( Not inputError )
    legalLineCount = ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES )

    DocumentIsValid = allDataRead And legalLineCount And ( Not ErrorProcessing() )
End Function
```

This example assumes that *ErrorProcessing()* is a boolean function that indicates the current processing status. Now, when you read through the main flow of the code, you don’t have to read the complicated test:

Visual Basic Example of the Main Flow of the Code Without the Complicated Test

```
If ( DocumentIsValid( document, lineCount, inputError ) ) Then
    ' do something or other
    ...
End If
```

If you use the test only once, you might not think it’s worthwhile to put it into a routine. But putting the test into a well-named function improves readability and makes it easier for you to see what your code is doing, and that is a sufficient reason to do it. The new function name introduces an abstraction into the program which documents the purpose of the test *in code*. That’s even better than documenting the test with comments because the code is more likely to be read than the comments and it’s more likely to be kept up to date too.

Use decision tables to replace complicated conditions

Sometimes you have a complicated test involving several variables. It can be helpful to use a decision table to perform the test rather than using *ifs* or *cases*. A decision-table lookup is easier to code initially, having only a couple of lines of code and no tricky control structures. This minimization of complexity minimizes the opportunity for mistakes. If your data changes, you can change a decision table without changing the code; you only need to update the contents of the data structure.

179

180 **I ain't not no undummy.**

181 — Homer Simpson

182

183

184

185

186

187

188 Here's the negative not.

189

190

191

192

193

194

195

196

197

198 The test in this line has been

199 The code in this block has
200 been switched ...

201

202

203 ...with the code in this block.

204

205

CROSS-REFERENCE The recommendation to frame boolean expressions positively sometimes contradicts the recommendation to code the nominal case after the *if* rather than the *else*. (See Section 15.1, “*if Statements.”*) In such a case, you have to think about the benefits of each approach and decide which is better for your situation.

Forming Boolean Expressions Positively

Not a few people don't have not any trouble understanding a nonshort string of nonpositives—that is, most people have trouble understanding a lot of negatives. You can do several things to avoid complicated negative boolean expressions in your programs.

In if statements, convert negatives to positives and flip-flop the code in the if and else clauses

Here's an example of a negatively expressed test:

Java Example of a Confusing Negative Boolean Test

```
if ( !statusOK ) {  
    // do something  
    ...  
}  
else {  
    // do something else  
    ...  
}
```

You can change this to the following positively expressed test:

Java Example of a Clearer Positive Boolean Test

```
if ( statusOK ) {  
    // do something else  
    ...  
}  
else {  
    // do something  
    ...  
}
```

The second code fragment is logically the same as the first but is easier to read because the negative expression has been changed to a positive.

Alternatively, you could choose a different variable name, one that would reverse the truth value of the test. In the example, you could replace *statusOK* with *ErrorDetected*, which would be true when *statusOK* was false.

Apply DeMorgan's Theorems to simplify boolean tests with negatives

DeMorgan's Theorems let you exploit the logical relationship between an expression and a version of the expression that means the same thing because it's doubly negated. For example, you might have a code fragment that contains the following test:

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

231 **CROSS-REFERENCE** For
 232 an example of using
 233 parentheses to clarify other
 234 kinds of expressions, see
 235 “Parentheses” in Section
 236 31.2.

237

238

239

240

241

Java Example of a Negative Test

```
if ( !displayOK || !printerOK ) ...
```

This is logically equivalent to the following:

Java Example After Applying DeMorgan's Theorem

```
if ( !( displayOK && printerOK ) ) ...
```

Here you don't have to flip-flop *if* and *else* clauses; the expressions in the last two code fragments are logically equivalent. To apply DeMorgan's Theorems to the logical operator *and* or the logical operator *or* and a pair of operands, you negate each of the operands, switch the *ands* and *ors*, and negate the entire expression. Table 19-1 summarizes the possible transformations under DeMorgan's Theorems:

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

Initial Expression	Equivalent Expression
not A and not B	not (A or B)
not A and B	not (A or not B)
A and not B	not (not A or B)
A and B	not (not A or not B)
not A or not B*	not (A and B)
not A or B	not (A and not B)
A or not B	not (not A and B)
A or B	not (not A and not B)

* This is the expression used in the example.

Using Parentheses to Clarify Boolean Expressions

If you have a complicated boolean expression, rather than relying on the language's evaluation order, parenthesize to make your meaning clear. Using parentheses makes less of a demand on your reader, who might not understand the subtleties of how your language evaluates boolean expressions. If you're smart, you won't depend on your own or your reader's in-depth memorization of evaluation precedence—especially when you have to switch among two or more languages. Using parentheses isn't like sending a telegram: you're not charged for each character—the extra characters are free.

Here's an expression with too few parentheses:

Java Example of an Expression Containing Too Few Parentheses

```
if ( a < b == c == d ) ...
```

242 This is a confusing expression to begin with, and it's even more confusing
243 because it's not clear whether the coder means to test $(a < b) == (c == d)$ or $(a < b) == c == d$. The following version of the expression is still a little
244 confusing, but the parentheses help:
245

Java Example of an Expression Better Parenthesized

```
246 if ((a < b) == (c == d)) ...
```

247 In this case, the parentheses help readability and the program's correctness—the
248 compiler wouldn't have interpreted the first code fragment this way. When in
249 doubt, parenthesize.

250 **CROSS-REFERENCE** Many programmer-oriented text
251 editors have commands that
252 match parentheses, brackets,
253 and braces. For details on
254 programming editors, see
255 “Editing” in Section 30.2.
256

257

Java Example of Balanced Parentheses

258 *Read this.*

```
259 if (((a < b) == (c == d)) && !done) ...  
260 | | | | | | | |  
261 0 1 2 3 2 3 2 1 0
```

262 In this example, you ended with a 0, so the parentheses are balanced. In the next
263 example, the parentheses aren't balanced:

Java Example of Unbalanced Parentheses

264 *Read this.*

```
265 if ((a < b) == (c == d)) && !done) ...  
266 | | | | | | | |  
267 0 1 2 1 2 1 0 -1
```

268 The 0 before you get to the last closing parenthesis is a tip-off that a parenthesis
269 is missing before that point. You shouldn't get a 0 until the last parenthesis of the
270 expression.

Fully parenthesize logical expressions

271 Parentheses are cheap, and they aid readability. Fully parenthesizing logical
272 expressions as a matter of habit is good practice.
273

Knowing How Boolean Expressions Are Evaluated

274 Many languages have an implied form of control that comes into play in the
275 evaluation of boolean expressions. Compilers for some languages evaluate each
276 term in a boolean expression before combining the terms and evaluating the
277 whole expression. Compilers for other languages have “short-circuit” or “lazy”
278

279 evaluation, evaluating only the pieces necessary. This is particularly significant
280 when, depending on the results of the first test, you might not want the second
281 test to be executed. For example, suppose you're checking the elements of an
282 array and you have the following test:

283 Pseudocode Example of an Erroneous Test

```
284 while ( i < MAX_ELEMENTS and item[ i ] > 0 ) ...  
285 If this whole expression is evaluated, you'll get an error on the last pass through  
286 the loop. The variable i equals maxElements, so the expression item[ i ] is  
287 equivalent to item[ maxElements ], which is an array-index error. You might  
288 argue that it doesn't matter since you're only looking at the value, not changing  
289 it. But it's sloppy programming practice and could confuse someone reading the  
290 code. In many environments it will also generate either a run-time error or a  
291 protection violation.
```

292 In pseudocode, you could restructure the test so that the error doesn't occur:

293 Pseudocode Example of a Correctly Restructured Test

```
294 while ( i < MAX_ELEMENTS )  
295     if ( item[ i ] > 0 ) then  
296         ...
```

297 This is correct because *item[i]* isn't evaluated unless *i* is less than *maxElements*.

298 Many modern languages provide facilities that prevent this kind of error from
299 happening in the first place. For example, C++ uses short-circuit evaluation: If
300 the first operand of the *and* is false, the second isn't evaluated because the whole
301 expression would be false anyway. In other words, in C++ the only part of

```
302     if ( SomethingFalse && SomeCondition ) ...  
303 that's evaluated is SomethingFalse. Evaluation stops as soon as SomethingFalse  
304 is identified as false.
```

305 Evaluation is similarly short-circuited with the *or* operator. In Java and C++, the
306 only part of

```
307     if ( SomethingTrue || SomeCondition ) ...  
308 that is evaluated is SomethingTrue. The evaluation stops as soon as  
309 SomethingTrue is identified as true. As a result of this method of evaluation, the  
310 following statement is a fine, legal statement.
```

311 Java Example of a Test That Works Because of Short-Circuit Evaluation

```
312 if ( ( denominator != 0 ) && ( ( item / denominator ) > MIN_VALUE ) ) ...  
313 If this full expression were evaluated when denominator equaled 0, the division  
314 in the second operand would produce a divide-by-zero error. But since the
```

315 second part isn't evaluated unless the first part is true, it is never evaluated when
316 *denominator* equals 0, so no divide-by-zero error occurs.

317 On the other hand, since the `&&` (*and*) is evaluated left to right, the following
318 logically equivalent statement doesn't work:

319 Java Example of a Test That Short-Circuit Evaluation Doesn't Rescue

```
320 if ( ( item / denominator ) > MIN_VALUE ) && ( denominator != 0 ) ...
```

321 In this case, *item / denominator* is evaluated before *denominator* `!= 0`.

322 Consequently, this code commits the divide-by-zero error.

323 Java further complicates this picture by providing logical operators and
324 "conditional" operators. Java and C++'s `&&` and `<|$LB><|$LB>` operators
325 function similarly. Java's logical `&` and `<|$LB>` operators do not necessarily
326 short-circuit the evaluation of the right-hand term when the left-hand term
327 determines the truth or falsity of the expression. In other words, in Java, this is
328 safe:

329 Java Example of a Test That Works Because of Short-Circuit 330 (Conditional) Evaluation

```
331 if ( ( denominator != 0 ) && ( ( item / denominator ) > MIN_VALUE ) ) ...
```

332 but this is not:

333 Java Example of a Test That Doesn't Work Because Short-Circuit 334 Evaluation Isn't Guaranteed

```
335 if ( ( denominator != 0 ) & ( ( item / denominator ) > MIN_VALUE ) ) ...
```

336 **KEY POINT**
337 Different languages use different kinds of evaluation, and language
338 implementers tend to take liberties with expression evaluation, so check the
339 manual for the specific version of the language you're using to find out what
340 kind of evaluation your language uses. Better yet, since a reader of your code
341 might not be as sharp as you are, use nested tests to clarify your intentions
instead of depending on evaluation order and short-circuit evaluation.

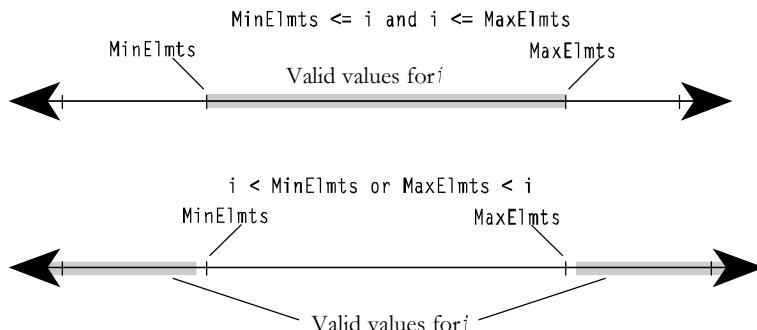
342 Writing Numeric Expressions in Number-Line 343 Order

344 Organize numeric tests so that they follow the points on a number line. In
345 general, structure your numeric tests so that you have comparisons like

```
346 MIN_ELEMENTS <= i and i <= MAX_ELEMENTS  
347 i < MIN_ELEMENTS or MAX_ELEMENTS < i
```

348 The idea is to order the elements left to right, from smallest to largest. In the first
349 line, *MIN_ELEMENTS* and *MAX_ELEMENTS* are the two endpoints, so they go

350 at the ends. The variable i is supposed to be between them, so it goes in the
 351 middle. In the second example, you're testing whether i is outside the range, so i
 352 goes on the outside of the test at either end and MIN_ELEMENTS and
 353 MAX_ELEMENTS go on the inside. This approach maps easily to a visual image
 354 of the comparison:



355

356

357

358

F19xx01**Figure 19-1***Examples of using number-line ordering for boolean tests.*

359

360

361

362

363

364

365

366

367

368

369

370

371

372

373

374

375

376

If you're testing i against MIN_ELEMENTS only, the position of i varies depending on where i is when the test is successful. If i is supposed to be smaller, you'll have a test like

```
while ( i < MIN_ELEMENTS ) ...
```

But if i is supposed to be larger, you'll have a test like

```
while ( MIN_ELEMENTS < i ) ...
```

This approach is clearer than tests like

```
( i > MIN_ELEMENTS ) and ( i < MAX_ELEMENTS )
```

which give the reader no help in visualizing what is being tested.

Guidelines for Comparisons to 0

Programming languages use 0 for several purposes. It's a numeric value. It's a null terminator in a string. It's the lowest address a pointer can have. It's the value of the first item in an enumeration. It's *False* in logical expressions. Because it's used for so many purposes, you should write code that highlights the specific way 0 is used.

Compare logical variables implicitly

As mentioned earlier, it's appropriate to write logical expressions such as

```
while ( !done ) ...
```

377 This implicit comparison to 0 is appropriate because the comparison is in a
378 logical expression.

379 **Compare numbers to 0**

380 Although it's appropriate to compare logical expressions implicitly, you should
381 compare numeric expressions explicitly. For numbers, write

382 `while (balance != 0) ...`

383 rather than

384 `while (balance) ...`

385 **Compare characters to the null terminator (<QS>\0<QS>) explicitly**

386 Characters, like numbers, aren't logical expressions. Thus, for characters, write

387 `while (*charPtr != '\0') ...`

388 rather than

389 `while (*charPtr) ...`

390 This recommendation goes against the common C convention for handling
391 character data (as in the second example), but it reinforces the idea that the
392 expression is working with character data rather than logical data. Some C
393 conventions aren't based on maximizing readability or maintainability, and this
394 is an example of one. Fortunately, this whole issue is fading into the sunset as
395 more code is written using C++ and STL strings and other non-C-null-terminated
396 strings.

397 **Compare pointers to NULL**

398 For pointers, write

399 `while (bufferPtr != NULL) ...`

400 rather than

401 `while (bufferPtr) ...`

402 Like the recommendation for characters, this one goes against the established C
403 convention, but the gain in readability justifies it.

404 **Common Problems with Boolean Expressions**

405 Boolean expressions are subject to a few additional pitfalls that pertain to
406 specific languages.

407 **In C and C++, put constants on the left side of comparisons**

408 C++ poses some special problems with boolean expressions. In C++,
409 interchanging bitwise operators with logical operators is a common gotcha. It's
410 easy to use <LB> instead of <LB><LB> or & instead of &&.

411 If you have problems mistyping = instead of ==, consider the programming
412 convention of putting constants and literals on the left sides of expressions, like
413 this:

**C++ Example of Putting a Constant on the Left Side of an Expression—
An Error that the Compiler Will Catch**

```
414 if ( MIN_ELEMENTS = i ) ...
```

In this expression, the compiler should flag the single = as an error since
assigning anything to a constant is invalid. In contrast, in this expression:

**C++ Example of Putting a Constant on the Right Side of an
Expression—An Error that the Compiler Might not Catch**

```
415 if ( i = MIN_ELEMENTS ) ...
```

the compiler will flag this only as a warning, and only if you have compiler
warnings fully turned on.

416 This recommendation conflicts with the recommendation to use number-line
417 ordering. My personal preference is to use number line ordering and let the
418 compiler warn me about unintended assignments.

***In C++, consider creating preprocessor macro substitutions for &&,
<;\$LB><;\$LB>, and == (but only as a last resort)***

419 If you have such a problem, it's possible to create #define macros for boolean
420 and and or, and use AND and OR instead of && and <;\$LB><;\$LB>. Similarly,
421 using = when you mean == is an easy mistake to make. If you get stung often
422 by this one, you might create a macro like EQUALS for logical equals (==).

423 Many experienced programmers view this approach as aiding readability for the
424 programmer who can't keep details of the programming language straight but
425 degrading readability for the programmer who is more fluent in the language. In
426 addition, most compilers will provide error warnings for usages of assignment
427 and bitwise operators that seem like errors. Turning on full compiler warnings is
428 usually a better option than creating non-standard macros.

In Java, know the difference between a==b and a.equals(b)

429 In Java, *a==b* tests for whether *a* and *b* refer to the same object, whereas
430 *a.equals(b)* tests for whether the objects have the same logical value. In general,
431 Java programs should use expressions like *a.equals(b)* rather than *a==b*.
432

443

19.2 Compound Statements (Blocks)

A “compound statement” or “block” is a collection of statements that are treated as a single statement for purposes of controlling the flow of a program. Compound statements are created by writing { and } around a group of statements in C++, C#, C, and Java. Sometimes they are implied by the keywords of a command, such as *For* and *Next* in Visual Basic. Here are some guidelines for using compound statements effectively:

450 **CROSS-REFERENCE** Many
451 y programmer-oriented text
452 editors have commands that
453 match braces, brackets, and
454 parentheses. For details, see
“Editing” in Section 30.2.

455
456
457
458
459
460
461
462
463
464
465

466
467
468
469
470
471

472
473

Write pairs of braces together

Fill in the middle after you write both the opening and closing parts of a block. People often complain about how hard it is to match pairs of braces or *begin-and-end* pairs, and that’s a completely unnecessary problem. If you follow this guideline, you will never have trouble matching such pairs again.

Write this first:

```
for ( i =0; i < maxLines; i++ )
```

Write this next:

```
for ( i =0; i < maxLines; i++ ) {  
}
```

Write this last:

```
for ( i =0; i < maxLines; i++ ) {  
// whatever goes in here  
...  
}
```

This applies to all blocking structures including *if*, *for* and *while* in C++ and Java and to *If-Then-Else*, *For-Next*, and *While-Wend* combinations in Visual Basic.

Use braces to clarify conditionals

Conditionals are hard enough to read without having to determine which statements go with the *if* test. Putting a single statements after an *if* test is sometimes appealing aesthetically, but under maintenance such statements tend to become more complicated blocks, and single statements are error prone when that happens.

Use blocks to clarify your intentions regardless of whether the code inside the block is 1 line or 20.

474

19.3 Null Statements

475
476

In C++, it’s possible to have a null statement, a statement consisting entirely of a semicolon, as shown here:

C++ Example of a Traditional Null Statement

```
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() )
```

479
480
481
482

483 **CROSS-REFERENCE** The
484 best way to handle null
485 statements is probably to
486 avoid them. For details, see
487 “Avoid empty loops” in
488 Section 16.2.

489
490
491
492
493
494

495 *This is one way to show the*
496 *null statement.*

497 *This is another way to show it.*

498
499500
501502
503
504
505
506
507
508
509
510511
512
513
514
515
516

;

The *while* in C++ requires that a statement follow, but it can be a null statement. The semicolon on a line by itself is a null statement. Here are guidelines for handling null statements in C++:

Call attention to null statements

Null statements are uncommon, so make them obvious. One way is to give the semicolon of a null statement a line of its own. Indent it, just as you would any other statement. This is the approach shown in the previous example.

Alternatively, you can use a set of empty braces to emphasize the null statement. Here are two examples:

C++ Examples of a Null Statement That’s Emphasized

```
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) {};  
  
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) {  
    ;  
}
```

Create a preprocessor null() macro or inline function for null statements

The statement doesn’t do anything but make indisputably clear the fact that nothing is supposed to be done. This is similar to marking blank document pages with the statement “This page intentionally left blank.” The page isn’t really blank, but you know nothing else is supposed to be on it.

Here’s how you can make your own null statement in C++ using *#define*. (You could also create it as an *inline* function, which would have the same effect.)

C++ Example of a Null Statement That’s Emphasized with null()

```
#define null()  
...  
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) {  
    null();  
}
```

In addition to using *null()* in empty *while* and *for* loops, you can use it for unimportant choices of a *switch* statement; including *null()* makes it clear that the case was considered and nothing is supposed to be done.

Note that this *null()* is different from the traditional preprocessor macro *NULL* that’s used for a null pointer. The value of the pointer *NULL* depends on your hardware but is usually *0*, or *OL*, or something like that. It’s never simply empty, as the *null()* here is. If your language doesn’t support preprocessor macros or inline functions, you could create a *null()* routine that simply immediately returns control back to the calling routine.

517
518
519
520

Consider whether the code would be clearer with a non-null loop body

Most of the code that results in loops with empty bodies relies on side effects in the loop control code. In most cases, the code is more readable when the side effects are made explicit, as shown below:

521 **C++ Examples of Rewriting Code to be Clearer with a non-Null Loop**
522 **Body**

523 RecordType record = recordArray.Read(index);
524 index++;
525 while (record != recordArray.EmptyRecord()) {
526 record = recordArray.Read(index);
527 index++
528 };

This approach introduces an additional loop-control variable and requires more lines of code, but it emphasizes straightforward programming practice rather than clever use of side effects, which is preferable in production code.

532

19.4 Taming Dangerously Deep Nesting

533 **HARD DATA**

534
535
536
537
538
539
540

Excessive indentation, or “nesting,” has been pilloried in computing literature for 25 years and is still one of the chief culprits in confusing code. Studies by Noam Chomsky and Gerald Weinberg suggest that few people can understand more than three levels of nested *ifs* (Yourdon 1986a), and many researchers recommend avoiding nesting to more than three or four levels (Myers 1976, Marca 1981, and Ledgard and Tauer 1987a). Deep nesting works against what Chapter 5 describes as Software’s Major Technical Imperative: Managing Complexity. That is reason enough to avoid deep nesting.

541 **KEY POINT**

542
543
544

It’s not hard to avoid deep nesting. If you have deep nesting, you can redesign the tests performed in the *if* and *else* clauses or you can break code into simpler routines. The following sections present several ways to reduce the nesting depth.

545 **CROSS-REFERENCE** Rete
546 sting part of the condition to
547 reduce complexity is similar
548 to retesting a status variable.
That technique is

549 **CODING HORROR**

Section 17.3.

550
551
552
553

Simplify a nested if by retesting part of the condition

If the nesting gets too deep, you can decrease the number of nesting levels by retesting some of the conditions. Here’s a code example with nesting that’s deep enough to warrant restructuring:

C++ Example of Badly, Deeply, Nested Code

```
if ( inputStatus == InputStatus_Success ) {  
    // lots of code  
    ...  
    if ( printerRoutine != NULL ) {
```

```
554     // lots of code  
555     ...  
556     if ( SetupPage() ) {  
557         // lots of code  
558         ...  
559         if ( AllocMem( &printData ) ) {  
560             // lots of code  
561             ...  
562         }  
563     }  
564 }  
565 }
```

This example is contrived to show nesting levels. The *// lots of code* parts are intended to suggest that the routine has enough code to stretch across several screens or across the page boundary of a printed code listing. Here's the code revised to use retesting rather than nesting:

C++ Example of Code Mercifully Unnested by Retesting

```
570 if ( inputStatus == InputStatus_Success ) {  
571     // lots of code  
572     ...  
573     if ( printerRoutine != NULL ) {  
574         // lots of code  
575         ...  
576     }  
577 }  
578  
579 if ( ( inputStatus == InputStatus_Success ) &&  
580     ( printerRoutine != NULL ) && SetupPage() ) {  
581     // lots of code  
582     ...  
583     if ( AllocMem( &printData ) ) {  
584         // lots of code  
585         ...  
586     }  
587 }
```

This is a particularly realistic example because it shows that you can't reduce the nesting level for free; you have to put up with a more complicated test in return for the reduced level of nesting. A reduction from four levels to two is a big improvement in readability, however, and is worth considering.

Simplify a nested if by using a break block

An alternative to the approach described above is to define a section of code that will be executed as a block. If some condition in the middle of the block fails, execution continues at the end of the block.

597

C++ Example of Using a *break* Block

```
598 do {  
599     // begin break block  
600     if ( inputStatus != InputStatus_Success ) {  
601         break; // break out of block  
602     }  
603  
604     // lots of code  
605     ...  
606     if ( printerRoutine == NULL ) {  
607         break; // break out of block  
608     }  
609  
610     // lots of code  
611     ...  
612     if ( !SetupPage() ) {  
613         break; // break out of block  
614     }  
615  
616     // lots of code  
617     ...  
618     if ( !AllocMem( &printData ) ) {  
619         break; // break out of block  
620     }  
621  
622     // lots of code  
623     ...  
624 } while (FALSE); // end break block
```

This technique is uncommon enough that it should be used only when your entire team is familiar with it and when it has been adopted by the team as an accepted coding practice.

628

629 *Convert a nested if to a set of if-then-elses*
630 If you think about a nested *if* test critically, you might discover that you can
631 reorganize it so that it uses *if-then-elses* rather than nested *ifs*. Suppose you have
a bushy decision tree like this:

632

Java Example of an Overgrown Decision Tree

```
633 if ( 10 < quantity ) {  
634     if ( 100 < quantity ) {  
635         if ( 1000 < quantity ) {  
636             discount = 0.10;  
637         }  
638         else {  
639             discount = 0.05;
```

```
640         }
641     }
642     else {
643         discount = 0.025;
644     }
645 }
646 else {
647     discount = 0.0;
648 }
```

This test is poorly organized in several ways, one of which is that the tests are redundant. When you test whether *quantity* is greater than *1000*, you don't also need to test whether it's greater than *100* and greater than *10*. Consequently, you can reorganize the code:

653 Java Example of a Nested *if* Converted to a Set of *if-then-elses*

```
654 if ( 1000 < quantity ) {
655     discount = 0.10;
656 }
657 else if ( 100 < quantity ) {
658     discount = 0.05;
659 }
660 else if ( 10 < quantity ) {
661     discount = 0.025;
662 }
663 else {
664     discount = 0;
665 }
```

This solution is easier than some because the numbers increase neatly. Here's how you could rework the nested *if* if the numbers weren't so tidy:

668 Java Example of a Nested *if* Converted to a Set of *if-then-elses* When 669 the Numbers Are "Messy"

```
670 if ( 1000 < quantity ) {
671     discount = 0.10;
672 }
673 else if ( ( 100 < quantity ) && ( quantity <= 1000 ) ) {
674     discount = 0.05;
675 }
676 else if ( ( 10 < quantity ) && ( quantity <= 100 ) ) {
677     discount = 0.025;
678 }
679 else if ( quantity <= 10 ) {
680     discount = 0;
681 }
```

682
683
684
685
686
The main difference between this code and the previous code is that the
expressions in the *else-if* clauses don't rely on previous tests. This code doesn't
need the *else* clauses to work, and the tests actually could be performed in any
order. The code could consist of four *ifs* and no *elses*. The only reason the *else*
version is preferable is that it avoids repeating tests unnecessarily.

687
Convert a nested if to a case statement
688
You can recode some kinds of tests, particularly those with integers, to use a
689
case statement rather than chains of *ifs* and *elses*. You can't use this technique in
690
some languages, but it's a powerful technique for those in which you can. Here's
691
how to recode the example in Visual Basic:

692 **Visual Basic Example of Converting a Nested if to a case Statement**

```
693     Select Case quantity
694       Case 0 To 10
695         discount = 0.0
696       Case 11 To 100
697         discount = 0.025
698       Case 101 To 1000
699         discount = 0.05
700       Case Else
701         discount = 0.10
702     End Select
```

703 This example reads like a book. When you compare it to the two examples of
704 multiple indentations a few pages earlier, it seems like a particularly clean
705 solution.

706 **Factor deeply nested code into its own routine**
707
708 If deep nesting occurs inside a loop, you can often improve the situation by
709 putting the inside of the loop into its own routine. This is especially effective if
710 the nesting is a result of both conditionals and iterations. Leave the *if-then-else*
711 branches in the main loop to show the decision branching, and then move the
712 statements within the branches to their own routines. Here's an example of code
that needs to be improved by such a modification:

713 **C++ Example of Nested Code That Needs to Be Broken into Routines**

```
714     while ( !TransactionsComplete() ) {
715         // read transaction record
716         transaction = ReadTransaction();

717         // process transaction depending on type of transaction
718         if ( transaction.Type == TransactionType_Deposit ) {
719             // process a deposit
720             if ( transaction.AccountType == AccountType_Checking ) {
```

```
722         if ( transaction.AccountSubType == AccountSubType_Business )
723             MakeBusinessCheckDep( transaction.AccountNum, transaction.Amount );
724         else if ( transaction.AccountSubType == AccountSubType_Personal )
725             MakePersonalCheckDep( transaction.AccountNum, transaction.Amount );
726         else if ( transaction.AccountSubType == AccountSubType_School )
727             MakeSchoolCheckDep( transaction.AccountNum, transaction.Amount );
728     }
729     else if ( transaction.AccountType == AccountType_Savings )
730         MakeSavingsDep( transaction.AccountNum, transaction.Amount );
731     else if ( transaction.AccountType == AccountType_DebitCard )
732         MakeDebitCardDep( transaction.AccountNum, transaction.Amount );
733     else if ( transaction.AccountType == AccountType_MoneyMarket )
734         MakeMoneyMarketDep( transaction.AccountNum, transaction.Amount );
735     else if ( transaction.AccountType == AccountType_Cd )
736         MakeCDDep( transaction.AccountNum, transaction.Amount );
737     }
738     else if ( transaction.Type == TransactionType_Withdrawal ) {
739         // process a withdrawal
740         if ( transaction.AccountType == AccountType_Checking )
741             MakeCheckingWithdrawal( transaction.AccountNum, transaction.Amount );
742         else if ( transaction.AccountType == AccountType_Savings )
743             MakeSavingsWithdrawal( transaction.AccountNum, transaction.Amount );
744         else if ( transaction.AccountType == AccountType_DebitCard )
745             MakeDebitCardWithdrawal( transaction.AccountNum, transaction.Amount );
746     }
747     Here's the
748     TransactionType_Transfer
749     transaction type.
750
751
752
753
754
755
756
757
758
759 }
```

Although it's complicated, this isn't the worst code you'll ever see. It's nested to only four levels, it's commented, it's logically indented, and the functional decomposition is adequate, especially for the *TransactionType_Transfer* transaction type. In spite of its adequacy, however, you can improve it by breaking the contents of the inner *if* tests into their own routines.

765 **CROSS-REFERENCE** This
766 kind of functional
767 decomposition is especially
768 easy if you initially built the
769 routine using the steps
770 described in Chapter 9, “The
771 Pseudocode Programming
772 Process.” Guidelines for
773 functional decomposition are
774 given in “Divide and
775 Conquer” in Section 5.4.

776

777

778

779

780

781

782

783

784

785

786

787

788

789

790

791

792

793

794

795

796

797

798

799

800

801

802

803

804

805

806

807

C++ Example of Good, Nested Code After Decomposition into Routines

```
766 while ( !TransactionsComplete() ) {  
767     // read transaction record  
768     transaction = ReadTransaction();  
769  
770     // process transaction depending on type of transaction  
771     if ( transaction.Type == TransactionType_Deposit ) {  
772         ProcessDeposit(  
773             transaction.AccountType,  
774             transaction.AccountSubType,  
775             transaction.AccountNum,  
776             transaction.Amount  
777         );  
778     }  
779     else if ( transaction.Type == TransactionType_Withdrawal ) {  
780         ProcessWithdrawal(  
781             transaction.AccountType,  
782             transaction.AccountNum,  
783             transaction.Amount  
784         );  
785     }  
786     else if ( transaction.Type == TransactionType_Transfer ) {  
787         MakeFundsTransfer(  
788             transaction.SourceAccountType,  
789             transaction.TargetAccountType,  
790             transaction.AccountNum,  
791             transaction.Amount  
792         );  
793     }  
794     else {  
795         // process unknown transaction type  
796         LogTransactionError("Unknown Transaction Type", transaction );  
797     }  
798 }
```

The code in the new routines has simply been lifted out of the original routine and formed into new routines. (The new routines aren't shown here.) The new code has several advantages. First, two-level nesting makes the structure simpler and easier to understand. Second, you can read, modify, and debug the shorter *while* loop on one screen—it doesn't need to be broken across screen or printed-page boundaries. Third, putting the functionality of *ProcessDeposit()* and *ProcessWithdrawal()* into routines accrues all the other general advantages of modularization. Fourth, it's now easy to see that the code could be broken into a *switch-case* statement, which would make it even easier to read, as shown below:

**808 C++ Example of Good, Nested Code After Decomposition and Use of a
809 switch-case Statement**

```
810       while ( !TransactionsComplete() ) {  
811           // read transaction record  
812           transaction = ReadTransaction();  
813             
814           // process transaction depending on type of transaction  
815           switch ( transaction.Type ) {  
816             case ( TransactionType_Deposit ):  
817               ProcessDeposit(  
818                 transaction.AccountType,  
819                 transaction.AccountSubType,  
820                 transaction.AccountNum,  
821                 transaction.Amount  
822                 );  
823               break;  
824             
825             case ( TransactionType_Withdrawal ):  
826               ProcessWithdrawal(  
827                 transaction.AccountType,  
828                 transaction.AccountNum,  
829                 transaction.Amount  
830                 );  
831               break;  
832             
833             case ( TransactionType_Transfer ):  
834               MakeFundsTransfer(  
835                 transaction.SourceAccountType,  
836                 transaction.TargetAccountType,  
837                 transaction.AccountNum,  
838                 transaction.Amount  
839                 );  
840               break;  
841             
842             default:  
843               // process unknown transaction type  
844               LogTransactionError("Unknown Transaction Type", transaction );  
845               break;  
846       }  
847 }
```

848 Use a more object-oriented approach

849 A straightforward way to simplify this particular code in an object-oriented
850 environment is to create an abstract *Transaction* base class and subclasses for
851 *Deposit*, *Withdrawal*, and *Transfer*.

852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885

C++ Example of Good Code That Uses Polymorphism

```
TransactionData transactionData;
Transaction *transaction;

while ( !TransactionsComplete() ) {
    // read transaction record
    transactionData = ReadTransaction();

    // create transaction object, depending on type of transaction
    switch ( transactionData.Type ) {
        case ( TransactionType_Deposit ):
            transaction = new Deposit( transactionData );
            break;

        case ( TransactionType_Withdrawal ):
            transaction = new Withdrawal( transactionData );
            break;

        case ( TransactionType_Transfer ):
            transaction = new Transfer( transactionData );
            break;

        default:
            // process unknown transaction type
            LogTransactionError("Unknown Transaction Type", transaction );
            break;
    }
    transaction->Complete();
    delete transaction;
}
```

886 **CROSS-REFERENCE** For
887 more beneficial code
888 improvements like this, see
889 Chapter 24, "Refactoring."

890
891
892
893
894
895

In a system of any size, the *switch* statement would be converted to use a factory method that could be reused anywhere an object of *Transaction* type needed to be created. If this code were in such a system, this part of it would become even simpler:

C++ Example of Good Code That Uses Polymorphism and an Object Factory

```
TransactionData transactionData;
Transaction *transaction;

while ( !TransactionsComplete() ) {
    // read transaction record and complete transaction
    transactionData = ReadTransaction();
    transaction = TransactionFactory.Create( transactionData );
    transaction->Complete();
```

```
896     delete transaction;  
897 }  
898
```

For the record, the code in the *TransactionFactory.Create()* routine is a simple adaptation of the code from the prior example's *switch* statement:

C++ Example of Good Code For an Object Factory

```
900  
901 Transaction *TransactionFactory::Create(  
902     TransactionData transactionData  
903 ) {  
904  
905     // create transaction object, depending on type of transaction  
906     switch ( transactionData.Type ) {  
907         case ( TransactionType_Deposit ):  
908             return new Deposit( transactionData );  
909             break;  
910  
911         case ( TransactionType_Withdrawal ):  
912             return new Withdrawal( transactionData );  
913             break;  
914  
915         case ( TransactionType_Transfer ):  
916             return new Transfer( transactionData );  
917             break;  
918  
919         default:  
920             // process unknown transaction type  
921             LogTransactionError( "Unknown Transaction Type", transaction );  
922             return NULL;  
923     }  
924 }
```

Redesign deeply nested code

Some experts argue that *case* statements virtually always indicate poorly factored code in object-oriented programming, and that *case* statements are rarely if ever needed (Meyer 1997). This is one such example.

More generally, complicated code is a sign that you don't understand your program well enough to make it simple. Deep nesting is a warning sign that indicates a need to break out a routine or redesign the part of the code that's complicated. It doesn't mean you have to modify the routine, but you should have a good reason if you don't.

934

935

936

937

938

939

940

941

942

943

944

945

946

947

Summary of Techniques for Reducing Deep Nesting

Here is a summary of the techniques you can use to reduce deep nesting, along with references to the section in this book that discuss the technique:

- Retest part of the condition (this section)
- Convert to *if-then-elses* (this section)
- Convert to a *case* statement (this section)
- Factor deeply nested code into its own routine (this section)
- Use objects and polymorphic dispatch (this section)
- Rewrite the code to use a status variable (in Section 17.3.)
- Use guard clauses to exit a routine and make the nominal path through the code clearer (in Section 17.1.)
- Use exceptions (Section 8.4)
- Redesign deeply nested code entirely (this section)

19.5 A Programming Foundation: Structured Programming

The term “structured programming” originated in a landmark paper, “Structured Programming,” presented by Edsger Dijkstra at the 1969 NATO conference on software engineering (Dijkstra 1969). By the time structured programming came and went, the term “structured” had been applied to every software-development activity, including structured analysis, structured design, and structured goofing off. The various structured methodologies weren’t joined by any common thread except that they were all created at a time when the word “structured” gave them extra cachet.

The core of structured programming is the simple idea that a program should use only one-in, one-out control constructs (also called single-entry, single-exit control constructs). A one-in, one-out control construct is a block of code that has only one place it can start and only one place it can end. It has no other entries or exits. Structured programming isn’t the same as structured, top-down design. It applies only at the detailed coding level.

A structured program progresses in an orderly, disciplined way, rather than jumping around unpredictably. You can read it from top to bottom, and it executes in much the same way. Less disciplined approaches result in source

967 code that provides a less meaningful, less readable picture of how a program
968 executes in the machine. Less readability means less understanding and,
969 ultimately, lower program quality.

970 The central concepts of structured programming are still useful today and apply
971 to considerations in using *break*, *continue*, *throw*, *catch*, *return*, and other topics.

972 The Three Components of Structured 973 Programming

974 The next few sections describe the three constructs that constitute the core of
975 structured programming.

976 Sequence

977 **CROSS-REFERENCE** For
978 details on using sequences,
see Chapter 14, "Organizing
Straight-Line Code."

979

980 // a sequence of assignment statements

981 a = "1";
982 b = "2";
983 c = "3";

984 // a sequence of calls to routines

985 System.out.println(a);
986 System.out.println(b);
987 System.out.println(c);

988

989 **CROSS-REFERENCE** For
990 details on using selections,
see Chapter 15, "Using
991 Conditions."

992

993 A *case* statement is another example of selection control. The *switch* statement
in C++ and Java and the *select* statement in Visual Basic are all examples of
case. In each instance, one of several cases is selected for execution.

994 Conceptually, *if* statements and *case* statements are similar. If your language
995 doesn't support *case* statements, you can emulate them with *if* statements. Here
996 are two examples of selection:

1000 Java Examples of Selection

1001 // selection in an if statement

```
1002  
1003     if ( totalAmount > 0.0 ) {  
1004         // do something  
1005         ...  
1006     }  
1007     else {  
1008         // do something else  
1009         ...  
1010     }  
1011  
1012     // selection in a case statement  
1013     switch ( commandShortcutLetter ) {  
1014         case 'a':  
1015             PrintAnnualReport();  
1016             break;  
1017         case 'q':  
1018             PrintQuarterlyReport();  
1019             break;  
1020         case 's':  
1021             PrintSummaryReport();  
1022             break;  
1023         default:  
1024             DisplayInternalError( "Internal Error 905: Call customer support." );  
1025 }
```

1025

1026 **CROSS-REFERENCE** For
1027 details on using iterations,
1028 see Chapter 16, “Controlling
1029 Loops.”

Iteration

An iteration is a control structure that causes a group of statements to be executed multiple times. An iteration is commonly referred to as a “loop.” Kinds of iterations include *For-Next* in Visual Basic, and *while* and *for* in C++ and Java. The code fragment below shows examples of iteration in Visual Basic:

1030

Visual Basic Examples of Iteration

```
1031     ' example of iteration using a For loop  
1032     For index = first To last  
1033         DoSomething( index )  
1034     Next  
1035  
1036     ' example of iteration using a while loop  
1037     index = first  
1038     While ( index <= last )  
1039         DoSomething ( index )  
1040         index = index + 1  
1041     Wend  
1042  
1043     ' example of iteration using a loop-with-exit loop  
1044     index = first  
1045     Do
```

```
1046 If ( index > last ) Then Exit Do  
1047 DoSomething ( index )  
1048 index = index + 1  
1049 Loop
```

The core thesis of structured programming is that any control flow whatsoever can be created from these three constructs of sequence, selection, and iteration (Böhm Jacopini 1966). Programmers sometimes favor language structures that increase convenience, but programming seems to have advanced largely by restricting what we are allowed to do with our programming languages. Prior to structured programming, use of *gotos* provided the ultimate in control-flow convenience, but code written that way turned out to be incomprehensible and unmaintainable. My belief is that use of any control structure other than the three standard structured programming constructs—that is, the use of *break*, *continue*, *return*, *throw-catch*, and so on—should be viewed with a critical eye.

1060

19.6 Control Structures and Complexity

1061
1062
1063

1064 *Make things as simple as
1065 possible—but no simpler.*
1066 —Albert Einstein

1067
1068
1069
1070

One reason so much attention has been paid to control structures is that they are a big contributor to overall program complexity. Poor use of control structures increases complexity; good use decreases it.

One measure of “programming complexity” is the number of mental objects you have to keep in mind simultaneously in order to understand a program. This mental juggling act is one of the most difficult aspects of programming and is the reason programming requires more concentration than other activities. It’s the reason programmers get upset about “quick interruptions”—such interruptions are tantamount to asking a juggler to keep three balls in the air and hold your groceries at the same time.

1071 **KEY POINT**1072
1073
1074
1075
1076
1077

Intuitively, the complexity of a program would seem to largely determine the amount of effort required to understand it. Tom McCabe published an influential paper arguing that a program’s complexity is defined by its control flow (1976). Other researchers have identified factors other than McCabe’s cyclomatic complexity metric (such as the number of variables used in a routine), but they agree that control flow is at least one of the largest contributors to complexity, if not the largest.

1078

1079 **CROSS-REFERENCE** For
1080 more on complexity, see
1081 "Software's Primary
1082 Technical Imperative:
1083 Managing Complexity" in
1084 Section 5.2.

1085

1086

1087 **HARD DATA**

1088

1089

1090

1091

1092

1093

1094

1095

1096

1097

1098

1099

1100

1101

1102

1103

1104

1105

1106 **FURTHER READING** The
1107 approach described here is
1108 based on Tom McCabe's
1109 influential paper "A
1110 Complexity Measure"
1111 (1976).

1112

How Important Is Complexity?

Computer-science researchers have been aware of the importance of complexity for at least two decades. Many years ago, Edsger Dijkstra cautioned against the hazards of complexity: "The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility" (Dijkstra 1972). This does not imply that you should increase the capacity of your skull to deal with enormous complexity. It implies that you can never deal with enormous complexity and must take steps to reduce it wherever possible.

Control-flow complexity is important because it has been correlated with low reliability and frequent errors (McCabe 1976, Shen et al. 1985). William T. Ward reported a significant gain in software reliability resulting from using McCabe's complexity metric at Hewlett-Packard (1989b). McCabe's metric was used on one 77,000-line program to identify problem areas. The program had a post-release defect rate of 0.31 defects per thousand lines of code. A 125,000-line program had a post-release defect rate of 0.02 defects per thousand lines of code. Ward reported that because of their lower complexity both programs had substantially fewer defects than other programs at Hewlett-Packard. My own company, Construx Software, has experienced similar results using complexity measures to identify problematic routines in the 2000s.

General Guidelines for Reducing Complexity

You can better deal with complexity in one of two ways. First, you can improve your own mental juggling abilities by doing mental exercises. But programming itself is usually enough exercise, and people seem to have trouble juggling more than about five to nine mental entities (Miller 1956). The potential for improvement is small. Second, you can decrease the complexity of your programs and the amount of concentration required to understand them.

How to Measure Complexity

You probably have an intuitive feel for what makes a routine more or less complex. Researchers have tried to formalize their intuitive feelings and have come up with several ways of measuring complexity. Perhaps the most influential of the numeric techniques is Tom McCabe's, in which complexity is measured by counting the number of "decision points" in a routine. Table 19-2 describes a method for counting decision points.

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

-
1. Start with 1 for the straight path through the routine.
 2. Add 1 for each of the following keywords, or their equivalents: *if while repeat for and or*

3. Add 1 for each case in a *case* statement.

1113 Here's an example:

```
1114 if ( (status = Success) and done ) or  
1115     ( not done and ( numLines >= maxLines ) ) then ...
```

1116 In this fragment, you count 1 to start; 2 for the *if*; 3 for the *and*; 4 for the *or*; and
1117 5 for the *and*. Thus, this fragment contains a total of five decision points.

1118 What to Do with Your Complexity Measurement

1119 After you have counted the decision points, you can use the number to analyze
1120 your routine's complexity. If the score is

- 1121 0–5 The routine is probably fine.
- 1122 6–10 Start to think about ways to simplify the routine.
- 1123 10+ Break part of the routine into a second routine and call it from the first
1124 routine.

1125 Moving part of a routine into another routine doesn't reduce the overall
complexity of the program; it just moves the decision points around. But it
reduces the amount of complexity you have to deal with at any one time. Since
the important goal is to minimize the number of items you have to juggle
mentally, reducing the complexity of a given routine is worthwhile.

1126 The maximum of 10 decision points isn't an absolute limit. Use the number of
1127 decision points as a warning flag that indicates a routine might need to be
1128 redesigned. Don't use it as an inflexible rule. A *case* statement with many cases
1129 could be more than 10 elements long, and, depending on the purpose of the *case*
1130 statement, it might be foolish to break it up.

1131 Other Kinds of Complexity

1132 **FURTHER READING** For an
1133 excellent discussion of
1134 complexity metrics, see
1135 *Software Engineering*
1136 *Metrics and Models* (Conte,
Dunsmore, and Shen 1986).

CC2E.COM/1985

CHECKLIST: Control-Structure Issues

- 1141 Do expressions use *True* and *False* rather than *1* and *0*?
- 1142 Are boolean values compared to *True* and *False* implicitly?

- 1143 Are numeric values compared to their test values explicitly?
- 1144 Have expressions been simplified by the addition of new boolean variables
1145 and the use of boolean functions and decision tables?
- 1146 Are boolean expressions stated positively?
- 1147 Do pairs of braces balance?
- 1148 Are braces used everywhere they're needed for clarity?
- 1149 Are logical expressions fully parenthesized?
- 1150 Have tests been written in number-line order?
- 1151 Do Java tests uses *a.equals(b)* style instead of *a == b* when appropriate?
- 1152 Are null statements obvious?
- 1153 Have nested statements been simplified by retesting part of the conditional,
1154 converting to *if-then-else* or *case* statements, moving nested code into its
1155 own routine, converting to a more object-oriented design, or improved in
1156 some other way?
- 1157 If a routine has a decision count of more than 10, is there a good reason for
1158 not redesigning it?

1160 Key Points

- 1161 • Making boolean expressions simple and readable contributes substantially to
1162 the quality of your code.
- 1163 • Deep nesting makes a routine hard to understand. Fortunately, you can avoid
1164 it relatively easily.
- 1165 • Structured programming is a simple idea that is still relevant: you can build
1166 any program out of a combination of sequences, selections, and iterations.
- 1167 • Minimizing complexity is a key to writing high-quality code.

20

The Software-Quality Landscape

Contents

- 20.1 Characteristics of Software Quality
- 20.2 Techniques for Improving Software Quality
- 20.3 Relative Effectiveness of Quality Techniques
- 20.4 When to Do Quality Assurance
- 20.5 The General Principle of Software Quality

Related Topics

- Collaborative construction: Chapter 21
- Developer testing: Chapter 22
- Debugging: Chapter 23
- Prerequisites to construction: Chapters 3 and 4
- Do prerequisites apply to modern software projects? in Section 3.1

THIS CHAPTER SURVEYS SOFTWARE-QUALITY techniques. The whole book is about improving software quality, of course, but this chapter focuses on quality and quality assurance per se. It focuses more on big-picture issues than it does on hands-on techniques. If you're looking for practical advice about collaborative development, testing, and debugging, move on to the next three chapters.

20.1 Characteristics of Software Quality

Software has both external and internal quality characteristics. External characteristics are characteristics that a user of the software product is aware of, including

23 **FURTHER READING** For a classic discussion of quality attributes, see *Characteristics of Software Quality* (Boehm et al. 1978).

- 26 • Correctness. The degree to which a system is free from faults in its
27 specification, design, and implementation.
- 28 • Usability. The ease with which users can learn and use a system.
- 29 • Efficiency. Minimal use of system resources, including memory and
30 execution time.
- 31 • Reliability. The ability of a system to perform its required functions under
32 stated conditions whenever required—having a long mean time between
33 failures.
- 34 • Integrity. The degree to which a system prevents unauthorized or improper
35 access to its programs and its data. The idea of integrity includes restricting
36 unauthorized user accesses as well as ensuring that data is accessed
37 properly—that is, that tables with parallel data are modified in parallel, that
38 date fields contain only valid dates, and so on.
- 39 • Adaptability. The extent to which a system can be used, without
40 modification, in applications or environments other than those for which it
41 was specifically designed.
- 42 • Accuracy. The degree to which a system, as built, is free from error,
43 especially with respect to quantitative outputs. Accuracy differs from
44 correctness; it is a determination of how well a system does the job it's built
45 for rather than whether it was built correctly.
- 46 • Robustness. The degree to which a system continues to function in the
47 presence of invalid inputs or stressful environmental conditions.

48 Some of these characteristics overlap, but all have different shades of meaning
49 that are applicable more in some cases, less in others.

50 External characteristics of quality are the only kind of software characteristics
51 that users care about. Users care about whether the software is easy to use, not
52 about whether it's easy for you to modify. They care about whether the software
53 works correctly, not about whether the code is readable or well structured.

54 Programmers care about the internal characteristics of the software as well as the
55 external ones. This book is code-centered, so it focuses on the internal quality
56 characteristics. They include

- 57 • Maintainability. The ease with which you can modify a software system to
58 change or add capabilities, improve performance, or correct defects.
- 59 • Flexibility. The extent to which you can modify a system for uses or
60 environments other than those for which it was specifically designed.

- 61 ● Portability. The ease with which you can modify a system to operate in an
62 environment different from that for which it was specifically designed.
- 63 ● Reusability. The extent to which and the ease with which you can use parts
64 of a system in other systems.
- 65 ● Readability. The ease with which you can read and understand the source
66 code of a system, especially at the detailed-statement level.
- 67 ● Testability. The degree to which you can unit-test and system-test a system;
68 the degree to which you can verify that the system meets its requirements.
- 69 ● Understandability. The ease with which you can comprehend a system at
70 both the system-organizational and detailed-statement levels.
71 Understandability has to do with the coherence of the system at a more
72 general level than readability does.

73 As in the list of external quality characteristics, some of these internal
74 characteristics overlap, but they too each have different shades of meaning that
75 are valuable.

76 The internal aspects of system quality are the main subject of this book and
77 aren't discussed further in this chapter.

78 The difference between internal and external characteristics isn't completely
79 clear-cut because at some level internal characteristics affect external ones.
80 Software that isn't internally understandable or maintainable impairs your ability
81 to correct defects, which in turn affects the external characteristics of correctness
82 and reliability. Software that isn't flexible can't be enhanced in response to user
83 requests, which in turn affects the external characteristic of usability. The point
84 is that some quality characteristics are emphasized to make life easier for the
85 user and some are emphasized to make life easier for the programmer. Try to
86 know which is which.

87 The attempt to maximize certain characteristics invariably conflicts with the
88 attempt to maximize others. Finding an optimal solution from a set of competing
89 objectives is one activity that makes software development a true engineering
90 discipline. Figure 20-1 shows the way in which focusing on some external
91 quality characteristics affects others. The same kinds of relationships can be
92 found among the internal characteristics of software quality.

93 The most interesting aspect of this chart is that focusing on a specific
94 characteristic doesn't always mean a trade-off with another characteristic.
95 Sometimes one hurts another, sometimes one helps another, and sometimes one
96 neither hurts nor helps another. For example, correctness is the characteristic of
97 functioning exactly to specification. Robustness is the ability to continue
98 functioning even under unanticipated conditions. Focusing on correctness hurts

99 robustness and vice versa. In contrast, focusing on adaptability helps robustness
 100 and vice versa.

101 The chart shows only typical relationships among the quality characteristics. On
 102 any given project, two characteristics might have a relationship that's different
 103 from their typical relationship. It's useful to think about your specific quality
 104 goals and whether each pair of goals is mutually beneficial or antagonistic.

How focusing on the factor below affects the factor to the right	Correctness	Usability	Efficiency	Reliability	Integrity	Adaptability	Accuracy	Robustness
Correctness	↑	↑ ↑				↑	↓	
Usability		↑			↑ ↑			
Efficiency	↓	↑ ↓	↓ ↓	↓ ↓	↓ ↓	↓ ↓	↓ ↓	
Reliability	↑ ↑		↑ ↑	↑ ↑		↑ ↑	↓ ↓	
Integrity		↓ ↑	↑ ↑					
Adaptability				↓	↑		↑ ↑	
Accuracy	↑	↓ ↑	↑		↓	↑ ↑	↓ ↓	
Robustness	↓	↑ ↓	↓ ↓	↓ ↓	↑ ↑	↓ ↓	↑ ↑	

Helps it ↑
Hurts it ↓

F20xx01

Figure 20-1

Focus on one external characteristic of software quality can affect other characteristics positively, adversely, or not at all.

20.2 Techniques for Improving Software Quality

Software quality assurance is a planned and systematic program of activities designed to ensure that a system has the desired characteristics. Although it might seem that the best way to develop a high-quality product would be to focus on the product itself, in software quality assurance the best place to focus is on the process. Here are some of the elements of a software-quality program:

Software-quality objectives

One powerful technique for improving software quality is setting explicit quality objectives from among the external and internal characteristics described in the last section. Without explicit goals, programmers can work to maximize

121 characteristics different from the ones you expect them to maximize. The power
122 of setting explicit goals is discussed in more detail later in this section.

123 ***Explicit quality-assurance activity***
124 One common problem in assuring quality is that quality is perceived as a
125 secondary goal. Indeed, in some organizations, quick and dirty programming is
126 the rule rather than the exception. Programmers like Gary Goto, who litter their
127 code with defects and “complete” their programs quickly, are rewarded more
128 than programmers like High-Quality Henry, who write excellent programs and
129 make sure that they are usable before releasing them. In such organizations, it
130 shouldn’t be surprising that programmers don’t make quality their first priority.
131 The organization must show programmers that quality is a priority. Making the
132 quality-assurance activity independent makes the priority clear, and
133 programmers will respond accordingly.

134 **CROSS-REFERENCE** For
135 details on testing, see Chapter
136 22, “Developer Testing.”

137
138
139
140
141

142 **CROSS-REFERENCE** For
143 a discussion of one class of
144 software-engineering
145 guidelines appropriate for
146 construction, see Section 4.2,
147 “Programming Conventions.”
148

149
150
151
152

153 **CROSS-REFERENCE** Revi
154 ews and inspections are
155 discussed in Chapter 21,
156 “Collaborative Construction.”
157
158
159

160 architecture, architecture and detailed design and construction, and construction
161 and system testing. The “gate” can be a peer review, a customer review, an
162 inspection, a walkthrough, or an audit.

163 A “gate” does not mean that architecture or requirements need to be 100 percent
164 complete or frozen; it does mean that you will use the gate to determine whether
165 the requirements or architecture are good enough to support downstream
166 development. “Good enough” might mean that you’ve sketched out the most
167 critical 20 percent of the requirements or architecture, or it might mean you’ve
168 specified 95 percent in excruciating detail—which end of the scale you should
169 aim for depends on the nature of your specific project.

170 **External audits**
171 An external audit is a specific kind of technical review used to determine the
172 status of a project or the quality of a product being developed. An audit team is
173 brought in from outside the organization and reports its findings to whoever
174 commissioned the audit, usually management.

175 **FURTHER READING** For a
176 discussion of software
177 development as a process, see
178 *Professional Software
179 Development* (McConnell
1994).
180

181 **CROSS-REFERENCE** For
182 details on change control, see
183 Section 28.2, “Configuration
184 Management.”
185
186
187
188
189
190
191

192 **Change-control procedures**
193 One big obstacle to achieving software quality is uncontrolled changes.
194 Uncontrolled requirements changes can result in disruption to design and coding.
195 Uncontrolled changes in architecture or design can result in code that doesn’t
196 agree with its design, inconsistencies in the code, or the use of more time in
modifying code to meet the changing design than in moving the project forward.
Uncontrolled changes in the code itself can result in internal inconsistencies and
uncertainties about which code has been fully reviewed and tested and which
hasn’t. Uncontrolled changes in requirements, architecture, design, or code can
have all of these effects. Consequently, handling changes effectively is a key to
effective product development.

192 **Measurement of results**
193 Unless results of a quality-assurance plan are measured, you’ll have no way of
194 knowing whether the plan is working. Measurement tells you whether your plan
195 is a success or a failure and also allows you to vary your process in a controlled
way to see whether it can be improved.

197 **What gets measured, gets**

198 **done.**

199 —**Tom Peters**

200

201 **HARD DATA**

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

Measurement has a second, motivational, effect. People pay attention to whatever is measured, assuming that it's used to evaluate them. Choose what you measure carefully. People tend to focus on work that's measured and to ignore work that isn't.

Prototyping

Prototyping is the development of realistic models of a system's key functions. A developer can prototype parts of a user interface to determine usability, critical calculations to determine execution time, or typical data sets to determine memory requirements. A survey of 16 published and 8 unpublished case studies compared prototyping to traditional, specification-development methods. The comparison revealed that prototyping can lead to better designs, better matches with user needs, and improved maintainability (Gordon and Bieman 1991).

Setting Objectives

Explicitly setting quality objectives is a simple, obvious step in achieving quality software, but it's easy to overlook. You might wonder whether, if you set explicit quality objectives, programmers will actually work to achieve them? The answer is, yes, they will, if they know what the objectives are and the objectives are reasonable. Programmers can't respond to a set of objectives that change daily or that are impossible to meet.

Gerald Weinberg and Edward Schulman conducted a fascinating experiment to investigate the effect on programmer performance of setting quality objectives (1974). They had five teams of programmers work on five versions of the same program. The same five quality objectives were given to each of the five teams, and each team was told to maximize a different objective. One team was told to minimize the memory required, another was told to produce the clearest possible output, another was told to build the most readable code, another was told to use the minimum number of statements, and the last group was told to complete the program in the least amount of time possible. Here is how each team was ranked according to each objective:

Team Ranking on Each Objective

Objective Team Was Told to Optimize	Minimum memory use	Most readable output	Most readable code	Least code	Minimum programming time
Minimum memory	1	4	4	2	5
Output readability	5	1	1	5	3
Program readability	3	2	2	3	4
Minimum statements	2	5	3	1	3

226
227228 **HARD DATA**229
230231
232
233
234
235236
237238
239
240
241

242

243 *If builders built buildings
244 the way programmers
245 wrote programs, then the
246 first woodpecker that
247 came along would destroy
civilization.*

248 —Gerald Weinberg

Team Ranking on Each Objective

Objective Team Was Told to Optimize	Minimum memory use	Most readable output	Most readable code	Least code	Minimum programming time
Minimum programming time	4	3	5	4	1

Source: Adapted from “Goals and Performance in Computer Programming” (Weinberg and Schulman 1974).

The results of this study were remarkable. Four of the five teams finished first in the objective they were told to optimize. The other team finished second in its objective. None of the teams did consistently well in all objectives.

The surprising implication is that people actually do what you ask them to do. Programmers have high achievement motivation: They will work to the objectives specified, but they must be told what the objectives are. The second implication is that, as expected, objectives conflict and it’s generally not possible to do well on all of them.

20.3 Relative Effectiveness of Quality Techniques

The various quality-assurance practices don’t all have the same effectiveness. Many techniques have been studied, and their effectiveness at detecting and removing defects is known. This and several other aspects of the “effectiveness” of the quality-assurance practices are discussed in this section.

Percentage of Defects Detected

Some practices are better at detecting defects than others, and different methods find different kinds of defects. One way to evaluate defect-detection methods is to determine the percentage of defects they find out of the total defects found over the life of a product. Table 20-1 shows the percentages of defects detected by several common defect-detection techniques.

Table 20-1. Defect-Detection Rates

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

249

250

251

252 HARD DATA

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268 HARD DATA

269

270

271

272

Source: Adapted from Programming Productivity (Jones 1986a), “Software Defect Removal Efficiency” (Jones 1996), and “What We Have Learned About Fighting Defects” (Shull et al 2002).

The most interesting fact that this data reveals is that the modal rates don't rise above 75 percent for any single technique, and the techniques average about 40 percent. Moreover, for the most common kind of defect detection, unit testing, the modal rate is only 30 percent. The typical organization uses a test-heavy defect-removal approach, and achieves only about 85% defect removal efficiency. Leading organizations use a wider variety of techniques and achieve defect removal efficiencies of 95 percent or higher (Jones 2000).

The strong implication is that if project developers are striving for a higher defect-detection rate, they need to use a combination of techniques. A classic study by Glenford Myers confirmed this implication (Myers 1978b). Myers studied a group of programmers with a minimum of 7 and an average of 11 years of professional experience. Using a program with 15 known errors, he had each programmer look for errors using one of these techniques:

- Execution testing against the specification
- Execution testing against the specification with the source code
- Walkthrough/inspection using the specification and the source code

Myers found a huge variation in the number of defects detected in the program, ranging from 1.0 to 9.0 defects found. The average number found was 5.1, or about a third of those known.

When used individually, no method had a statistically significant advantage over any of the others. The variety of errors people found was so great, however, that

any combination of two methods (including having two independent groups using the same method) increased the total number of defects found by a factor of almost 2. A study at NASA's Software Engineering Laboratory also reported that different people tend to find different defects. Only 29 percent of the errors found by code reading were found by both of two code readers (Kouchakdjian, Green, and Basili 1989).

Glenford Myers points out that human processes (inspections and walkthroughs, for instance) tend to be better than computer-based testing at finding certain kinds of errors and that the opposite is true for other kinds of errors (1979). This result was confirmed in a later study, which found that code reading detected more interface defects and functional testing detected more control defects (Basili, Selby, and Hutchens 1986). Test guru Boris Beizer reports that informal test approaches typically achieve only 50-60% test coverage unless you're using a coverage analyzer (Johnson 1994).

KEY POINT

The upshot is that defect-detection methods work better in combination than they do singly. Jones made the same point when he observed that cumulative defect-detection efficiency is significantly higher than that of any individual technique. The outlook for the effectiveness of testing used by itself is bleak. Jones points out that a combination of unit testing, functional testing, and system testing often results in a cumulative defect detection of less than 60 percent, which is usually inadequate for production software.

This data can also be used to understand why programmers who begin working with a disciplined defect removal technique such as Extreme Programming experience higher defect removal levels than they have experienced previously. As Table 20-2 illustrates, the set of defect removal practices used in Extreme Programming would be expected to achieve about 90% defect removal efficiency in the average case and 97% in the best case, which is far better than the industry average of 85% defect removal. This result is not due to any mysterious "synergy" among extreme programming's practices; it is a predictable outcome of using these specific defect removal practices. Other combinations of practices can work equally well or better, and the determination of which specific defect removal practices will be used to achieve the desired quality level is one part of effective project planning.

Table 20-2. Extreme Programming's Estimated Defect-Detection Rate

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews (pair programming)	25%	35%	40%
Informal code reviews (pair programming)	20%	25%	35%

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
Expected cumulative defect removal efficiency	~74%	~90%	~97%

307

308 Some defect-detection practices cost more than others. The most economical
309 practices result in the least cost per defect found, all other things being equal.
310 The qualification that all other things must be equal is important because per
311 defect cost is influenced by the total number of defects found, the stage at which
312 each defect is found, and other factors besides the economics of a specific
313 defect-detection technique.

314 In the 1978 Myers study cited earlier, the difference in cost per defect between
315 the two execution-testing methods (with and without source code) wasn't
316 statistically significant, but the walkthrough/inspection method cost over twice
317 as much per defect found as the test methods (Myers 1978). These results have
318 been consistent for decades. A later study at IBM found that only 3.5 staff hours
319 were needed to find each error using code inspections, whereas 15-25 hours were
320 needed to find each error through testing (Kaplan 1995).

321 **HARD DATA**

322 Organizations tend to become more effective at doing inspections as they gain
323 experience. Consequently, more recent studies have shown conclusively that
324 inspections are cheaper than testing. One study of three releases of a system
325 showed that on the first release, inspections found only 15 percent of the errors
326 found with all techniques. On the second release, inspections found 41 percent,
327 and on the third, 61 percent (Humphrey 1989). If this history were applied to
328 Myers's study, it might turn out that inspections would eventually cost half as
329 much per defect as testing instead of twice as much. A study at the Software
330 Engineering Laboratory found that code reading detected about 80 percent more
faults per hour than testing (Basili and Selby 1987).

331 Cost of Fixing Defects

332 **CROSS-REFERENCE** For
333 details on the fact that defects
334 become more expensive the
longer they stay in a system,

335 **HARD DATA**
Section 5.1, "For an up-close
336 look at errors themselves, see
337 Section 22.4, "Typical
338 Errors."

339
340
341
342
343
344
345
346
347

348
349
350

351
352
353
354

355

356 **CROSS-REFERENCE** Qual-
357 ity assurance of upstream
358 activities—requirements and
359 architecture, for instance—is
360 outside the scope of this
book. The "Additional
361 Resources" section at the end
362 of the chapter describes
363 books you can turn to for
more information about them.
364

The cost of finding defects is only one part of the cost equation. The other is the cost of fixing defects. It might seem at first glance that how the defect is found wouldn't matter—it would always cost the same amount to fix.

That isn't true because the longer a defect remains in the system, the more expensive it becomes to remove. A detection technique that finds the error earlier therefore results in a lower cost of fixing it. Even more important, some techniques, such as inspections, detect the symptoms and causes of defects in one step; others, such as testing, find symptoms but require additional work to diagnose and fix the root cause. The result is that one-step techniques are substantially cheaper overall than two-step ones. Microsoft's applications division has found that it takes 3 hours to find and fix a defect using code inspection, a one-step technique, and 12 hours to find and fix a defect using testing, a two-step technique (Moore 1992). Collofello and Woodfield reported on a 700,000-line program built by over 400 developers (1989). They found that code reviews were several times as cost-effective as testing—1.38 return on investment vs. 0.17.

The bottom line is that an effective software-quality program must include a combination of techniques that apply to all stages of development. Here's a recommended combination:

- Formal design inspections of the critical parts of a system
- Modeling or prototyping using a rapid prototyping technique
- Code reading or inspections
- Execution testing

355 20.4 When to Do Quality Assurance

As Chapter 3 noted, the earlier an error is inserted into software, the more embedded it becomes in other parts of the software and the more expensive it becomes to remove. A fault in requirements can produce one or more corresponding faults in design, which can produce many corresponding faults in code. A requirements error can result in extra architecture or in bad architectural decisions. The extra architecture results in extra code, test cases, and documentation. Just as it's a good idea to work out the defects in the blueprints for a house before pouring the foundation in concrete, it's a good idea to catch requirements and architecture errors before they affect later activities.

365
366
367
368
369

In addition, errors in requirements or architecture tend to be more sweeping than construction errors. A single architectural error can affect several classes and dozens of routines, whereas a single construction error is unlikely to affect more than one routine or class. For this reason, too, it's cost-effective to catch errors as early as you can.

370 **KEY POINT**

371
372
373
374

Defects creep into software at all stages. Consequently, you should emphasize quality-assurance work in the early stages and throughout the rest of the project. It should be planned into the project as work begins; it should be part of the technical fiber of the project as work continues; and it should punctuate the end of the project, verifying the quality of the product as work ends.

375
376

20.5 The General Principle of Software Quality

377 **KEY POINT**

378
379
380
381

There's no such thing as a free lunch, and even if there were, there's no guarantee that it would be any good. Software development is a far cry from *haute cuisine*, however, and software quality is unusual in a significant way. The General Principle of Software Quality is that improving quality reduces development costs.

382
383
384
385
386
387
388

Understanding this principle depends on understanding a key observation: The best way to improve productivity and quality is to reduce the time spent reworking code, whether the rework is from changes in requirements, changes in design, or debugging. The industry-average productivity for a software product is about 10 to 50 of lines of delivered code per person per day (including all non-coding overhead). It takes only a matter of minutes to type in 10 to 50 lines of code, so how is the rest of the day spent?

389 **CROSS-REFERENCE** For details on the difference between writing an individual program and writing a software product, see "Programs, Products, Systems, and System Products" in Section 27.5.

395
396
397
398
399

Part of the reason for these seemingly low productivity figures is that industry average numbers like these factor non-programmer time into the lines-of-code-per-day figure. Tester time, project manager time, and administrative support time are all included. Non-coding activities like requirements development and architecture work are also typically factored into those lines-of-code-per-day figures. But none of that is what takes up so much time.

The single biggest activity on most projects is debugging and correcting code that doesn't work properly. Debugging and associated rework consume about 50 percent of the time on a traditional, naive software-development cycle. (See Section 3.1 for more details.) Reducing debugging by preventing errors improves productivity. Therefore, the most obvious method of shortening a development

400
401

schedule is to improve the quality of the product and decrease the amount of time spent debugging and reworking the software.

402 **HARD DATA**

403
404
405
406

This analysis is confirmed by field data. In a review of 50 development projects involving over 400 work-years of effort and almost 3 million lines of code, a study at NASA's Software Engineering Laboratory found that increased quality assurance was associated with decreased error rate but no increase or decrease in overall development cost (Card 1987).

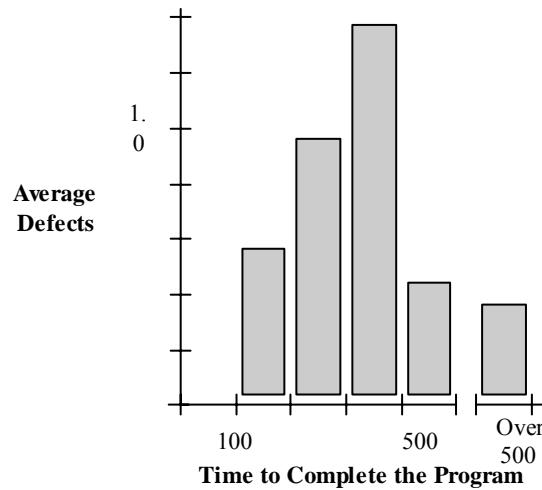
407 A study at IBM produced similar findings:

408 *Software projects with the lowest levels of defects had
409 the shortest development schedules and the highest
410 development productivity. ...Software defect removal is
411 actually the most expensive and time-consuming form of work
412 for software (Jones 2000).*

413 **HARD DATA**

414
415
416
417
418
419
420

The same effect holds true on a smaller scale. In a 1985 study, 166 professional programmers wrote programs from the same specification. The resulting programs averaged 220 lines of code and a little under five hours to write. The fascinating result was that programmers who took the median time to complete their programs produced programs with the greatest number of errors. The programmers who took more or less than the median time produced programs with significantly fewer errors (DeMarco and Lister 1985). Figure 20-2 graphs the results:



F20xx02

Figure 20-2

Neither the fastest nor the slowest development approach produces the software with the most defects.

The two slowest groups took about five times as long to achieve roughly the same defect rate as the fastest group. It's not necessarily the case that writing software without defects takes more time than writing software with defects. As the graph shows, it can take less.

Admittedly, on certain kinds of projects, quality assurance costs money. If you're writing code for the space shuttle or for a medical life-support system, the degree of reliability required makes the project more expensive.

People have argued for decades that fix-defects-early analysis doesn't apply to them. In the 1980s, people argued that such analysis didn't apply to them any more because structured programming was so much faster than traditional programming. In the 1990s, people argued that it didn't apply to them because object-oriented programming was so much faster than traditional techniques. In the 2000s, people assert that the argument doesn't apply to them because agile practices are so much better than traditional techniques. The pattern in these statements across the decades obvious, and, as Section 3.1 described in detail, the available data says that late corrections and late changes cost more than early corrections and changes when agile practices are used just as they did when object-oriented practices, structured practices, and machine-language practices were used.

Compared to the traditional code-test-debug cycle, an enlightened software-quality program saves money. It redistributes resources away from debugging

421

422

423

424

425

426

427

428

429

430

431

432

433

434

435

436

437

438

439

440

441

442

443

444

447 and into upstream quality-assurance activities. Upstream activities have more
448 leverage on product quality than downstream activities, so the time you invest
449 upstream saves more time downstream. The net effect is fewer defects, shorter
450 development time, and lower costs. You'll see several more examples of the
451 General Principle of Software Quality in the next three chapters.

CC2E.COM/2043

CHECKLIST: A Quality-Assurance Plan

- 453 Have you identified specific quality characteristics that are important to your
454 project?
- 455 Have you made others aware of the project's quality objectives?
- 456 Have you differentiated between external and internal quality
457 characteristics?
- 458 Have you thought about the ways in which some characteristics may
459 compete with or complement others?
- 460 Does your project call for the use of several different error-detection
461 techniques suited to finding several different kinds of errors?
- 462 Does your project include a plan to take steps to assure software quality
463 during each stage of software development?
- 464 Is the quality measured in some way so that you can tell whether it's
465 improving or degrading?
- 466 Does management understand that quality assurance incurs additional costs
467 up front in order to save costs later?
-

CC2E.COM/2050

Additional Resources

470 It's not hard to list books in this section because virtually any book on effective
471 software methodologies describes techniques that result in improved quality and
472 productivity. The difficulty is finding books that deal with software quality per
473 se. Here are two:

474 Ginac, Frank P.. *Customer Oriented Software Quality Assurance*, Englewood
475 Cliffs, N.J.: Prentice Hall, 1998. This is a very short book that describes quality
476 attributes, quality metrics, QA programs, and the role of testing in quality as well
477 as well-known quality improvement programs including the Software
478 Engineering Institute's CMM and ISO 9000.

479 Lewis, William E. *Software Testing and Continuous Quality Improvement, 2d.*
480 Ed., Auerbach Publishing, 2000. This book provides a comprehensive discussion

481 of a quality lifecycle, as well as extensive discussion of testing techniques. It
482 also provides numerous forms and checklists.

483 Howard, Michael, and David LeBlanc. *Writing Secure Code*, 2d Ed., Redmond,
484 WA: Microsoft Press, 2003. Software security has become one of the significant
485 technical challenges in modern computing. This book provides easy-to-read
486 practical advice for creating secure software. Although the title suggests that the
487 book focuses solely on code, the book is more comprehensive, spanning a full
488 range of requirements, design, code, and test issues.

CC2E.COM/2057

489 Relevant Standards

490 *IEEE Std 730-2002: IEEE Standard for Software Quality Assurance Plans.*

491 *IEEE Std 1061-1998: IEEE Standard for a Software Quality Metrics*
492 *Methodology.*

493 *IEEE Std 1028-1997, Standard for Software Reviews*

494 *IEEE Std 1008-1987 (R1993), Standard for Software Unit Testing*

495 *IEEE Std 829-1998, Standard for Software Test Documentation*

496 Key Points

- 497 • Quality is free, in the end, but it requires a reallocation of resources so that
498 defects are prevented cheaply instead of fixed expensively.
- 499 • Not all quality-assurance goals are simultaneously achievable. Explicitly
500 decide which goals you want to achieve, and communicate the goals to other
501 people on your team.
- 502 • No single defect-detection technique is effective by itself. Testing by itself is
503 not effective at removing errors. Successful quality-assurance programs use
504 several different techniques to detect different kinds of errors.
- 505 • You can apply effective techniques during construction and many equally
506 powerful techniques before construction. The earlier you find a defect, the
507 less damage it will cause.
- 508 • Quality assurance in the software arena is process-oriented. Software
509 development doesn't have a repetitive phase that affects the final product
510 like manufacturing does, so the quality of the result is controlled by the
511 process used to develop the software.

21

Collaborative Construction

CC2E.COM/2185

Contents

- 21.1 Overview of Collaborative Development Practices
- 21.2 Pair Programming
- 21.3 Formal Inspections
- 21.4 Other Kinds of Collaborative Development Practices

Related Topics

The software-quality landscape: Chapter 20

Developer testing: Chapter 22

Debugging: Chapter 23

Prerequisites to construction: Chapters 3 and 4

YOU MIGHT HAVE HAD AN EXPERIENCE common to many programmers: You walk into another programmer's cubicle and say, "Would you mind looking at this code? I'm having some trouble with it." You start to explain the problem. "It can't be a result of this thing, because I did that. And it can't be the result of this other thing, because I did this. And it can't be the result of—wait a minute. It *could* be the result of that. Thanks!" You've solved your problem before your "helper" has had a chance to say a word.

In one way or another, all collaborative construction techniques are attempts to formalize the process of showing your work to someone else for the purpose of flushing out errors.

If you've read about inspections and pair programming before, you won't find much new information in this chapter. The extent of the hard data about the effectiveness of inspections in Section 21.3 might surprise you, and you might not have considered the code-reading alternative described in Section 21.4. You might also take a look at Table 21-1, "Comparison of Collaborative Construction Techniques," at the end of the chapter. If your knowledge is all from your own experience, read on! Other people have had different experiences, and you'll find some new ideas.

31 **21.1 Overview of Collaborative Development**

32 **Practices**

33 “Collaborative construction” refers to pair programming, formal inspections,
34 informal technical reviews, and document reading, as well as other techniques in
35 which developers share responsibility for creating code and other workproducts.
36 At my company, the term “collaborative construction” was coined by Matt
37 Peloquin about 2000. The term appears to have been coined independently at
38 other companies in the same timeframe.

39 All collaborative construction techniques, despite their differences, are based on
40 the idea that developers are blind to some of the trouble spots in their work, that
41 other people don’t have the same blind spots, and that it’s beneficial to have
42 someone else look at their work.

43 **Collaborative Construction Complements Other**

44 **Quality-Assurance Techniques**

45 KEY POINT
46 HARD DATA

47 The primary purpose of collaborative construction is to improve software
48 quality. As noted in Chapter 22, software testing has limited effectiveness when
49 used alone—the average defect-detection rate is only about 30 percent for unit
50 testing, 35 percent for integration testing, and 35% for low-volume beta testing.
51 In contrast, the average effectivenesses of design and code inspections are 55 and
52 60 percent (Jones 1996). The auxiliary benefit of collaborative construction is
53 that it decreases development time, which in turn lowers development costs.

54 Early reports on pair programming suggest that it can achieve a code-quality
55 level similar to formal inspections (Shull et al 2002). The cost of full-up pair
56 programming is probably higher than the cost of solo development—on the order
57 of 10-25% higher—but the reduction in development time appears to be on the
58 order of 45%, which in some cases may be a decisive advantage over solo
59 development (Boehm and Turner 2004), although not over inspections which
60 have produced similar results.

61 HARD DATA

62 Technical reviews have been studied much longer than pair programming, and
63 case studies of their results have been impressive:

- 64 • IBM found that each hour of inspection prevented about 100 hours of related
65 work (testing and defect correction) (Holland 1999).
- 66 • Raytheon reduced its cost of defect correction (rework) from about 40% of
67 total project cost to about 20% through an initiative that focused on
68 inspections (Haley 1996).

- 66 ● Hewlett-Packard reported that its inspection program saved an estimated
67 \$21.5 million per year (Grady and Van Slack 1994).
- 68 ● Imperial Chemical Industries found that the cost of maintaining a portfolio
69 of about 400 programs was only about 10% as high as the cost of
70 maintaining a similar set of programs that had not been inspected (Gilb and
71 Graham 1993).
- 72 ● A study of large programs found that each hour spent on inspections avoided
73 an average of 33 hours of maintenance work, and inspections were up to 20
74 times more efficient than testing (Russell 1991).
- 75 ● In a software-maintenance organization, 55 percent of one-line maintenance
76 changes were in error before code reviews were introduced. After reviews
77 were introduced, only 2 percent of the changes were in error (Freedman and
78 Weinberg 1990). When all changes were considered, 95 percent were correct
79 the first time after reviews were introduced. Before reviews were introduced,
80 under 20 percent were correct the first time.
- 81 ● A group of 11 programs were developed by the same group of people and all
82 were released to production. The first 5 were developed without reviews
83 and averaged 4.5 errors per 100 lines of code. The other 6 were inspected
84 and averaged only 0.82 errors per 100 lines of code. Reviews cut the errors
85 by over 80 percent (Freedman and Weinberg 1990).

86 Capers Jones reports that all of the software projects he has studied that have
87 achieved 99 percent defect removal rates or better have used formal inspections;
88 none of the projects that achieved less than 75 percent defect removal efficiency
89 used formal inspections (Jones 2000).

90 These results dramatically illustrate the General Principle of Software Quality,
91 which holds that reducing the number of defects in the software also improves
92 development time.

93 **KEY POINT**
94 Various studies have shown that in addition to being more effective at catching
95 errors than testing, collaborative practices find different kinds of errors than
96 testing does (Myers 1978; Basili, Selby, and Hutchens 1986). As Karl Wiegers
97 points out, “A human reviewer can spot unclear error messages, inadequate
98 comments, hard-coded variable values, and repeated code patterns that should be
99 consolidated. Testing won’t” (Wiegers 2002). A secondary effect is that when
100 people know their work will be reviewed, they scrutinize it more carefully. Thus,
101 even when testing is done effectively, reviews or other kinds of collaboration are
needed as part of a comprehensive quality program.

102

103

104 *Informal review*
105 *procedures were passed*
106 *on from person to person*
107 *in the general culture of*
108 *computing for many*
109 *years before they were*
110 *acknowledged in print.*
111 *The need for reviewing*
112 *was so obvious to the best*
113 *programmers that they*
114 *rarely mentioned it in*
115 *print, while the worst*
116 *programmers believed*
117 *they were so good that*
118 *their work did not need*
reviewing.

119 —Daniel Freedman and
120 Gerald Weinberg

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

Collaborative Construction Provides Mentoring in Corporate Culture and Programming Expertise

Software standards can be written down and distributed, but if no one talks about them or encourages others to use them, they won't be followed. Reviews are an important mechanism for giving programmers feedback about their code. The code, the standards, and the reasons for making the code meet the standards are good topics for review discussions.

In addition to feedback about how well they follow standards, programmers need feedback about more subjective aspects of programming—formatting, comments, variable names, local and global variable use, design approaches, the-way-we-do-things-around-here, and so on. Programmers who are still wet behind the ears need guidance from those who are more knowledgeable. More knowledgeable programmers tend to be busy and need to be encouraged to spend time sharing what they know. Reviews create a venue for more experienced and less experienced programmers to communicate about technical issues. As such, reviews are an opportunity for cultivating quality improvements in the future as much as in the present.

One team that used formal inspections reported that inspections quickly brought all the developers up to the level of the best developers (Tackett and Van Doren 1999).

Collective Ownership Applies to All Forms of Collaborative Construction

A concept that spans all collaborative construction techniques is the idea of collective ownership. In some development models, programmers own the code they write, and there are official or unofficial restrictions on modifying someone else's code.

With collective ownership, all code is owned by the group rather than by individuals and can be modified by various members of the group. This produces several valuable benefits:

- Better code quality arises from multiple sets of eyes seeing the code and multiple programmers working on the code
- The risk of someone leaving the project is lower because multiple people are familiar with each section of code
- Defect-correction cycles are shorter overall because any of several programmers can potentially be assigned to fix bugs on an as-available basis

137 Some methodologies like Extreme Programming recommend formally pairing
138 programmers and rotating their work assignments over time. At my company,
139 we've found that programmers don't need to pair up formally to achieve good
140 code coverage; over time, we achieve cross-coverage through a combination of
141 formal and informal technical reviews, pair programming when needed, and
142 rotating defect-correction assignments.

143 **Collaboration Applies As Much Before 144 Construction As After**

145 This book is about construction, so reviews of detailed design and code are the
146 focus of this chapter. However, most of the comments about reviews in this
147 chapter also apply to estimates, plans, requirements, architecture, and
148 maintenance work. By reading between the lines and studying the references at
149 the end of the chapter, you can apply reviews to any stage of software
150 development.

151 **21.2 Pair Programming**

152 Pair programming one programmer types in code at the keyboard, and another
153 programmer watches for mistakes and thinks strategically about whether the
154 code is being written right and whether the right code is being written. Pair
155 programming was originally associated with Extreme Programming (Beck
156 2000), but it is now being used more widely (Williams and Kessler 2002).

157 **Keys to Success with Pair Programming**

158 The basic concept of pair programming is simple, but it nonetheless benefits
159 from a few guidelines.

160 *Support pair programming with coding standards*

161 Pair programming will not be effective if the two people in the pair spend their
162 time arguing about coding style. Try to standardize what Chapter 5 refers to as
163 the "accidental attributes" of programming so that the programmers can focus on
164 the "essential" task at hand.

165 *Don't let pair programming turn into watching*

166 The person without the keyboard should be an active participant in the
167 programming. That person is analyzing the code, thinking ahead to what will be
168 coded next, evaluating the design, and planning how to test the code.

169 *Don't force pair programming of the easy stuff*

170 One group that used pair programming for the most complicated code found it
171 more expedient to do detailed design at the whiteboard for 15 minutes and then
172 program solo (Manzo 2002). Most organizations that have tried pair
173 programming eventually settle into using pairs for part of their work but not all
174 of it (Boehm and Turner 2004).

175 *Rotate pairs and work assignments regularly*

176 In pair programming, as with other collaborative development practices, benefit
177 arises from different programmers learning different parts of the system. Rotate
178 pair assignments regularly to encourage cross-pollination—some experts
179 recommend changing pairs as often as daily (Reifer 2002).

180 *Encourage pairs to match each other's pace*

181 One partner going too fast limits the benefit of having the other partner. The
182 faster partner needs to slow down, or the pair should be broken up and
183 reconfigured with different partners.

184 *Make sure both partners can see the monitor*

185 Even a seemingly-mundane issue like being able to see the monitor can cause
186 problems.

187 *Don't force people who don't like each other to pair*

188 Sometimes personality conflicts prevent people from pairing effectively. It's
189 pointless to force people who don't get along to pair, so be sensitive to
190 personality matches (Beck 2000, Reifer 2002).

191 *Avoid pairing all newbies*

192 Pair programming works best when at least one of the partners has paired before
193 (Larman 2004).

194 *Assign a team leader*

195 If your whole team wants to do 100 percent of its programming in pairs, you'll
196 still need to assign one person to coordinate work assignments, be held
197 accountable for results, and act as the point of contact for people outside the
198 project.

199 **Benefits of Pair Programming**

200 The basic concept of pair programming is simple, but it produces numerous
201 benefits:

- 202 • It holds up better under stress than solo development. Pairs help keep each
203 other honest and encourage each other to keep code quality high even when
204 there's pressure to write quick and dirty code.

- 205 • Code quality improves. The readability and understandability of the code
206 tends to rise to the level of the best programmer on the team.
207 • It produces all the other general benefits of collaborative construction
208 including disseminating corporate culture, mentoring junior programmers,
209 and fostering collective ownership

CC2E.COM/2192

210 **CHECKLIST: Effective Pair Programming**

- 211 Do you have a coding standard to support pair programming that's focused
212 on programming rather than on philosophical coding-style discussions?
213 Are both partners participating actively?
214 Are you avoiding pair programming everything, instead selecting the
215 assignments that will really benefit from pair programming?
216 Are you rotating pair assignments and work assignments regularly?
217 Are the pairs well matched in terms of pace and personality?
218 Is there a team leader to act as the focal point for management and other
219 people outside the project?
-

221 **21.3 Formal Inspections**

222 **FURTHER READING** If you
223 want to read the original
224 article on inspections, see
225 “Design and Code
226 Inspections to Reduce Errors
227 in Program Development”
228 (Fagan 1976).

An inspection is a specific kind of review that has been shown to be extremely effective in detecting defects and to be relatively economical compared to testing. Inspections were developed by Michael Fagan and used at IBM for several years before Fagan published the paper that made them public. Although any review involves reading designs or code, an inspection differs from a run-of-the-mill review in several key ways:

- 228 • Checklists focus the reviewers' attention on areas that have been problems in
229 the past.
- 230 • The emphasis is on defect detection, not correction.
- 231 • Reviewers prepare for the inspection meeting beforehand and arrive with a
232 list of the problems they've discovered.
- 233 • Distinct roles are assigned to all participants.
- 234 • The moderator of the inspection isn't the author of the work product under
235 inspection.
- 236 • The moderator has received specific training in moderating inspections.

- 237
- Data is collected at each inspection and is fed into future inspections to improve them.
- 238
- General management doesn't attend the inspection meeting. Technical leaders might.
- 239

240

241 **What Results Can You Expect from Inspections?**

242 **HARD DATA**

243 Individual inspections typically catch about 60% of defects, which is higher than
244 other techniques except prototyping and high-volume beta testing. These results
245 have been confirmed numerous times at organizations including Harris BCSD,
246 National Software Quality Experiment, Software Engineering Institute, Hewlett
Packard, and so on (Shull. et al 2002).

247 The combination of design and code inspections usually removes 70-85 percent
248 or more of the defects in a product (Jones 1996). Inspections identify error-prone
249 classes early, and Capers Jones reports that they result in 20-30 percent fewer
250 defects per 1000 lines of code than less formal review practices. Designers and
251 coders learn to improve their work through participating in inspections, and
252 inspections increase productivity by about 20 percent (Fagan 1976, Humphrey
253 1989, Gilb and Graham 1993, Wiegers 2002). On a project that uses inspections
254 for design and code, the inspections will take up about 10-15 percent of project
255 budget, and will typically reduce overall project cost.

256 Inspections can also be used for assessing progress, but it is the technical
257 progress that is assessed. That usually means answering two questions: (1) Is the
258 technical work being done? and (2) Is the technical work being done *well*? The
259 answers to both questions are by-products of formal inspections.

260

Roles During an Inspection

261 One key characteristic of an inspection is that each person involved has a distinct
262 role to play. Here are the roles:

263

Moderator

264 The moderator is responsible for keeping the inspection moving at a rate that's
265 fast enough to be productive but slow enough to find the most errors possible.
266 The moderator must be technically competent—not necessarily an expert in the
267 particular design or code under inspection, but capable of understanding relevant
268 details. This person manages other aspects of the inspection, such as distributing
269 the design or code to be reviewed and the inspection checklist, setting up a
270 meeting room, reporting inspection results, and following up on the action items
271 assigned at the inspection meeting.

272

273

274

275

276

277

278

279

280

281

282

283

284

285

286

287

288

289

290

291

292

293

294

295

296

297

298

299

300

301

302

303

304

305

306

307

308

309

Author

The person who wrote the design or code plays a relatively minor role in the inspection. Part of the goal of an inspection is to be sure that the design or code speaks for itself. If the design or code under inspection turns out to be unclear, the author will be assigned the job of making it clearer. Otherwise, the author's duties are to explain parts of the design or code that are unclear and, occasionally, to explain why things that seem like errors are actually acceptable. If the project is unfamiliar to the reviewers, the author might present an overview of the project in preparation for the inspection meeting.

Reviewer

A reviewer is anyone who has a direct interest in the design or code but who is not the author. A reviewer of a design might be the programmer who will implement the design. A tester or higher-level architect might also be involved. The role of the reviewers is to find defects. They usually find defects during preparation, and, as the design or code is discussed at the inspection meeting, the group should find considerably more defects.

Scribe

The scribe records errors that are detected and the assignments of action items during the inspection meeting. Sometimes the scribe's role is performed by the moderator and sometimes by another person. Neither the author nor a reviewer should be the scribe.

Management

Not usually a good idea. The point of a software inspection is that it is a purely technical review. Management's presence changes the technical interactions; people feel that they, instead of the review materials, are under evaluation, which changes the focus from technical to political. Management has a right to know the results of an inspection, and an inspection report is prepared to keep management informed.

Similarly, under no circumstances should inspection results be used for performance appraisals. Don't kill the goose that lays the golden eggs. Code examined in an inspection is still under development. Evaluation of performance should be based on final products, not on work that isn't finished.

Overall, an inspection should have no fewer than three participants. It's not possible to have a separate moderator, author, and reviewer with fewer than three people, and those roles shouldn't be combined. Traditional advice is to limit an inspection to about six people because, with any more, the group becomes too large to manage. Researchers have generally found that having more than two to three reviewers doesn't appear to increase the number of defects found (Bush

310 and Kelly 1989, Porter and Votta 1997). However, these general findings are not
311 unanimous, and results appear to vary depending on the kind of material being
312 inspected (Wiegers 2002). Pay attention to your experience and adjust your
313 approach accordingly.

314 General Procedure for an Inspection

315 An inspection consists of several distinct stages:

316 Planning

317 The author gives the design or code to the moderator. The moderator decides
318 who will review the material and when and where the inspection meeting will
319 occur, and then distributes the design or code and a checklist that focuses the
320 attention of the inspectors.

321 Overview

322 When the reviewers aren't familiar with the project they are reviewing, the
323 author can spend up to an hour or so describing the technical environment within
324 which the design or code has been created. Having an overview tends to be a
325 dangerous practice because it can lead to a glossing over of unclear points in the
326 design or code under inspection. The design or code should speak for itself; the
327 overview shouldn't speak for it.

328 Preparation

329 **CROSS-REFERENCE** For
330 a list of checklists you can
331 use to improve code quality,
see page 000.

332 For a review of application code written in a high-level language, reviewers can
333 prepare at about 500 lines of code per hour. For a review of system code written
334 in a high-level language, reviewers can prepare at only about 125 lines of code
335 per hour (Humphrey 1989). The most effective rate of review varies a great deal,
336 so keep records of preparation rates in your organization to determine the rate
337 that's most effective in your environment.

338 Perspectives

339 Some organizations have found that inspections are more effective when each
340 reviewer is assigned a specific perspective. A reviewer might be asked to inspect
341 the design or code from the point of view of the maintenance programmer, the
342 customer, or the designer, for example. Research on perspective-based reviews
343 has not been comprehensive, but it suggests that perspective-based reviews
344 might uncover more errors than general reviews.

Scenarios

An additional variation in inspection preparation is to assign each reviewer one or more scenarios to check. Scenarios can involve specific questions that a reviewer is assigned to answer, such as "Are there any requirements that are not satisfied by this design?" A scenario might also involve a specific task that a reviewer is assigned to perform, such as listing the specific requirements that a particular design element satisfies.

Inspection Meeting

The moderator chooses someone—usually someone other than the author—to paraphrase the design or read the code (Wiegers 2003). All logic is explained, including each branch of each logical structure. During this presentation, the scribe records errors as they are detected, but discussion of an error stops as soon as it's recognized as an error. The scribe notes the type and the severity of the error, and the inspection moves on.

The rate at which the design or the code is considered should be neither too slow nor too fast. If it's too slow, attention can lag and the meeting won't be productive. If it's too fast, the group can overlook errors it would otherwise catch. Optimal inspection rates vary from environment to environment, as preparation rates do. Keep records so that over time you can determine the optimal rate for your environment. Other organizations have found that for system code, an inspection rate of 90 lines of code per hour is optimal. For applications code, the inspection rate can be as rapid as 500 lines of code per hour (Humphrey 1989). An average of about 150-200 non-blank, non-comment source statements per hour is a good place to start (Wiegers 2002).

Don't discuss solutions during the meeting. The group should stay focused on identifying defects. Some inspection groups don't even allow discussion about whether a defect is really a defect. They assume that if someone is confused enough to think it's a defect, the design, code, or documentation needs to be clarified.

The meeting generally should not last more than two hours. This doesn't mean that you have to fake a fire alarm to get everyone out at the two-hour mark, but experience at IBM and other companies has been that reviewers can't concentrate for much more than about two hours at a time. For the same reason, it's unwise to schedule more than one inspection on the same day.

Inspection Report

Within a day of the inspection meeting, the moderator produces an inspection report (email, or equivalent) that lists each defect, including its type and severity. The inspection report helps to ensure that all defects will be corrected and is used to develop a checklist that emphasizes problems specific to the organization. If

384 you collect data on the time spent and the number of errors found over time, you
385 can respond to challenges about inspection's efficacy with hard data. Otherwise,
386 you'll be limited to saying that inspections seem better. That won't be as
387 convincing to someone who thinks testing seems better. You'll also be able to
388 tell if inspections aren't working in your environment and modify or abandon
389 them, as appropriate. Data collection is also important because any new
390 methodology needs to justify its existence.

391 **Rework**

392 The moderator assigns defects to someone, usually the author, for repair. The
393 assignee resolves each defect on the list.

394 **Follow-Up**

395 The moderator is responsible for seeing that all rework assigned during the
396 inspection is carried out. If more than 5 percent of the design or code needs to be
397 reworked, the whole inspection process should be repeated. If less, the moderator
398 may still call for a re-inspection or choose to verify the rework personally.

399 **Third-Hour Meeting**

400 Even though during the inspection participants aren't allowed to discuss
401 solutions to the problems raised, some might still want to. You can hold an
402 informal, third-hour meeting to allow interested parties to discuss solutions after
403 the official inspection is over.

404 **Fine-Tuning the Inspection**

405 Once you become skilled at performing inspections "by the book," you can
406 usually find several ways to improve them. Don't introduce changes willy-nilly,
407 though. "Instrument" the inspection process so that you know whether your
408 changes are beneficial.

409 Companies have often found that removing or combining any of the parts costs
410 more than is saved (Fagan 1986). If you're tempted to change the inspection
411 process without measuring the effect of the change, don't. If you have measured
412 the process and you know that your changed process works better than the one
413 described here, go right ahead.

414 As you do inspections, you'll notice that certain kinds of errors occur more
415 frequently than other kinds. Create a checklist that calls attention to those kinds
416 of errors so that reviewers will focus on them. Over time, you'll find kinds of
417 errors that aren't on the checklist; add those to it. You might find that some
418 errors on the initial checklist cease to occur; remove those. After a few
419 inspections, your organization will have a checklist for inspections customized to
420 its needs, and it might also have some clues about trouble areas in which its

421
422

423

424 **FURTHER READING** For a
425 discussion of egoless
426 programming, see *The*
427 *Psychology of Computer*
428 *Programming, 2d Ed.*
429 (Weinberg 1998).

430
431
432
433

434
435
436
437
438
439
440
441

442
443
444
445

446

447
448
449
450
451
452
453
454
455
456

programmers need more training or support. Limit your checklist to one page or less. Longer ones are hard to use at the level of detail needed in an inspection.

Egos in Inspections

The point of the inspection itself is to discover defects in the design or code. It is not to explore alternatives or to debate about who is right and who is wrong. The point is most certainly not to criticize the author of the design or code. The experience should be a positive one for the author in which it's obvious that group participation improves the program and is a learning experience for all involved. It should not convince the author that some people in the group are jerks or that it's time to look for a new job. Comments like "Anyone who knows Java knows that it's more efficient to loop from 0 to num-1, not 1 to num" are totally inappropriate, and if they occur, the moderator should make their inappropriateness unmistakably clear.

Because the design or code is being criticized and the author probably feels somewhat attached to it, the author will naturally feel some of the heat directed at the code. The author should anticipate hearing criticisms of several defects that aren't really defects and several more that seem debatable. In spite of that, the author should acknowledge each alleged defect and move on.

Acknowledging a criticism doesn't imply that the author agrees it's true. The author should not try to defend the work under review. After the review, the author can think about each point in private and decide whether it's valid.

Reviewers must remember that the author has the ultimate responsibility for deciding what to do about a defect. It's fine to enjoy finding defects (and outside the review, to enjoy proposing solutions), but each reviewer must respect the author's ultimate right to decide how to resolve an error.

Inspections and Code Complete

I had a personal experience using inspections on the second edition of *Code Complete*. For the first edition of this book I initially wrote a rough draft. After letting the rough draft of each chapter sit in a drawer for a week or two, I reread the chapter cold and corrected the errors I found. I then circulated the revised chapter to about a dozen peers for review, several of whom reviewed it quite thoroughly. I corrected the errors they found. After a few more weeks, I reviewed it again myself and corrected more errors. Finally, I submitted the manuscript to the publisher, where it was reviewed by a copy editor, technical editor, and proofreader. The book was in print for more than 10 years, and readers sent in about 200 corrections during that time.

457 You might think there wouldn't be many errors left in the book that had gone
458 through all that review activity. But that wasn't the case. To create the second
459 edition, I used formal inspections of the first edition to identify issues that
460 needed to be addressed in the second edition. Teams of 3-4 reviewers prepared
461 according to the guidelines described in this chapter. Somewhat to my surprise,
462 our formal inspections found several hundred errors in the first edition text that
463 had not previously been detected through any of the numerous review activities.

464 If I had had any doubts about the value of formal inspections, my experience in
465 creating the second edition of *Code Complete* eliminated them.

466 Inspection Summary

467 Inspection checklists encourage focused concentration. The inspection process is
468 systematic because of its standard checklists and standard roles. It is also self-
469 optimizing because it uses a formal feedback loop to improve the checklists and
470 to monitor preparation and inspection rates. With this control over the process
471 and continuing optimization, inspection quickly becomes a powerful technique
472 almost no matter how it begins.

473 **FURTHER READING** For
474 more details on the SEI's
475 concept of developmental
476 maturity, see *Managing the*
477 *Software Process* (Humphrey
478 1989).

479
480

CC2E.COM/2199

481

- 482
-
- 483 **CHECKLIST: Effective Inspections**
-
- 484
- 485 Do you have checklists that focus reviewer attention on areas that have been
486 problems in the past?
 - 487 Is the emphasis on defect detection rather than correction?
 - 488 Are inspectors given enough time to prepare before the inspection meeting,
489 and is each one prepared?
 - 490 Does each participant have a distinct role to play?
 - 491 Does the meeting move at a productive rate?
 - 492 Is the meeting limited to two hours?
 - 493 Has the moderator received specific training in conducting inspections?
 - 494 Is data about error types collected at each inspection so that you can tailor
495 future checklists to your organization?

- 493 Is data about preparation and inspection rates collected so that you can
494 optimize future preparation and inspections?
495 Are the action items assigned at each inspection followed up, either
496 personally by the moderator or with a re-inspection?
497 Does management understand that it should not attend inspection meetings?
498
-

499

500 21.4 Other Kinds of Collaborative Development Practices

501 Other kinds of collaboration haven't accumulated the body of empirical support
502 that inspections or pair programming have, so they're covered in less depth here.
503 The kinds covered in this section include walkthroughs, code reading, and dog-
504 and-pony shows.

505

506 Walkthroughs

507 A walkthrough is a popular kind of review. The term is loosely defined, and at
508 least some of its popularity can be attributed to the fact that people can call
 virtually any kind of review a "walkthrough."

509 Because the term is so loosely defined, it's hard to say exactly what a walk-
510 through is. Certainly, a walkthrough involves two or more people discussing a
511 design or code. It might be as informal as an impromptu bull session around a
512 whiteboard; it might be as formal as a scheduled meeting with a Microsoft
513 Powerpoint presentation prepared by the art department and a formal summary
514 sent to management. In one sense, "where two or three are gathered together,"
515 there is a walkthrough. Proponents of walkthroughs like the looseness of such a
516 definition, so I'll just point out a few things that all walkthroughs have in
517 common and leave the rest of the details to you:

- 518 KEY POINT
- 519
- 520 • The walkthrough is usually hosted and moderated by the author of the
 design or code under review.
 - 521 • The walkthrough focuses on technical issues; it's a working meeting.
 - 522 • All participants prepare for the walkthrough by reading the design or code
 and looking for errors.
 - 523 • The walkthrough is a chance for senior programmers to pass on experience
 and corporate culture to junior programmers. It's also a chance for junior
 programmers to present new methodologies and to challenge timeworn,
 possibly obsolete, assumptions.

- 527
 - A walkthrough usually lasts 30 to 60 minutes.
 - The emphasis is on error detection, not correction.
 - Management doesn't attend.
 - The walkthrough concept is flexible and can be adapted to the specific needs of the organization using it.
- 528
- 529
- 530
- 531

532 **What Results Can You Expect From A Walkthrough?**

533 Used intelligently and with discipline, a walkthrough can produce results similar
534 to those of an inspection—that is, it can typically find between 30 and 70 percent
535 of the errors in a program (Myers 1979, Boehm 1987b, Yourdon 1989b, Jones
536 1996). But in general, walkthroughs have been found to be significantly less
537 effective than inspections (Jones 1996).

538 | **HARD DATA**

539 Used unintelligently, walkthroughs are more trouble than they're worth. The low
540 end of their effectiveness, 30 percent, isn't worth much, and at least one
541 organization (Boeing Computer Services) found peer reviews of code to be
542 “extremely expensive.” Boeing found it was difficult to motivate project
543 personnel to apply walkthrough techniques consistently, and when project
pressures increased, walkthroughs became nearly impossible (Glass 1982).

544 I've become more critical of walkthroughs during the past 10 years as a result of
545 what I've seen in my company's consulting business. I've found that when
546 people have bad experiences with technical reviews, it is nearly always with
547 informal practices such as walkthroughs rather than with formal inspections. A
548 review is basically a meeting, and meetings are expensive. If you're going to
549 incur the overhead of holding a meeting, it's worthwhile to structure the meeting
550 as a formal inspection. If the work product you're reviewing doesn't justify the
551 overhead of a formal inspection, it doesn't justify the overhead of a meeting at
552 all. You're better off using document reading or another less interactive
553 approach.

554 Inspections seem to be more effective than walkthroughs at removing errors. So
555 why would anyone choose to use walkthroughs?

556 If you have a large review group, a walkthrough is a good review choice because
557 it brings many diverse viewpoints to bear on the item under review. If everyone
558 involved in the walkthrough can be convinced that the solution is all right, it
559 probably doesn't have any major flaws.

560 If reviewers from other organizations are involved, a walkthrough might also be
561 preferable. Roles in an inspection are more formalized and require some practice
562 before people perform them effectively. Reviewers who haven't participated in

563

564

565 KEY POINT

566

567

568

569

570

571

572

573 HARD DATA

574

575

576

577

578

579

580

581

582

583

584

585

586

587

588

589

590

591

592

593

594

595

596 KEY POINT

597

598

inspections before are at a disadvantage. If you want to solicit their contributions, a walkthrough might be the best choice.

Inspections are more focused than walkthroughs and generally pay off better. Consequently, if you're choosing a review standard for your organization, think hard about choosing inspections.

Code Reading

Code reading is an alternative to inspections and walkthroughs. In code reading, you read source code and look for errors. You also comment on qualitative aspects of the code such as its design, style, readability, maintainability, and efficiency.

A study at NASA's Software Engineering Laboratory found that code reading detected about 3.3 defects per hour of effort. Testing detected about 1.8 errors per hour (Card 1987). Code reading also found 20 to 60 percent more errors over the life of the project than the various kinds of testing did.

Like the idea of a walkthrough, the concept of code reading is loosely defined. A code reading usually involves two or more people reading code independently and then meeting with the author of the code to discuss it. Here's how code reading goes:

- In preparation for the meeting, the author of the code hands out source listings to the code readers. The listings are from 1000 to 10,000 lines of code; 4000 lines is typical.
- Two or more people read the code. Use at least two people to encourage competition between the reviewers. If you use more than two, measure everyone's contribution so that you know how much the extra people contribute.
- Reviewers read the code independently. Estimate a rate of about 1000 lines a day.
- When the reviewers have finished reading the code, the code-reading meeting is hosted by the author of the code. The meeting lasts one or two hours and focuses on problems discovered by the code readers. No one makes any attempt to walk through the code line by line. The meeting is not even strictly necessary.
- The author of the code fixes the problems identified by the reviewers.

The difference between code reading on the one hand and inspections and walkthroughs on the other is that code reading focuses more on individual review of the code than on the meeting. The result is that each reviewer's time is

599
600
601
602
603
604

focused on finding problems in the code. Less time is spent in meetings in which each person contributes only part of the time and in which a substantial amount of the effort goes into moderating group dynamics. Less time is spent delaying meetings until each person in the group can meet for two hours. Code readings are especially valuable in situations in which reviewers are geographically dispersed.

605 **HARD DATA**

606
607
608

A study of 13 reviews at AT&T found that the importance of the review meeting itself was overrated; 90 percent of the defects were found in preparation for the review meeting, and only about 10 percent were found during the review itself (Votta 1991, Glass 1999).

609

Dog-and-Pony Shows

610
611
612
613
614
615

Dog-and-pony shows are reviews in which a software product is demonstrated to a customer. Customer reviews are common in software developed for government contracts, which often stipulate that reviews will be held for requirements, design, and code. The purpose of a dog-and-pony show is to demonstrate to the customer that the project is OK, so it's a management review rather than a technical review.

616
617
618
619
620

Don't rely on dog-and-pony shows to improve the technical quality of your products. Preparing for them might have an indirect effect on technical quality, but usually more time is spent in making good-looking Microsoft Powerpoint slides than in improving the quality of the software. Rely on inspections, walkthroughs, or code reading for technical quality improvements.

621
622

Comparison of Collaborative Construction Techniques

623
624

What are the differences between the various kinds of collaborative construction? Here's a summary of the major characteristics:

625

Table 21-1. Comparison of Collaborative Construction Techniques

Property	Pair Programming	Formal Inspection	Informal Review (Walkthroughs)
Defined participant roles	Yes	Yes	No
Formal training in how to perform the roles	Maybe, through coaching	Yes	No
Who "drives" the collaboration	Person with the keyboard	Moderator	Author, usually

Focus of collaboration	Design, coding, testing, and defect correction	Defect detection only	Varies
Focused review effort—looks for the most frequently found kinds of errors	Informal, if at all	Yes	No
Follow-up to reduce bad fixes	Yes	Yes	No
Fewer future errors because of detailed error feedback to individual programmers	Incidental	Yes	Incidental
Improved process efficiency from analysis of results	No	Yes	No
Useful for non-construction activities	Possibly	Yes	Yes
Typical percentage of defects found	40%-60%	45%-70%	20-40%

626

627

628

629

Pair programming doesn't have decades of data supporting its effectiveness like formal inspections does, but the initial data suggests it's on roughly equal footing with inspections, and anecdotal reports have also been positive.

630

631

632

633

634

635

636

637

638

If pair programming and formal inspections produce similar results for quality, cost, and schedule, the choice between pair programming and formal inspections becomes a matter of personal style preference than of technical substance. Some people prefer to work solo, only occasionally breaking out of solo mode for inspection meetings. Others prefer to spend more of their time directly working with others. The choice between the two techniques can be driven by the work-style preference of a team's specific developers, and subgroups within the team might even be allowed to choose which way they would like to do most of their work.

CC2E.COM/2106

639

Additional Resources

640

Pair Programming

641

Williams, Laurie and Robert Kessler. *Pair Programming Illuminated*, Boston, Mass.: Addison Wesley, 2002. This book explains the detailed ins and outs of

642

643 pair programming including how to handle various personality matches (expert
644 and inexpert, introvert and extrovert) and other implementation issues.

645 Beck, Kent. *Extreme Programming: Embrace Change*, Reading, Mass.: Addison
646 Wesley, 2000. This book touches on pair programming briefly and shows how it
647 can be used in conjunction with other mutually supportive techniques, including
648 coding standards, frequent integration, and regression testing.

649 Reifer, Donald. "How to Get the Most Out of Extreme Programming/Agile
650 Methods," *Proceedings, XP/Agile Universe 2002*. New York: Springer; pp. 185-
651 196. This paper summarizes industrial experience with extreme programming
652 and agile methods and presents keys to success for pair programming.

653 **Inspections**

654 Wiegers, Karl. *Peer Reviews in Software: A Practical Guide*, Boston, Mass.:
655 Addison Wesley, 2002. This well-written book describes the ins and outs of
656 various kinds of reviews including formal inspections and other, less formal
657 practices. It's well researched, has a practical focus, and is easy to read.

658 Gilb, Tom and Dorothy Graham. *Software Inspection*. Wokingham, England:
659 Addison-Wesley, 1993. This contains a thorough discussion of inspections circa
660 the early 1990s. It has a practical focus and includes case studies that describe
661 experiences several organizations have had in setting up inspection programs.

662 Fagan, Michael E. "Design and Code Inspections to Reduce Errors in Program
663 Development." *IBM Systems Journal* 15, no. 3 (1976): 182-211.

664 Fagan, Michael E. "Advances in Software Inspections." *IEEE Transactions on*
665 *Software Engineering*, SE-12, no. 7 (July 1986): 744-51. These two articles
666 were written by the developer of inspections. They contain the meat of what you
667 need to know to run an inspection, including all the standard inspection forms.

668 **Relevant Standards**

669 *IEEE Std 1028-1997, Standard for Software Reviews*

670 *IEEE Std 730-2002, Standard for Software Quality Assurance Plans*

671 **Key Points**

- 672 • Collaborative development practices tend to find a higher percentage of
673 defects than testing and to find them more efficiently.

- 674 ● Collaborative development practices tend to find different kinds of errors
675 than testing does, implying that you need to use both reviews and testing to
676 ensure the quality of your software.
- 677 ● Formal inspections use checklists, preparation, well-defined roles, and
678 continual process improvement to maximize error-detection efficiency. They
679 tend to find more defects than walkthroughs.
- 680 ● Pair programming typically costs about the same as inspections and
681 produces similar quality code. Pair programming is especially valuable when
682 schedule reduction is desired. Some developers prefer working in pairs to
683 working solo.
- 684 ● Formal inspections can be used on workproducts such as requirements,
685 designs, and test cases as well as on code.
- 686 ● Walkthroughs and code reading are alternatives to inspections. Code reading
687 offers more flexibility in using each person's time effectively.

22

Developer Testing

Contents

- 22.1 Role of Developer Testing in Software Quality
- 22.2 Recommended Approach to Developer Testing
- 22.3 Bag of Testing Tricks
- 22.4 Typical Errors
- 22.5 Test-Support Tools
- 22.6 Improving Your Testing
- 22.7 Keeping Test Records

Related Topics

The software-quality landscape: Chapter 20

Collaborative construction practices: Chapter 21

Debugging: Chapter 23

Integration: Chapter 29

Prerequisites to construction: Chapter 3

TESTING IS THE MOST POPULAR quality-improvement activity—a practice supported by a wealth of industrial and academic research and by commercial experience.

Software is tested in numerous ways, some of which are typically performed by developers and some of which are more commonly performed by specialized test personnel:

Unit testing is the execution of a complete class, routine, or small program that has been written by a single programmer or team of programmers, which is tested in isolation from the more complete system.

Component testing is the execution of a class, package, small program, or other program element that involves the work of multiple programmers or programming teams, which is tested in isolation from the more complete system.

29 *Integration testing* is the combined execution of two or more classes, packages,
30 components, subsystems that have been created by multiple programmers or
31 programming teams. This kind of testing typically starts as soon as there are two
32 classes to test and continues until the entire system is complete.

33 *Regression testing* is the repetition of previously executed test cases for the
34 purpose of finding defects in software that previously passed the same set of
35 tests.

36 *System testing* is the execution of the software in its final configuration,
37 including integration with other software and hardware systems. It tests for
38 security, performance, resource loss, timing problems, and other issues that can't
39 be tested at lower levels of integration.

40 In this chapter, “testing” refers to testing by the developer—which typically
41 consists of unit tests, component tests, and integration tests, and which may
42 sometimes consist of regression tests and system tests. Numerous additional
43 kinds of testing are performed by specialized test personnel and are rarely
44 performed by developers (including beta tests, customer-acceptance tests,
45 performance tests, configuration tests, platform tests, stress tests, usability tests,
46 and so on). These kinds of testing are not discussed further in this chapter.

47 Testing is usually broken into two broad categories: black box testing and white
48 box (or glass box) testing. “Black box testing” refers to tests in which the tester
49 cannot see the inner workings of the item being tested. This obviously does not
50 apply when you test code that you have written! “White box testing” refers to
51 tests in which the tester is aware of the inner workings of the item being tested.
52 This is the kind of testing that you as a developer use to test your own code. Both
53 black box and white box testing have strengths and weaknesses; this chapter
54 focuses on white box testing because that is the kind of testing that developers
55 perform.

56 Some programmers use the terms “testing” and “debugging” interchangeably,
57 but careful programmers distinguish between the two activities. Testing is a
58 means of detecting errors. Debugging is a means of diagnosing and correcting
59 the root causes of errors that have already been detected. This chapter deals
60 exclusively with error detection. Error correction is discussed in detail in Chapter
61 23, “Debugging.”

62 The whole topic of testing is much larger than the subject of testing during
63 construction. System testing, stress testing, black box testing, and other topics
64 for test specialists are discussed in the “Additional Resources” section at the end
65 of the chapter.

66

67

68 **CROSS-REFERENCE** For
69 details on reviews, Chapter
70 21, "Collaborative
Construction."

71

72

73

74

75

76 *Programs do not acquire
77 bugs as people acquire
78 germs, by hanging
79 around other buggy
programs. Programmers
must insert them.*

80 —Harlan Mills

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97 **HARD DATA**

98

99

100

101

102

103

104

22.1 Role of Developer Testing in Software Quality

Testing is an important part of any software-quality program, and in many cases it's the only part. This is unfortunate, because collaborative development practices in their various forms have been shown to find a higher percentage of errors than testing does, and they cost less than half as much per error found as testing does (Card 1987, Russell 1991, Kaplan 1995). Individual testing steps (unit test, component test, and integration test) typically find less than 50% of the errors present each. The combination of testing steps often finds less than 60% of the errors present (Jones 1998).

If you were to list a set of software-development activities on "Sesame Street" and ask, "Which of these things is not like the others?", the answer would be "Testing." Testing is a hard activity for most developers to swallow for several reasons:

- Testing's goal runs counter to the goals of other development activities. The goal is to find errors. A successful test is one that breaks the software. The goal of every other development activity is to prevent errors and keep the software from breaking.
- Testing can never completely prove the absence of errors. If you have tested extensively and found thousands of errors, does it mean that you've found all the errors or that you have thousands more to find? An absence of errors could mean ineffective or incomplete test cases as easily as it could mean perfect software.
- Testing by itself does not improve software quality. Test results are an indicator of quality, but in and of themselves, they don't improve it. Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often. What you eat before you step onto the scale determines how much you will weigh, and the software-development techniques you use determine how many errors testing will find. If you want to lose weight, don't buy a new scale; change your diet. If you want to improve your software, don't just test more; develop better.
- Testing requires you to assume that you'll find errors in your code. If you assume you won't, you probably won't, but only because you'll have set up a self-fulfilling prophecy. If you execute the program hoping that it won't have any errors, it will be too easy to overlook the errors you find. In a study that has become a classic, Glenford Myers had a group of experienced programmers test a program with 15 known defects. The average programmer found only 5 of the 15 errors. The best found only 9. The main source of undetected errors was that erroneous output was not examined

105
106 carefully enough. The errors were visible but the programmers didn't notice
them (Myers 1978).

107
108 You must hope to find errors in your code. Such a hope might seem like an
109 unnatural act, but you should hope that it's you who finds the errors and not
someone else.

110 A key question is, How much time should be spent in developer testing on a
111 typical project? A commonly cited figure for all testing is 50% of the time spent
112 on the project, but that's misleading for several reasons. First, that particular
113 figure combines testing and debugging; testing alone takes less time. Second,
114 that figure represents the amount of time that's typically spent rather than the
115 time that should be spent. Third, the figure includes independent testing as well
116 as developer testing.

117 As Figure 22-1 shows, depending on the project's size and complexity,
118 developer testing should probably take 8 to 25% of the total project time. This is
119 consistent with much of the data that has been reported.

120 **Error! Objects cannot be created from editing field codes.**

121 **F22xx01**

122 **Figure 22-1**

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

126 A second question is, What do you do with the results of developer testing? Most
127 immediately, you can use the results to assess the reliability of the product under
128 development. Even if you never correct the defects that testing finds, testing
129 describes how reliable the software is. Another use for the results is that they can
130 and usually do guide corrections to the software. Finally, over time, the record of
131 defects found through testing helps reveal the kinds of errors that are most
132 common. You can use this information to select appropriate training classes,
133 direct future technical review activities, and design future test cases.

134 **Testing During Construction**

135 **KEY POINT**
136 The big, wide world of testing sometimes ignores the subject of this chapter:
137 "white-box" or "glass-box" testing. You generally want to design a class to be a
138 black box—a user of the class won't have to look past the interface to know
139 what the class does. In testing the class, however, it's advantageous to treat it as
140 a glass box, to look at the internal source code of the class as well as its inputs
141 and outputs. If you know what's inside the box, you can test the class more
142 thoroughly. Of course you also have the same blind spots in testing the class that
you had in writing it, and so there are some advantages to black box testing too.

143 **CROSS-REFERENCE** Top-
144 down, bottom-up,
145 incremental, and partitioned
146 builds used to be thought of
147 as alternative approaches to
148 testing, but they are really
149 techniques for integrating a
150 program. These alternatives
151 are discussed in Chapter 29,
“Integration.”

152
153
154
155
156
157

During construction you generally write a routine or class, check it mentally, and then review it or test it. Regardless of your integration or system-testing strategy, you should test each unit thoroughly before you combine it with any others. If you’re writing several routines, you should test them one at a time. Routines aren’t really any easier to test individually, but they’re much easier to debug. If you throw several untested routines together at once and find an error, any of the several routines might be guilty. If you add one routine at a time to a collection of previously tested routines, you know that any new errors are the result of the new routine or of interactions with the new routine. The debugging job is easier.

Collaborative construction practices have many strengths to offer that testing can’t match. But part of the problem with testing is that testing often isn’t performed as well as it could be. A developer can perform hundreds of tests and still achieve only partial code coverage. A *feeling* of good test coverage doesn’t mean that actual test coverage is adequate. An understanding of basic test concepts can support better testing and raise testing’s effectiveness.

158
159

160
161

162
163
164
165
166
167

168
169
170
171

172
173
174
175
176

177
178

22.2 Recommended Approach to Developer Testing

A systematic approach to developer testing maximizes your ability to detect errors of all kinds with a minimum of effort. Be sure to cover this ground:

- Test for each relevant requirement to make sure that the requirements have been implemented. Plan the test cases for this step at the requirements stage or as early as possible—preferably before you begin writing the unit to be tested. Consider testing for common omissions in requirements. The level of security, storage, the installation procedure, and system reliability are all fair game for testing and are often overlooked at requirements time.
- Test for each relevant design concern to make sure that the design has been implemented. Plan the test cases for this step at the design stage or as early as possible—before you begin the detailed coding of the routine or class to be tested.
- Use “basis testing” to add detailed test cases to those that test the requirements and the design. Add data-flow tests, and then add the remaining test cases needed to thoroughly exercise the code. At a minimum, you should test every line of code. Basis testing and data-flow testing are described later in this chapter.

Build the test cases along with the product. This can help avoid errors in requirements and design, which tend to be more expensive than coding errors.

179 Plan to test and find defects as early as possible because it's cheaper to fix
180 defects early.

181 **Test First or Test Last?**

182 Developers sometimes wonder whether it's better to write test cases after the
183 code has been written or beforehand (Beck 2003). The defect-cost increase graph
184 suggests that writing test cases first will minimize the amount of time between
185 when a defect is inserted into the code and when the defect is detected and
186 removed. This turns out to be one of many reasons to write test cases first:

- 187 • Writing test cases before writing the code doesn't take any more effort than
188 writing test cases after the code; it simply resequences the test-case-writing
189 activity.
- 190 • When you write test cases first, you detect defects earlier and you can
191 correct them more easily.
- 192 • Writing test cases first forces you to think at least a little bit about the
193 requirements and design before writing code, which tends to produce better
194 code.
- 195 • Writing test cases first exposes requirements problems sooner, before the
196 code is written, because it's hard to write a test case for a poor requirement.
- 197 • If you save your test cases (which you should), you can still test last, in
198 addition to testing first.

199 All in all, I think test-first programming is one of the most beneficial software
200 practices to emerge during the past decade and is a good general approach. But it
201 isn't a panacea, because it is subject to the general limitations of developer
202 testing, which are described next.

203 **Limitations of Developer Testing**

204 Watch for the following limitations with developer testing.

205 ***Developer tests tend to be “clean tests”***

206 Developers tend to test for whether the code works (clean tests) rather than to
207 find all the ways the code breaks (dirty tests). Immature testing organizations
208 tend to have about five clean tests for every dirty test. Mature testing
209 organizations tend to have five dirty tests for every clean test. This ratio is not
210 reversed by reducing the clean tests; it's done by creating 25 times as many dirty
211 tests (Boris Beizer in Johnson 1994).

Developer testing tends to have an optimistic view of test coverage

Average programmers believe they are achieving 95% test coverage, but they're typically achieving more like 80% test coverage in the best case, 30% in the worst case, and more like 50-60% in the average case (Boris Beizer in Johnson 1994).

Developer testing tends to skip more sophisticated kinds of test coverage

Most developers view the kind of test coverage known as "100% statement coverage" as adequate. This is a good start, but hardly sufficient. A better coverage standard is to meet what's called "100% branch coverage," with every predicate term being tested for at least one true and one false value. Section 22.3, "Bag of Testing Tricks," provides more details about how to accomplish this.

None of these points reduce the value of developer testing, but they do help put developer testing into proper perspective. As valuable as developer testing is, it isn't sufficient to provide adequate quality assurance on its own and should be supplemented with other practices including independent testing and collaborative construction techniques.

22.3 Bag of Testing Tricks

Why isn't it possible to prove that a program is correct by testing it? To use testing to prove that a program works, you'd have to test every conceivable input value to the program and every conceivable combination of input values. Even for simple programs, such an undertaking would become massively prohibitive. Suppose, for example, that you have a program that takes a name, an address, and a phone number and stores them in a file. This is certainly a simple program, much simpler than any whose correctness you'd really be worried about. Suppose further that each of the possible names and addresses is 20 characters long and that there are 26 possible characters to be used in them. This would be the number of possible inputs:

Name	26^{20} (20 characters, each with 26 possible choices)
Address	26^{20} (20 characters, each with 26 possible choices)
Phone Number	10^{10} (10 digits, each with 10 possible choices)
Total Possibilities	$= 26^{20} * 26^{20} * 10^{10} \approx 10^{66}$

Even with this relatively small amount of input, you have one-with-66-zeros possible test cases. To put this in perspective: If Noah had gotten off the ark and started testing this program at the rate of a trillion test cases per second, he would be far less than 1% of the way done today. Obviously, if you added a

243

244

245

246 **CROSS-REFERENCE** One
247 way of telling whether
248 you've covered all the code is
249 to use a coverage monitor.
250 For details, see "Coverage
Monitors" in "Coverage
Monitors" in Section 22.5,
251 later in this chapter.

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274 **CROSS-REFERENCE** This
275 procedure is similar to the
276 one for measuring
277 complexity in "How to
278 Measure Complexity" in
Section 19.6.
279

more realistic amount of data, the task of exhaustively testing all possibilities would become even more impossible.

Incomplete Testing

Since exhaustive testing is impossible, practically speaking, the art of testing is that of picking the test cases most likely to find errors. Of the 10^{66} possible test cases, only a few are likely to disclose errors that the others don't. You need to concentrate on picking a few that tell you different things rather than a set that tells you the same thing over and over.

When you're planning tests, eliminate those that don't tell you anything new—that is, tests on new data that probably won't produce an error if other, similar data didn't produce an error. Various people have proposed various methods of covering the bases efficiently, and several of these methods are discussed next.

Structured Basis Testing

In spite of the hairy name, structured basis testing is a fairly simple concept. The idea is that you need to test each statement in a program at least once. If the statement is a logical statement, say an *if* or a *while*, you need to vary the testing according to how complicated the expression inside the *if* or *while* is to make sure that the statement is fully tested. The easiest way to make sure that you've gotten all the bases covered is to calculate the number of paths through the program and then develop the minimum number of test cases that will exercise every path through the program.

You might have heard of "code coverage" testing or "logic coverage" testing. They are approaches in which you test all the paths through a program. Since they cover all paths, they're similar to structured basis testing, but they don't include the idea of covering all paths with a *minimal* set of test cases. If you use code coverage or logic coverage testing, you might create many more test cases than you would need to cover the same logic with structured basis testing.

You can compute the minimum number of cases needed for basis testing in the straightforward way outlined in Table 22-1.

Table 22-1. Determining the Number of Test Cases Needed for Structured Basis Testing

1. Start with 1 for the straight path through the routine.
2. Add 1 for each of the following keywords, or their equivalents: *if*, *while*, *repeat*, *for*, *and*, and *or*.
3. Add 1 for each case in a case statement. If the *case* statement doesn't have a default case, add 1 more.

Here's an example:

280
281
282 Count "1" for the routine itself.
283
284 Count "2" for the if.
285
286
287
288
289
290
291

Simple Example of Computing the Number of Paths Through a Java Program

```
Statement1;  
Statement2;  
if ( x < 10 ) {  
    Statement3;  
}  
Statement4;
```

In this instance, you start with one and count the *if* once to make a total of two. That means that you need to have at least two test cases to cover all the paths through the program. In this example, you'd need to have the following test cases:

- Statements controlled by *if* are executed ($x < 10$).
- Statements controlled by *if* aren't executed ($x \geq 10$).

The sample code needs to be a little more realistic to give you an accurate idea of how this kind of testing works. Realism in this case includes code containing defects.



G22xx01

The listing below is a slightly more complicated example. This piece of code is used throughout the chapter and contains a few possible errors.

Example of Computing the Number of Cases Needed for Basis Testing of a Java Program

```
1 // Compute Net Pay  
2 totalWithholdings = 0;  
3  
4 for ( id = 0; id < numEmployees; id++ ) {  
5  
6     // compute social security withholding, if below the maximum  
7     if ( m_employee[ id ].governmentRetirementWithheld < MAX_GOV_T_RETIREMENT ) {
```

Count "3" for the if.

```
310     governmentRetirement = ComputeGovernmentRetirement( m_employee[ id ] );
311 }
312
313     // set default to no retirement contribution
314     companyRetirement = 0;
315
316     // determine discretionary employee retirement contribution
317 Count "4" for the if and "5" for
318             the &&.
319
320         if ( m_employee[ id ].WantsRetirement &&
321             EligibleForRetirement( m_employee[ id ] ) ) {
322             companyRetirement = GetRetirement( m_employee[ id ] );
323         }
324
325         grossPay = ComputeGrossPay ( m_employee[ id ] );
326
327         // determine IRA contribution
328         personalRetirement = 0;
329
330         Count "6" for the if.
331         if ( EligibleForPersonalRetirement( m_employee[ id ] ) ) {
332             personalRetirement = PersonalRetirementContribution( m_employee[ id ],
333                         companyRetirement, grossPay );
334         }
335
336         // make weekly paycheck
337         withholding = ComputeWithholding( m_employee[ id ] );
338         netPay = grossPay - withholding - companyRetirement - governmentRetirement -
339                 personalRetirement;
340         PayEmployee( m_employee[ id ], netPay );
341
342         // add this employee's paycheck to total for accounting
343         totalWithholdings = totalWithholdings + withholding;
344         totalGovernmentRetirement = totalGovernmentRetirement + governmentRetirement;
345         totalRetirement = totalRetirement + companyRetirement;
346     }
347
348     41 SavePayRecords( totalWithholdings, totalGovernmentRetirement, totalRetirement );
```

In this example, you'll need one initial test case plus one for each of the five keywords, for a total of six. That doesn't mean that any six test cases will cover all the bases. It means that, at a minimum, six cases are required. Unless the cases are constructed carefully, they almost surely won't cover all the bases. The trick is to pay attention to the same keywords you used when counting the number of cases needed. Each keyword in the code represents something that can be either true or false; make sure you have at least one test case for each true and at least one for each false.

352 Here is a set of test cases that covers all the bases in this example:

Case	Test Description	Test Data
1	Nominal case	All boolean conditions are true
2	The initial <i>for</i> condition is false	$numEmployees < 1$
3	The first <i>if</i> is false	$m_employee[id].governmentRetirementWithheld \geq MAX_GOV_T_RETIREMENT$
4	The second <i>if</i> is false because the first part of the <i>and</i> is false	$not m_employee[id].WantsRetirement$
5	The second <i>if</i> is false because the second part of the <i>and</i> is false	$not EligibleForRetirement(m_employee[id])$
6	The third <i>if</i> is false	$not EligibleForPersonalRetirement(m_employee[id])$

353

Note: This table will be extended with additional test cases throughout the chapter.

354

If the routine were much more complicated than this, the number of test cases you'd have to use just to cover all the paths would increase pretty quickly.

355

Shorter routines tend to have fewer paths to test. Boolean expressions without a lot of *ands* and *ors* have fewer variations to test. Ease of testing is another good reason to keep your routines short and your boolean expressions simple.

356

Now that you've created six test cases for the routine and satisfied the demands of structured basis testing, can you consider the routine to be fully tested? Probably not. This kind of testing assures you only that all of the code will be executed. It does not account for variations in data.

363

Data-Flow Testing

364

Viewing the last subsection and this one together gives you another example illustrating that control flow and data flow are equally important in computer programming.

365

366

Data-flow testing is based on the idea that data usage is at least as error-prone as control flow. Boris Beizer claims that at least half of all code consists of data declarations and initializations (Beizer 1990).

367

368

369

Data can exist in one of three states:

370

Defined

The data has been initialized, but it hasn't been used yet.

371

372

Used

The data has been used for computation, as an argument to a routine, or for something else.

Killed

The data was once defined, but it has been undefined in some way. For example, if the data is a pointer, perhaps the pointer has been freed. If it's a *for*-loop index, perhaps the program is out of the loop and the programming language doesn't define the value of a *for*-loop index once it's outside the loop. If it's a pointer to a record in a file, maybe the file has been closed and the record pointer is no longer valid.

In addition to having the terms "defined," "used," and "killed," it's convenient to have terms that describe entering or exiting a routine immediately before or after doing something to a variable:

Entered

The control flow enters the routine immediately before the variable is acted upon. A working variable is initialized at the top of a routine, for example.

Exited

The control flow leaves the routine immediately after the variable is acted upon. A return value is assigned to a status variable at the end of a routine, for example.

Combinations of Data States

The normal combination of data states is that a variable is defined, used one or more times, and perhaps killed. View the following patterns suspiciously:

Defined-Defined

If you have to define a variable twice before the value sticks, you don't need a better program, you need a better computer! It's wasteful and error-prone, even if not actually wrong.

Defined-Exited

If the variable is a local variable, it doesn't make sense to define it and exit without using it. If it's a routine parameter or a global variable, it might be all right.

Defined-Killed

Defining a variable and then killing it suggests either that the variable is extraneous or that the code that was supposed to use the variable is missing.

407
408
409
410
411**Entered-Killed**

This is a problem if the variable is a local variable. It wouldn't need to be killed if it hasn't been defined or used. If, on the other hand, it's a routine parameter or a global variable, this pattern is all right as long as the variable is defined somewhere else before it's killed.

412
413
414
415
416**Entered-Used**

Again, this is a problem if the variable is a local variable. The variable needs to be defined before it's used. If, on the other hand, it's a routine parameter or a global variable, the pattern is all right if the variable is defined somewhere else before it's used.

417
418
419
420
421**Killed-Killed**

A variable shouldn't need to be killed twice. Variables don't come back to life. A resurrected variable indicates sloppy programming. Double kills are also fatal for pointers—one of the best ways to hang your machine is to kill (free) a pointer twice.

422
423
424
425
426**Killed-Used**

Using a variable after it has been killed is a logical error. If the code seems to work anyway (for example, a pointer that still points to memory that's been freed), that's an accident, and Murphy's Law says that the code will stop working at the time when it will cause the most mayhem.

427
428
429
430**Used-Defined**

Using and then defining a variable might or might not be a problem, depending on whether the variable was also defined before it was used. Certainly if you see a used-defined pattern, it's worthwhile to check for a previous definition.

431
432
433
434

Check for these anomalous sequences of data states before testing begins. After you've checked for the anomalous sequences, the key to writing data-flow test cases is to exercise all possible defined-used paths. You can do this to various degrees of thoroughness, including

435
436
437

- All definitions. Test every definition of every variable (that is, every place at which any variable receives a value). This is a weak strategy because if you try to exercise every line of code you'll do this by default.
- All defined-used combinations. Test every combination of defining a variable in one place and using it in another. This is a stronger strategy than testing all definitions because merely executing every line of code does not guarantee that every defined-used combination will be tested.

442

Here's an example:

Java Example of a Program Whose Data Flow Is to Be Tested

```
443  
444 if ( Condition 1 ) {  
445     x = a;  
446 }  
447 else {  
448     x = b;  
449 }  
450  
451     if ( Condition 2 ) {  
452         y = x + 1;  
453     }  
454 else {  
455     y = x - 1;  
456 }
```

To cover every path in the program, you need one test case in which *Condition 1* is true and one in which it's false. You also need a test case in which *Condition 2* is true and one in which it's false. This can be handled by two test cases: Case 1 (*Condition 1=True, Condition 2=True*) and Case 2 (*Condition 1=False, Condition 2=False*). Those two cases are all you need for structured basis testing. They're also all you need to exercise every line of code that defines a variable; they give you the weak form of data-flow testing automatically.

To cover every defined-used combination, however, you need to add a few more cases. Right now you have the cases created by having *Condition 1* and *Condition 2* true at the same time and *Condition 1* and *Condition 2* false at the same time:

```
464  
465     x = a  
466     ...  
467     y = x + 1  
468 and
```

```
472     x = b  
473     ...  
474     y = x - 1
```

But you need two more cases to test every defined-used combination. You need: (1) $x = a$ and then $y = x - 1$ and (2) $x = b$ and then $y = x + 1$. In this example, you can get these combinations by adding two more cases: Case 3 (*Condition 1=True, Condition 2=False*) and Case 4 (*Condition 1=False, Condition 2=True*).

A good way to develop test cases is to start with structured basis testing, which gives you some if not all of the defined-used data flows. Then add the cases you still need to have a complete set of defined-used data-flow test cases.

483 As discussed in the previous subsection, structured basis testing provided six test
484 cases for the routine on page TBD. Data-flow testing of each defined-used pair
485 requires several more test cases, some of which are covered by existing test cases
486 and some of which aren't. Here are all the data-flow combinations that add test
487 cases beyond the ones generated by structured basis testing:

Case	Test Description
7	Define <i>companyRetirement</i> in line 12 and use it first in line 26. This isn't necessarily covered by any of the previous test cases.
8	Define <i>companyRetirement</i> in line 15 and use it first in line 31. This isn't necessarily covered by any of the previous test cases.
9	Define <i>companyRetirement</i> in line 17 and use it first in line 31. This isn't necessarily covered by any of the previous test cases.

488 Once you run through the process of listing data-flow test cases a few times,
489 you'll get a sense of which cases are fruitful and which are already covered.
490 When you get stuck, list all the defined-used combinations. That might seem like
491 a lot of work, but it's guaranteed to show you any cases that you didn't test for
492 free in the basis-testing approach.

493 Equivalence Partitioning

494 **CROSS-REFERENCE** Equivalence partitioning is
495 discussed in far more depth
496 in the books listed in the
497 "Additional Resources"
498 section at the end of this
chapter.

499 In the listing on page TBD, line 7 is a good place to use equivalence partitioning.
500 The condition to be tested is *m_employee[ID].governmentRetirementWithheld*
501 < *MAX_GOV_T_RETIREMENT*. This case has two equivalence classes: the class
502 in which *m_employee[ID].governmentRetirementWithheld* is less than
503 *MAX_GOV_T_RETIREMENT* and the class in which it's greater than or equal to
504 *MAX_GOV_T_RETIREMENT*. Other parts of the program may have other, related
505 equivalence classes that imply that you need to test more than two possible
506 values of *m_employee[ID].governmentRetirementWithheld*, but as far as this
part of the program is concerned, only two are needed.

507 Thinking about equivalence partitioning won't give you a lot of new insight into
508 a program when you have already covered the program with basis and data-flow
509 testing. It's especially helpful, however, when you're looking at a program from
510 the outside (from a specification rather than the source code), or when the data is
511 complicated and the complications aren't all reflected in the program's logic.

512

513 **CROSS-REFERENCE** For
 514 details on heuristics, see
 515 Section 2.2, "How to Use
 516 Software Metaphors."

517

518

519

520

521

522

523

524

525

526

527

528

529

530

531

532

533

534

535

536

537

538

539

540

Error Guessing

In addition to the formal test techniques, good programmers use a variety of less formal, heuristic techniques to expose errors in their code. One heuristic is the technique of error guessing. The term "error guessing" is a lowbrow name for a sensible concept. It means creating test cases based upon guesses about where the program might have errors, although it implies a certain amount of sophistication in the guessing.

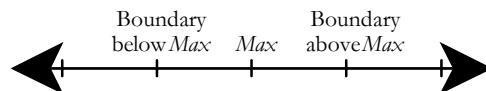
You can base guesses on intuition or on past experience. Chapter 21 points out that one virtue of inspections is that they produce and maintain a list of common errors. The list is used to check new code. When you keep records of the kinds of errors you've made before, you improve the likelihood that your "error guess" will discover an error.

The next few subsections describe specific kinds of errors that lend themselves to error guessing.

Boundary Analysis

One of the most fruitful areas for testing is boundary conditions—off-by-one errors. Saying *num-1* when you mean *num* and saying \geq when you mean $>$ are common mistakes.

The idea of boundary analysis is to write test cases that exercise the boundary conditions. Pictorially, if you're testing for a range of values that are less than *max*, you have three possible conditions:



G22xx02

As shown, there are three boundary cases: just *less than max*, *max* itself, and just greater than *max*. It takes three cases to ensure that none of the common mistakes has been made.

The example on page TBD contains a test for *m_employee[ID].governmentRetirementWithheld > MAX_GOV_T_RETIREMENT*. According to the principles of boundary analysis, three cases should be examined:

Case	Test Description
------	------------------

- 1 Case 1 is defined so that the true boolean condition for *m_employee[ID].governmentRetirementWithheld < MAX_GOV_T_RETIREMENT* is the true side of the boundary. Thus, the Case 1 test case sets *m_employee[ID].governmentRetirementWithheld* to *MAX_GOV_T_RETIREMENT-*

541

542
543
544
545546
547
548
549550
551
552
553
554555
556

3	1. This test case was already generated.
	Case 3 is defined so that the false boolean condition for <i>m_employee[ID].governmentRetirementWithheld < MAX_GOVRETIREMENT</i> is the false side of the boundary. Thus, the Case 3 test case sets <i>m_employee[ID].governmentRetirementWithheld</i> to <i>MAX_GOVRETIREMENT + 1</i> . This test case was also already generated.
10	An additional test case is added for the dead-on case in which <i>m_employee[ID].governmentRetirementWithheld = MAX_GOVRETIREMENT</i> .

Compound Boundaries

Boundary analysis also applies to minimum and maximum allowable values. In this example, it might be minimum or maximum *grossPay*, *companyRetirement*, or *PersonalRetirementContribution*, but since calculations of those values are outside the scope of the routine, test cases for them aren't discussed further here.

A more subtle kind of boundary condition occurs when the boundary involves a combination of variables. For example, if two variables are multiplied together, what happens when both are large positive numbers? Large negative numbers? 0? What if all the strings passed to a routine are uncommonly long?

In the running example, you might want to see what happens to the variables *totalWithholdings*, *totalGovernmentRetirement*, and *totalRetirement* when every member of a large group of employees has a large salary—say, a group of programmers at \$250,000 each. (We can always hope!) This calls for another test case:

Case	Test Description
------	------------------

11	A large group of employees, each of whom has a large salary (what constitutes "large" depends on the specific system being developed), for the sake of example we'll say 1000 employees each with a salary of \$250,000, none of whom have had any social security tax withheld and all of whom want retirement withholding.
----	--

A test case in the same vein but on the opposite side of the looking glass would be a small group of employees, each of whom has a salary of \$0.00.

Case	Test Description
------	------------------

12	A group of 10 employees, each of whom has a salary of \$0.00.
----	---

557

558

559

560

561

562

563

564

565

566

567

568

569

Classes of Bad Data

Aside from guessing that errors show up around boundary conditions, you can guess about and test for several other classes of bad data. Typical bad-data test cases include

- Too little data (or no data)
- Too much data
- The wrong kind of data (invalid data)
- The wrong size of data
- Uninitialized data

Some of the test cases you would think of if you followed these suggestions have already been covered. For example, “too little data” is covered by Cases 2 and 12, and it’s hard to come up with anything for “wrong size of data.” Classes of bad data nonetheless gives rise to a few more cases:

Case	Test Description
13	An array of 100,000,000 employees. Tests for too much data. Of course, how much is too much would vary from system to system, but for the sake of the example assume that this is far too much.
14	A negative salary. Wrong kind of data.
15	A negative number of employees. Wrong kind of data.

Classes of Good Data

When you try to find errors in a program, it’s easy to overlook the fact that the nominal case might contain an error. Usually the nominal cases described in the basis-testing section represent one kind of good data. Here are other kinds of good data that are worth checking:

- Nominal cases—middle-of-the-road, expected values
- Minimum normal configuration
- Maximum normal configuration
- Compatibility with old data

Checking each of these kinds of data can reveal errors, depending on the item being tested.

The minimum normal configuration is useful for testing not just one item, but a group of items. It’s similar in spirit to the boundary condition of many minimal values, but it’s different in that it creates the set of minimum values out of the set

570

571

572

573

574

575

576

577

578

579

580

581

582

583

584 of what is normally expected. One example would be to save an empty
585 spreadsheet when testing a spreadsheet. For testing a word processor, it would be
586 saving an empty document. In the case of the running example, testing the
587 minimum normal configuration would add the following test case:

Case	Test Description
16	A group of one employee. To test the minimum normal configuration.

588 The maximum normal configuration is the opposite of the minimum. It's similar
589 in spirit to boundary testing, but again, it creates a set of maximum values out of
590 the set of expected values. An example of this would be saving a spreadsheet
591 that's as large as the "maximum spreadsheet size" advertised on the product's
592 packaging. Or printing the maximum-size spreadsheet. For a word processor, it
593 would be saving a document of the largest recommended size. In the case of the
594 running example, testing the maximum normal configuration depends on the
595 maximum normal number of employees. Assuming it's 500, you would add the
596 following test case:

Case	Test Description
17	A group of 500 employees. To test the maximum normal configuration.

597 The last kind of normal data testing, testing for compatibility with old data,
598 comes into play when the program or routine is a replacement for an older
599 program or routine. The new routine should produce the same results with old
600 data that the old routine did, except in cases in which the old routine was
601 defective. This kind of continuity between versions is the basis for regression
602 testing, the purpose of which is to ensure that corrections and enhancements
603 maintain previous levels of quality without backsliding. In the case of the
604 running example, the compatibility criterion wouldn't add any test cases.

605 Use Test Cases That Make Hand-Checks 606 Convenient

607 Let's suppose you're writing a test case for a nominal salary; you need a nominal
608 salary, and the way you get one is to type in whatever numbers your hands land
609 on. I'll try it:

610 1239078382346

611 OK. That's a pretty high salary, a little over a trillion dollars, in fact, but if I trim
612 it so that it's somewhat realistic, I get \$90,783.82.

613 Now, further suppose that this test case succeeds, that is, it finds an error. How
614 do you know that it's found an error? Well, presumably, you know what the
615 answer is and what it should be because you calculated the correct answer by

616 hand. When you try to do hand-calculations with an ugly number like
617 \$90,783.82, however, you're as likely to make an error in the hand-calc as you
618 are to discover one in your program. On the other hand, a nice, even number like
619 \$20,000 makes number crunching a snap. The 0s are easy to punch into the
620 calculator, and multiplying by 2 is something most programmers can do without
621 using their fingers and toes.

622 You might think that an ugly number like \$90,783.82 would be more likely to
623 reveal errors, but it's no more likely than any other number in its equivalence
624 class.

625 22.4 Typical Errors

626 This section is dedicated to the proposition that you can test best when you know
627 as much as possible about your enemy: errors.

628 Which Classes Contain the Most Errors?

629 **KEY POINT**
630 It's natural to assume that defects are distributed evenly throughout your source
631 code. If you have an average of 10 defects per 1000 lines of code, you might
632 assume that you'll have 1 defect in a class contains 100 lines of code. This is a
natural assumption, but it's wrong.

633 Capers Jones reported a focused quality-improvement program at IBM identified
634 31 of 425 IMS classes as error prone. The 31 classes were repaired or completely
635 redeveloped, and, in less than a year, customer-reported defects against IMS
636 were reduced ten to one. Total maintenance costs were reduced by about 45%.
637 Customer satisfaction improved from "unacceptable" to "good" (Jones 2000).

638 Most errors tend to be concentrated in a few highly defective routines. Here is
639 the general relationship between errors and code:

- 640 **HARD DATA**
641
- Eighty percent of the errors are found in 20 percent of a project's classes or
routines (Endres 1975, Gremillion 1984, Boehm 1987b, Shull et al 2002).
 - Fifty percent of the errors are found in 5 percent of a project's classes (Jones
2000).

644 These relationships might not seem so important until you recognize a few
645 corollaries.

646 First, 20% of a project's routines contribute 80% of the cost of development
647 (Boehm 1987b). That doesn't necessarily mean that the 20% that cost the most

648

649

650 **HARD DATA**

651

652

653

654

655

656

657

658

659 **CROSS-REFERENCE** Anot
660 her class of routines that tend
661 to contain a lot of errors is
662 the class of overly complex
663 routines. For details on
664 identifying and simplifying
665 routines, see "General
Guidelines for Reducing
Complexity" in Section 19.6.

666

667

668

669

670

671

672 **CROSS-REFERENCE** For
673 a list of all the checklists in
674 the book, see the list of
675 checklists following the table
of contents.

676

677

678

are the same as the 20% with the most defects, but it's pretty doggone suggestive.

Second, regardless of the exact proportion of the cost contributed by highly defective routines, highly defective routines are extremely expensive. In a classic study in the 1960s, IBM performed a study of its OS/360 operating system and found that errors were not distributed evenly across all routines but were concentrated into a few. Those error-prone routines were found to be "the most expensive entities in programming" (Jones 1986a). They contained as many as 50 defects per 1000 lines of code, and fixing them often cost 10 times what it took to develop the whole system. (The costs included customer support and in-the-field maintenance.)

Third, the implication of expensive routines for development is clear. As the old expression goes, "time is money." The corollary is that "money is time," and if you can cut close to 80% of the cost by avoiding troublesome routines, you can cut a substantial amount of the schedule as well. This is a clear illustration of the General Principle of Software Quality, that improving quality improves the development schedule.

Fourth, the implication of avoiding troublesome routines for maintenance is equally clear. Maintenance activities should be focused on identifying, redesigning, and rewriting from the ground up those routines that have been identified as error-prone. In the IMS project mentioned above, productivity of IMS releases improved about 15% after removal of the error-prone classes (Jones 2000).

Errors by Classification

Several researchers have tried to classify errors by type and determine the extent to which each kind of error occurs. Every programmer has a list of errors that have been particularly troublesome: off-by-one errors, forgetting to reinitialize a loop variable, and so on. The checklists presented throughout the book provide more details.

Boris Beizer combined data from several studies, arriving at an exceptionally detailed error taxonomy (Beizer 1990). Following is a summary of his results:

- 25.18% Structural
- 22.44% Data
- 16.19% Functionality as implemented
- 9.88% Construction
- 8.98% Integration

- 8.12% Functional requirements
2.76% Test definition or execution
1.74% System, software architecture
4.71% Unspecified

679
680
681
682
683

Beizer reported his results to a precise two decimal places, but the research into
error types has generally been inconclusive. Different studies report wildly
different kinds of errors, and studies that report on similar kinds of errors arrive
at wildly different results, results that differ by 50% rather than by hundredths of
a percentage point.

684
685
686
687

Given the wide variations in reports, combining results from multiple studies as
Beizer has done probably doesn't produce meaningful data. But even if the data
isn't conclusive, some of it is suggestive. Here are some of the suggestions that
can be derived from it:

The scope of most errors is fairly limited

688 | **HARD DATA**
689
690

One study found that 85% of errors could be corrected without modifying more
than one routine (Endres 1975).

Many errors are outside the domain of construction

691
692
693
694
695

Researchers conducting a series of 97 interviews found that the three most
common sources of errors were thin application-domain knowledge, fluctuating
and conflicting requirements, and communication and coordination breakdown
(Curtis, Krasner, and Iscoe 1988).

Most construction errors are the programmers' fault

696 | **If you see hoof prints,
697 think horses—not zebras.
698 The OS is probably not
699 broken. And the database
700 is probably just fine.
701 —Andy Hunt and Dave
702 Thomas**
703

A pair of studies performed many years ago found that, of total errors reported,
roughly 95% are caused by programmers, 2% by systems software (the compiler
and the operating system), 2% by some other software, and 1% by the hardware
(Brown and Sampson 1973, Ostrand and Weyuker 1984). Systems software and
development tools are used by many more people today than they were in the
1970s and 1980s, and so my best guess is that, today, an even higher percentage
of errors are the programmer's fault.

Clerical errors (typos) are a surprisingly common source of problems

704 | **HARD DATA**
705
706
707
708
709
710
711
712

One study found that 36% of all construction errors were clerical mistakes
(Weiss 1975). A 1987 study of almost 3 million lines of flight-dynamics
software found that 18% of all errors were clerical (Card 1987). Another study
found that 4% of all errors were spelling errors in messages (Endres 1975). In
one of my programs, a colleague found several spelling errors simply by running
all the strings from the executable file through a spelling checker. Attention to
detail counts. If you doubt that, consider that three of the most expensive
software errors of all time cost \$1.6 billion, \$900 million, and \$245 million.

713 Each one involved the change of a *single character* in a previously correct
714 program (Weinberg 1983).

715 ***Misunderstanding the design is a recurring theme in studies of***
716 ***programmer errors***

717 Beizer's compilation study, for what it's worth, found that 16.19% of the errors
718 grew out of misinterpretations of the design (Beizer 1990). Another study found
719 that 19% of the errors resulted from misunderstood design (Weiss 1975). It's
720 worthwhile to take the time you need to understand the design thoroughly. Such
721 time doesn't produce immediate dividends (you don't necessarily look like
722 you're working), but it pays off over the life of the project.

723 ***Most errors are easy to fix***

724 About 85% of errors can be fixed in less than a few hours. About 15% can be
725 fixed in a few hours to a few days. And about 1% take longer (Weiss 1975,
726 Ostrand and Weyuker 1984). This result is supported by Barry Boehm's
727 observation that about 20% of the errors take about 80% of the resources to fix
728 (Boehm 1987b). Avoid as many of the hard errors as you can by doing
729 requirements and design reviews upstream. Handle the numerous small errors as
730 efficiently as you can.

731 ***It's a good idea to measure your own organization's experiences with***
732 ***errors***

733 The diversity of results cited in this section indicates that people in different
734 organizations have tremendously different experiences. That makes it hard to
735 apply other organizations' experiences to yours. Some results go against
736 common intuition; you might need to supplement your intuition with other tools.
737 A good first step is to start measuring your process so that you know where the
738 problems are.

739 **Proportion of Errors Resulting from Faulty
740 Construction**

741 If the data that classifies errors is inconclusive, so is much of the data that
742 attributes errors to the various development activities. One certainty is that
743 construction always results in a significant number of errors. Sometimes people
744 argue that the errors caused by construction are cheaper to fix than the errors
745 caused by requirements or design. Fixing individual construction errors might be
746 cheaper, but the evidence doesn't support such a claim about the total cost.

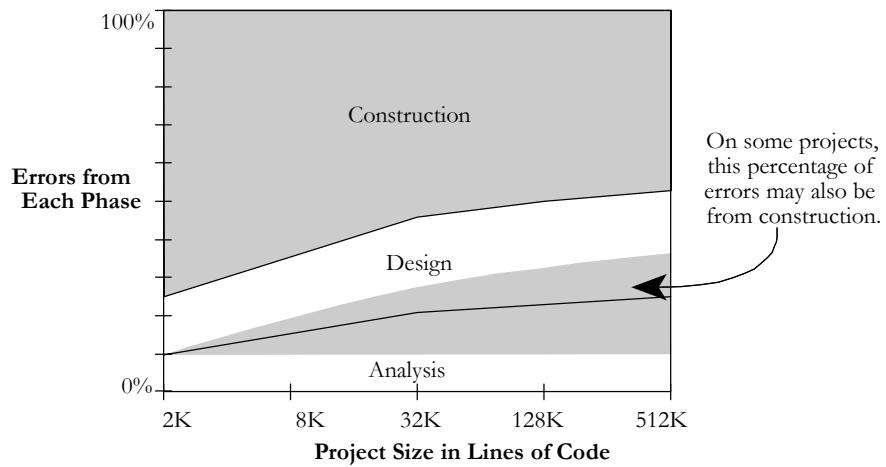
747 Here are my conclusions:

- 748 **HARD DATA**
 - On small projects, construction defects make up the vast bulk of all errors. In
one study of coding errors on a small project (1000 lines of code), 75% of

750 defects resulted from coding, compared to 10% from requirements and 15%
751 from design (Jones 1986a). This error breakdown appears to be
752 representative of many small projects.

- 753 • Construction defects account for at least 35% of all defects. Although the
754 proportion of construction defects is smaller on large projects, they still
755 account for at least 35% of all defects (Beizer 1990, Jones 2000). Some
756 researchers have reported proportions in the 75% range even on very large
757 projects (Grady 1987). In general, the better the application area is
758 understood, the better the overall architecture is. Errors then tend to be
759 concentrated in detailed design and coding (Basili and Perricone 1984).
- 760 • Construction errors, though cheaper to fix than requirements and design
761 errors, are still expensive. A study of two very large projects at Hewlett-
762 Packard found that the average construction defect cost 25 to 50% as much
763 to fix as the average design error (Grady 1987). When the greater number of
764 construction defects was figured into the overall equation, the total cost to
765 fix construction defects was one to two times as much as the cost attributed
766 to design defects.

767 Figure 22-2 provides a rough idea of the relationship between project size and
768 the source of errors.



F22xx02

Figure 22-2

769 *As the size of the project increases, the proportion of errors committed during
770 construction decreases. Nevertheless, construction errors account for 45-75% of all
771 errors on even the largest projects.*

775

776

777

778 **HARD DATA**

779

780

781

782

783

784

785

786

787

788

789 **HARD DATA**

790

791

792

793

794

795

796

797

798

799

800

801

802

803

804

805

806

807

808

809

810 **KEY POINT**

811

812

How Many Errors Should You Expect to Find?

The number of errors you should expect to find varies according to the quality of the development process you use. Here's the range of possibility:

- Industry average experience is about 1 to 25 errors per 1000 lines of code for delivered software. The software has usually been developed using a hodgepodge of techniques (Boehm 1981, Gremillion 1984, Yourdon 1989a, Jones 1998, Jones 2000, Weber 2003). Cases that have one-tenth as many errors as this are rare; cases that have 10 times more tend not to be reported. (They probably aren't ever completed!)
- The Applications Division at Microsoft experiences about 10 to 20 defects per 1000 lines of code during in-house testing, and 0.5 defect per 1000 lines of code in released product (Moore 1992). The technique used to achieve this level is a combination of the code-reading techniques described in Section 21.4 and independent testing.
- Harlan Mills pioneered "cleanroom development," a technique that has been able to achieve rates as low as 3 defects per 1000 lines of code during in-house testing, and 0.1 defect per 1000 lines of code in released product (Cobb and Mills 1990). A few projects—for example, the space-shuttle software—have achieved a level of 0 defects in 500,000 lines of code using a system of formal development methods, peer reviews, and statistical testing (Fishman 1996).
- Watts Humphrey reports that teams using the Team Software Process (TSP) have achieved defect levels of about 0.06 defects per 1000 lines of code. TSP focuses on training developers not to create defects in the first place (Weber 2003).

The results of the TSP and cleanroom projects confirm the General Principle of Software Quality: It's cheaper to build high-quality software than it is to build and fix low-quality software. Productivity for a fully checked-out, 80,000-line clean-room project was 740 lines of code per work-month. The industry average rate for fully checked out code, is closer to 250-300 lines per work-month, including all non-coding overhead (Cusumano et al 2003). The cost savings and productivity come from the fact that virtually no time is devoted to debugging on TSP or cleanroom projects. No time spent on debugging? That is truly a worthy goal!

Errors in Testing Itself

You may have had an experience like this: The software is found to be in error. You have a few immediate hunches about which part of the code might be wrong, but all that code seems to be correct. You run several more test cases to

813
814
815
816
817

try to refine the error, but all the new test cases produce correct results. You spend several hours reading and rereading the code and hand-calculating the results. They all check out. After a few more hours, something causes you to re-examine the test data. Eureka! The error's in the test data! How idiotic it feels to waste hours tracking down an error in the test data rather than in the code!

818 **HARD DATA**

819
820
821
822
823

This is a common experience. Test cases are often as likely or more likely to contain errors than the code being tested (Weiland 1983, Jones 1986a, Johnson 1994). The reasons are easy to find—especially when the developer writes the test cases. Test cases tend to be created on the fly rather than through a careful design and construction process. They are often viewed as one-time tests and are developed with the care commensurate with something to be thrown away.

824

You can do several things to reduce the number of errors in your test cases:

825
826
827
828
829

Check your work

Develop test cases as carefully as you develop code. Such care certainly includes double-checking your own testing. Step through test code in a debugger, line by line, just as you would production code. Walkthroughs and inspections of test data are appropriate.

830
831
832
833

Plan test cases as you develop your software

Effective planning for testing should start at the requirements stage or as soon as you get the assignment for the program. This helps to avoid test cases that are based on mistaken assumptions.

834
835
836
837

Keep your test cases

Spend a little quality time with your test cases. Save them for regression testing and for work on version 2. It's easy to justify the trouble if you know you're going to keep them rather than throw them away.

838
839
840
841

Plug unit tests into a test framework

Write code for unit tests first, but integrate them into a system-wide test framework (like JUnit) as you complete each test. Having an integrated test framework prevents the tendency to throw away test cases mentioned above.

842

22.5 Test-Support Tools

843
844
845
846

This section surveys the kinds of testing tools you can buy commercially or build yourself. It won't name specific products because they could easily be out of date by the time you read this. Refer to your favorite programmer's magazine for the most recent specifics.

847

848

849

850

851

852 **FURTHER READING** For
853 several good examples of
854 scaffolding, see Jon Bentley's
855 essay "A Small Matter of
856 Programming" in
857 *Programming Pearls*, 2d. Ed.
858 (2000).

859

860

861

862

863

864

865

866

867

868

869

870

871

872

873

874

875 **CROSS-REFERENCE** The
876 line between testing tools and
877 debugging tools is fuzzy. For
878 details on debugging tools,
879 see Section 23.5, "Debugging
880 Tools—Obvious and Not-So-
881 Obvious."

Building Scaffolding to Test Individual Classes

The term "scaffolding" comes from building construction. Scaffolding is built so that workers can reach parts of a building they couldn't reach otherwise. Software scaffolding is built for the sole purpose of making it easy to exercise code.

One kind of scaffolding is a class that's dummed up so that it can be used by another class that's being tested. Such a class is called a "mock object" or "stub object" (Mackinnon, Freedman, and Craig 2000; Thomas and Hunt 2002). A similar approach can be used with low-level routines, which are called "stub routines." You can make a mock object or stub routines more or less realistic, depending on how much veracity you need. It can

- Return control immediately, having taken no action
- Test the data fed to it
- Print a diagnostic message, perhaps an echo of the input parameters, or log a message to a file
- Get return values from interactive input
- Return a standard answer regardless of the input
- Burn up the number of clock cycles allocated to the real object or routine
- Function as a slow, fat, simple, or less accurate version of the real object or routine.

Another kind of scaffolding is a fake routine that calls the real routine being tested. This is called a "driver" or, sometimes, a "test harness." It can

- Call the object with a fixed set of inputs
- Prompt for input interactively and call the object with it
- Take arguments from the command line (in operating systems that support it) and call the object
- Read arguments from a file and call the object
- Run through predefined sets of input data in multiple calls to the object

A final kind of scaffolding is the dummy file, a small version of the real thing that has the same types of components that a full-size file has. A small dummy file offers a couple of advantages. Since it's small, you can know its exact contents and can be reasonably sure that the file itself is error-free. And since you create it specifically for testing, you can design its contents so that any error in using it is conspicuous.

881 CC2E.COM/2268

882
883
884
885
886
887
888
889
890

Obviously, building scaffolding requires some work, but if an error is ever detected in a class, you can reuse the scaffolding. And numerous tools exist to streamline creation of mock objects and other scaffolding. If you use scaffolding, the class can also be tested without the risk of its being affected by interactions with other classes. Scaffolding is particularly useful when subtle algorithms are involved. It's easy to get stuck in a rut in which it takes several minutes to execute each test case because the code being exercised is embedded in other code. Scaffolding allows you to exercise the code directly. The few minutes that you spend building scaffolding to exercise the deeply buried code can save hours of debugging time.

891
892
893
894
895
896
897
898
899
900
901
902
903
904

You can use any of the numerous test frameworks available to provide scaffolding for your programs (JUnit, CppUnit, and so on). If your environment isn't supported by one of the existing test frameworks, you can write a few routines in a class and include a *main()* scaffolding routine in the file to test the class, even though the routines being tested aren't intended to stand by themselves. The *main()* routine can read arguments from the command line and pass them to the routine being tested so that you can exercise the routine on its own before integrating it with the rest of the program. When you integrate the code, leave the routines and the scaffolding code that exercises them in the file and use preprocessor commands or comments to deactivate the scaffolding code. Since it's preprocessed out, it doesn't affect the executable code, and since it's at the bottom of the file, it's not in the way visually. No harm is done by leaving it in. It's there if you need it again, and it doesn't burn up the time it would take to remove and archive it.

905

906 **CROSS-REFERENCE** For
907 details on regression testing,
908 see "Retesting (Regression
909 Testing)" in Section 22.6.

910

911

912

913 CC2E.COM/2275

914

915

916

917

918

919

Diff Tools

Regression testing, or retesting, is a lot easier if you have automated tools to check the actual output against the expected output. One easy way to check printed output is to redirect the output to a file and use a file-comparison tool such as Diff to compare the new output against the expected output that was sent to a file previously. If the outputs aren't the same, you have detected a regression error.

Test-Data Generators

You can also write code to exercise selected pieces of a program systematically. A few years ago, I developed a proprietary encryption algorithm and wrote a file-encryption program to use it. The intent of the program was to encode a file so that it could be decoded only with the right password. The encryption didn't just change the file superficially; it altered the entire contents. It was critical that the program be able to decode a file properly, since the file would be ruined otherwise.

920 I set up a test-data generator that fully exercised the encryption and decryption
921 parts of the program. It generated files of random characters in random sizes,
922 from 0K through 500K. It generated passwords of random characters in random
923 lengths from 1 through 255. For each random case, it generated two copies of the
924 random file; encrypted one copy; reinitialized itself; decrypted the copy; and
925 then compared each byte in the decrypted copy to the unaltered copy. If any
926 bytes were different, the generator printed all the information I needed to
927 reproduce the error.

928 I weighted the test cases toward the average length of my files, 30K, which was
929 considerably shorter than the maximum length of 500K. If I had not weighted the
930 test cases toward a shorter length, file lengths would have been uniformly
931 distributed between 0K and 500K. The average tested file length would have
932 been 250K. The shorter average length meant that I could test more files,
933 passwords, end-of-file conditions, odd file lengths, and other circumstances that
934 might produce errors than I could have with uniformly random lengths.

935 The results were gratifying. After running only about 100 test cases, I found two
936 errors in the program. Both arose from special cases that might never have
937 shown up in practice, but they were errors nonetheless, and I was glad to find
938 them. After fixing them, I ran the program for weeks, encrypting and decrypting
939 over 100,000 files without an error. Given the range in file contents, lengths, and
940 passwords I tested, I could confidently assert that the program was correct.

941 Here are the lessons from this story:

- 942 • Properly designed random-data generators can generate unusual
943 combinations of test data that you wouldn't think of.
- 944 • Random-data generators can exercise your program more thoroughly than
945 you can.
- 946 • You can refine randomly generated test cases over time so that they
947 emphasize a realistic range of input. This concentrates testing in the areas
948 most likely to be exercised by users, maximizing reliability in those areas.
- 949 • Modular design pays off during testing. I was able to pull out the encryption
950 and decryption code and use it independently of the user-interface code,
951 making the job of writing a test driver straightforward.
- 952 • You can reuse a test driver if the code it tests ever has to be changed. Once I
953 had corrected the two early errors, I was able to start retesting immediately.

954

Coverage Monitors

AARDOMAR282

955 Karl Wiegers reports that testing done without measuring code coverage
956 typically exercises only about 50-60% of the code (Wiegers 2002). A coverage
957 monitor is a tool that keeps track of the code that's exercised and the code that
958 isn't. A coverage monitor is especially useful for systematic testing because it
959 tells you whether a set of test cases fully exercises the code. If you run your full
960 set of test cases and the coverage monitor indicates that some code still hasn't
961 been executed, you know that you need more tests.

962 Data Recorder

963 Some tools can monitor your program and collect information on the program's
964 state in the event of a failure—similar to the “black box” that airplanes use to
965 diagnose crash results. You can build your own data recorder by logging
966 significant events to a file. This functionality can be compiled in to the
967 development version of the code and compiled out of the released version.

968 Symbolic Debuggers

969 **CROSS-REFERENCE** The
970 availability of debuggers
971 varies according to the
972 maturity of the technology
973 environment. For more on
974 this phenomenon, see Section
4.3, “Your Location on the
Technology Wave.”

975

976

977

978

979

980 Walking through code in a debugger is in many respects the same process as
981 having other programmers step through your code in a review. Neither your
982 peers nor the debugger has the same blind spots that you do. The additional
983 benefit with a debugger is that it's less labor-intensive than a team review.
984 Watching your code execute under a variety of input-data sets is good assurance
985 that you've implemented the code you intended to.

986

987

988

989 System Perturbers

990 Another class of test-support tools are designed to perturb a system. Many
991 people have stories of programs that work 99 times out of 100 but fail on the
992 hundredth run-through with the same data. The problem is nearly always a
993 failure to initialize a variable somewhere, and it's usually hard to reproduce
994 because 99 times out of 100 the uninitialized variable happens to be 0.

995 This class includes tools that have a variety of capabilities:

- 996 • Memory filling. You want to be sure you don't have any uninitialized
997 variables. Some tools fill memory with arbitrary values before you run your
998 program so that uninitialized variables aren't set to 0 accidentally. In some
999 cases, the memory may be set to a specific value. For example, on the x86
1000 processor, the value 0xCC is the machine-language code for a breakpoint
1001 interrupt. If you fill memory with 0xCC and have an error that causes you to
1002 execute something you shouldn't, you'll hit a breakpoint in the debugger and
1003 detect the error.
- 1004 • Memory shaking. In multi-tasking systems, some tools can rearrange
1005 memory as your program operates so that you can be sure you haven't
1006 written any code that depends on data being in absolute rather than relative
1007 locations.
- 1008 • Selective memory failing. A memory driver can simulate low-memory
1009 conditions in which a program might be running out of memory, fail on a
1010 memory request, grant an arbitrary number of memory requests before
1011 failing, or fail on an arbitrary number of requests before granting one. This is
1012 especially useful for testing complicated programs that work with
1013 dynamically allocated memory.
- 1014 • Memory-access checking (bounds checking). Bounds checkers watch
1015 pointer operations to make sure your pointers behave themselves. Such a
1016 tool is useful for detecting uninitialized or dangling pointers.

1017 Error Databases

1018 CC2E.COM/2296 One powerful test tool is a database of errors that have been reported. Such a
1019 database is both a management and a technical tool. It allows you to check for
1020 recurring errors, track the rate at which new errors are being detected and
1021 corrected, and track the status of open and closed errors and their severity. For
1022 details on what information you should keep in an error database, see Section
1023 22.7, "Keeping Test Records."

22.6 Improving Your Testing

The steps for improving your testing are similar to the steps for improving any other process. You have to know exactly what the process does so that you can vary it slightly and observe the effects of the variation. When you observe a change that has a positive effect, you modify the process so that it becomes a little better. The following subsections describe how to do this with testing.

Planning to Test

One key to effective testing is planning from the beginning of the project to test. Putting testing on the same level of importance as design or coding means that time will be allocated to it, it will be viewed as important, and it will be a high-quality process. Test planning is also an element of making the testing process *repeatable*. If you can't repeat it, you can't improve it.

Retesting (Regression Testing)

Suppose that you've tested a product thoroughly and found no errors. Suppose that the product is then changed in one area and you want to be sure that it still passes all the tests it did before the change—that the change didn't introduce any new defects. Testing designed to make sure the software hasn't taken a step backwards, or "regressed," is called "regression testing."

One survey of data-processing personnel found that 52% of those surveyed weren't familiar with this concept (Beck and Perkins 1983). That's unfortunate because it's nearly impossible to produce a high-quality software product unless you can systematically retest it after changes have been made. If you run different tests after each change, you have no way of knowing for sure that no new defects have been introduced. Consequently, regression testing must run the same tests each time. Sometimes new tests are added as the product matures, but the old tests are kept too.

Automated Testing

The only practical way to manage regression testing is to automate it. People become numbed from running the same tests many times and seeing the same test results many times. It becomes too easy to overlook errors, which defeats the purpose of regression testing. Test guru Boriz Beizer reports that the error rate in manual testing is comparable to the bug rate in the code being tested. He estimates that in manual testing, only about half of all the tests are executed properly (Johnson 1994).

Here are some of the benefits of test automation:

- 1059 ● An automated test has a lower chance of being wrong than a manual test.
1060 ● Once you automate a test, it's readily available for the rest of the project
1061 with little incremental effort on your part.
1062 ● If tests are automated, they can be run frequently to see whether any code
1063 check-ins have broken the code. Test automation is part of the foundation of
1064 test-intensive practices like the daily build and smoke test and extreme
1065 programming.
1066 ● Automated tests improve your chances of detecting any given problem at the
1067 earliest possible moment, which tends to minimize the work needed to
1068 diagnose and correct the problem.
1069 ● Automated tests are especially useful in new, volatile technology
1070 environments because they flush out changes in the environments sooner
1071 rather than later.

1072 The main tools used to support automatic testing provide test scaffolding,
1073 generate input, capture output, and compare actual output with expected output.
1074 The variety of tools discussed in the preceding section will perform some or all
1075 of these functions.

1076 22.7 Keeping Test Records

1077 KEY POINT
1078 Aside from making the testing process repeatable, you need to measure the
1079 project so that you can tell for sure whether changes improve or damage it. Here
 are a few kinds of data you can collect to measure your project:

- 1080 ● Administrative description of the defect (the date reported, the person who
1081 reported it, a title or description, the date fixed)
1082 ● Full description of the problem
1083 ● Steps to take to repeat the problem
1084 ● Suggested workaround for the problem
1085 ● Related defects
1086 ● Severity of the problem—for example, fatal, bothersome, or cosmetic
1087 ● Origin of the defect—requirements, design, coding, or testing
1088 ● Subclassification of a coding defect—off-by-one, bad assignment, bad array
1089 index, bad routine call, and so on
1090 ● Location of the fix for the defect
1091 ● Classes and routines changed by the fix

- 1092 • Person responsible for the defect (this can be controversial and might be bad
1093 for morale)
- 1094 • Lines of code affected by the defect
- 1095 • Hours to find the defect
- 1096 • Hours to fix the defect

1097 Once you collect the data, you can crunch a few numbers to determine whether
1098 your project is getting sicker or healthier:

- 1099 • Number of defects in each class, sorted from worst class to best
- 1100 • Number of defects in each routine, sorted from worst routine to best
- 1101 • Average number of testing hours per defect found
- 1102 • Average number of defects found per test case
- 1103 • Average number of programming hours per defect fixed
- 1104 • Percentage of code covered by test cases
- 1105 • Number of outstanding defects in each severity classification

1106 Personal Test Records

1107 In addition to project-level test records, you might find it useful to keep track of
1108 your personal test records. These records can include both a checklist of the
1109 errors you most commonly make as well as a record of the amount of time you
1110 spend writing code, testing code, and correcting errors.

CC2E.COM/2203

1111 Additional Resources

1112 Federal truth-in-advising statutes compel me to disclose that several other books
1113 cover testing in more depth than this chapter does. Books that are devoted to
1114 testing discuss system and black box testing, which haven't been discussed in
1115 this chapter. They also go into more depth on developer topics. They discuss
1116 formal approaches such as cause-effect graphing and the ins and outs of
1117 establishing an independent test organization.

1118 Testing

1119 Kaner, Cem, Jack Falk, and Hung Q. Nguyen. *Testing Computer Software*, 2d
1120 Ed., New York: John Wiley & Sons, 1999. This is probably the best current book
1121 on software testing. It is most applicable to testing applications that will be
1122 distributed to a widespread customer base, such as high-volume websites and
1123 shrink-wrap applications, but it is also generally useful.

1124 Kaner, Cem, James Bach, and Bret Pettichord. *Lessons Learned in Software*
1125 *Testing*, New York: John Wiley & Sons, 2002. This book is a good supplement
1126 to *Testing Computer Software*, 2d. Ed. It's organized into 11 chapters that
1127 enumerate 250 lessons learned by the authors.

1128 Tamre, Louise. *Introducing Software Testing*, Boston, Mass.: Addison Wesley,
1129 2002. This is an accessible testing book targeted at developers who need to
1130 understand testing. Belying the title, the book goes into some depth on testing
1131 details that are useful even to experienced testers.

1132 Whittaker, James A. "What Is Software Testing? And Why Is It So Hard?" *IEEE*
1133 *Software*, January 2000, pp. 70-79. This article is a good introduction to software
1134 testing issues and explains some of the challenges associated with effectively
1135 testing software.

1136 Myers, Glenford J. *The Art of Software Testing*. New York: John Wiley, 1979.
1137 This is the classic book on software testing and is still in print (though quite
1138 expensive). The contents of the book are straightforward: A Self-Assessment
1139 Test; The Psychology and Economics of Program Testing; Program Inspections,
1140 Walkthroughs, and Reviews; Test-Case Design; Class Testing; Higher-Order
1141 Testing; Debugging; Test Tools and Other Techniques. It's short (177 pages) and
1142 readable. The quiz at the beginning gets you started thinking like a tester and
1143 demonstrates how many ways there are to break a piece of code.

1144 **Test Scaffolding**

1145 Bentley, Jon. "A Small Matter of Programming" in *Programming Pearls*, 2d. Ed.
1146 Boston, Mass.: Addison Wesley, 2000. This essay includes several good
1147 examples of test scaffolding.

1148 Mackinnon, Tim, Steve Freeman, and Philip Craig. "Endo-Testing: Unit Testing
1149 with Mock Objects," *eXtreme Programming and Flexible Processes Software*
1150 *Engineering - XP2000 Conference*, 2000. This is the original paper to discuss
1151 the use of mock objects to support developer testing.

1152 Thomas, Dave and Andy Hunt. "Mock Objects," *IEEE Software*, May/June
1153 2002. This is a highly readable introduction to using mock objects to support
1154 developer testing.

1155 **Test First Development**

1156 Beck, Kent. *Test Driven Development*, Boston, Mass.: Addison Wesley, 2003.
1157 Beck describes the ins and outs of "test driven development," a development
1158 approach that's characterized by writing test cases first, then writing the code to
1159 satisfy the test cases. Despite Beck's sometimes-evangelical tone, the advice is

1160
1161 sound, and the book is short and to the point. The book has an extensive running example with real code.

1162 **Relevant Standards**

1163 *IEEE Std 1008-1987 (R1993), Standard for Software Unit Testing*

1164 *IEEE Std 829-1998, Standard for Software Test Documentation*

1165 *IEEE Std 730-2002, Standard for Software Quality Assurance Plans*

CC2E.COM/2210

1166

CHECKLIST: Test Cases

- 1167 Does each requirement that applies to the class or routine have its own test case?
- 1168 Does each element from the design that applies to the class or routine have its own test case?
- 1169 Has each line of code been tested with at least one test case? Has this been verified by computing the minimum number of tests necessary to exercise each line of code?
- 1170 Have all defined-used data-flow paths been tested with at least one test case?
- 1171 Has the code been checked for data-flow patterns that are unlikely to be correct, such as defined-defined, defined-exited, and defined-killed?
- 1172 Has a list of common errors been used to write test cases to detect errors that have occurred frequently in the past?
- 1173 Have all simple boundaries been tested—maximum, minimum, and off-by-one boundaries?
- 1174 Have compound boundaries been tested—that is, combinations of input data that might result in a computed variable that's too small or too large?
- 1175 Do test cases check for the wrong kind of data—for example, a negative number of employees in a payroll program?
- 1176 Are representative, middle-of-the-road values tested?
- 1177 Is the minimum normal configuration tested?
- 1178 Is the maximum normal configuration tested?
- 1179 Is compatibility with old data tested? And are old hardware, old versions of the operating system, and interfaces with old versions of other software tested?
- 1180 Do the test cases make hand-checks easy?
-

1193

Key Points

1194

- Testing by the developer is a key part of a full testing strategy. Independent testing is also important but is outside the scope of this book.

1195

- Writing test cases before the code takes the same amount of time and effort as writing the test cases after the code, but it shortens defect-detection-debug-correction cycles.

1199

- Even considering the numerous kinds of testing available, testing is only one part of a good software-quality program. High-quality development methods, including minimizing defects in requirements and design, are at least as important. Collaborative development practices are also at least as effective at detecting errors as testing and detect different kinds of errors.

1200

- You can generate many test cases deterministically using basis testing, data-flow analysis, boundary analysis, classes of bad data, and classes of good data. You can generate additional test cases with error guessing.

1201

- Errors tend to cluster in a few error-prone classes and routines. Find that error-prone code, redesign it, and rewrite it.

1202

- Test data tends to have a higher error density than the code being tested. Because hunting for such errors wastes time without improving the code, test-data errors are more aggravating than programming errors. Avoid them by developing your tests as carefully as your code.

1203

- Automated testing is useful in general and essential for regression testing.

1204

- In the long run, the best way to improve your testing process is to make it regular, measure it, and use what you learn to improve it.

1205

1206

1207

1208

1209

1210

1211

1212

1213

1214

1215

23

Debugging

Contents

- 23.1 Overview of Debugging Issues
- 23.2 Finding a Defect
- 23.3 Fixing a Defect
- 23.4 Psychological Considerations in Debugging
- 23.5 Debugging Tools—Obvious and Not-So-Obvious

Related Topics

The software-quality landscape: Chapter 20

Developer testing: Chapter 22

Refactoring: Chapter 24

DEBUGGING IS THE PROCESS OF IDENTIFYING the root cause of an error and correcting it. It contrasts with testing, which is the process of detecting the error initially. On some projects, debugging occupies as much as 50 percent of the total development time. For many programmers, debugging is the hardest part of programming.

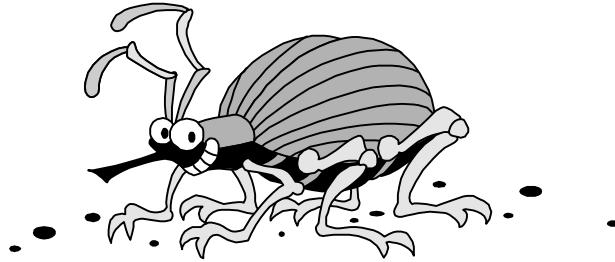
Debugging doesn't have to be the hardest part. If you follow the advice in this book, you'll have fewer errors to debug. Most of the defects you will have will be minor oversights and typos, easily found by looking at a source-code listing or stepping through the code in a debugger. For the remaining harder bugs, this chapter describes how to make debugging much easier than it usually is

23.1 Overview of Debugging Issues

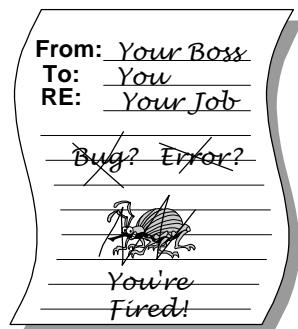
The late Rear Admiral Grace Hopper, co-inventor of COBOL, always said that the word “bug” in software dated back to the first large-scale digital computer, the Mark I (IEEE 1992). Programmers traced a circuit malfunction to the presence of a large moth that had found its way into the computer, and from that time on, computer problems were blamed on “bugs.” Outside software, the word

29 “bug” dates back at least to Thomas Edison, who is quoted as using it as early as
30 1878 (Tenner 1997).

31 The word “bug” is a cute word and conjures up images like this one:



32 **G23xx01**
33 The reality of software defects, however, is that bugs aren’t organisms that sneak
34 into your code when you forget to spray it with pesticide. They are errors. A bug
35 in software means that a programmer made a mistake. The result of the mistake
36 isn’t like the cute picture shown above. It’s more likely a note like this one:
37



38 **G23xx02**
39 In this context, technical accuracy requires that mistakes in the code be called
40 “errors,” “defects,” or “faults.”
41

42

Role of Debugging in Software Quality

43 Like testing, debugging isn’t a way to improve the quality of your software, per
44 se; it’s a way to diagnose defects. Software quality must be built in from the
45 start. The best way to build a quality product is to develop requirements
46 carefully, design well, and use high-quality coding practices. Debugging is a last
47 resort.

48

49

50 KEY POINT

51

52

53

54

55

Variations in Debugging Performance

Why talk about debugging? Doesn't everyone know how to debug?

No, not everyone knows how to debug. Studies of experienced programmers have found roughly a 20-to-1 difference in the time it takes experienced programmers to find the same set of defects. Moreover, some programmers find more defects and make corrections more accurately. Here are the results of a classic study that examined how effectively professional programmers with at least four years of experience debugged a program with 12 defects:

	Fastest Three Programmers	Slowest Three Programmers
Average debug time (minutes)	5.0	14.1
Average number of defects not found	0.7	1.7
Average number of defects made correcting defects	3.0	7.7

Source: "Some Psychological Evidence on How People Debug Computer Programs" (Gould 1975).

58 HARD DATA

59

60

61

62

63

The three programmers who were best at debugging were able to find the defects in about one-third the time and inserted only about two-fifths as many new defects as the three who were the worst. The best programmer found all the defects and didn't insert any new defects in correcting them. The worst missed 4 of the 12 defects and inserted 11 new defects in correcting the 8 defects he found.

64

65

66

67

68

69

70

71

72

73

74

75

76

But, this study doesn't really tell the whole story. After the first round of debugging the fastest three programmers still have 3.7 defects left in their code, and the slowest still have 9.4 defects. Neither group is done debugging yet. I wondered what would happen if I applied the same find-and-bad-fix ratios to additional debugging cycles. This isn't statistically valid, but it's still interesting. When I applied the same find-and-fix ratios to successive debugging cycles until each group had less than half a defect remaining, the fastest group required a total of 3 debugging cycles, whereas the slowest group required 14 debugging cycles. Bearing in mind that each cycle of the slower group takes almost 3 times as long as each cycle of the fastest group, the slowest group would take about 13 times as long to fully debug its programs as the fastest group, according to my non-scientific extrapolation of this study. Interestingly, this wide variation has been confirmed by other studies (Gilb 1977, Curtis 1981).

77 **CROSS-REFERENCE** For
78 details on the relationship
79 between quality and cost, see
80 Section 20.5, “The General
81 Principle of Software
Quality.”

In addition to providing insight into debugging, the evidence supports the General Principle of Software Quality: Improving quality reduces development costs. The best programmers found the most defects, found the defects most quickly, and made correct modifications most often. You don’t have to choose between quality, cost, and time—they all go hand in hand.

Defects as Opportunities

What does having a defect mean? Assuming that you don’t want the program to have defect, it means that you don’t fully understand what the program does. The idea of not understanding what the program does is unsettling. After all, if you created the program, it should do your bidding. If you don’t know exactly what you’re telling the computer to do, that’s only a small step from merely trying different things until something seems to work—that is, programming by trial and error. If you’re programming by trial and error, defects are guaranteed. You don’t need to learn how to fix defects; you need to learn how to avoid them in the first place.

Most people are somewhat fallible, however, and you might be an excellent programmer who has simply made a modest oversight. If this is the case, an error in your program represents a powerful opportunity. You can:

Learn about the program you’re working on

You have something to learn about the program because if you already knew it perfectly, it wouldn’t have a defect. You would have corrected it already.

Learn about the kind of mistakes you make

If you wrote the program, you inserted the defect. It’s not every day that a spotlight exposes a weakness with glaring clarity, but this particular day you have an opportunity to learn about your mistakes. Once you find the mistake, ask why did you make it? How could you have found it more quickly? How could you have prevented it? Does the code have other mistakes just like it? Can you correct them before they cause problems of their own?

Learn about the quality of your code from the point of view of someone who has to read it

You’ll have to read your code to find the defect. This is an opportunity to look critically at the quality of your code. Is it easy to read? How could it be better? Use your discoveries to refactor your current code or to improve the code you write next.

Learn about how you solve problems

Does your approach to solving debugging problems give you confidence? Does your approach work? Do you find defects quickly? Or is your approach to debugging weak? Do you feel anguish and frustration? Do you guess randomly?

115 Do you need to improve? Considering the amount of time many projects spend
116 on debugging, you definitely won't waste time if you observe how you debug.
117 Taking time to analyze and change the way you debug might be the quickest way
118 to decrease the total amount of time it takes you to develop a program.

119 ***Learn about how you fix defects***
120 In addition to learning how you find defects, you can learn about how you fix
121 them. Do you make the easiest possible correction, by applying *goto* Band-Aids
122 and special-case makeup that changes the symptom but not the problem? Or do
123 you make systemic corrections, demanding an accurate diagnosis and prescribing
124 treatment for the heart of the problem?

125 All things considered, debugging is an extraordinarily rich soil in which to plant
126 the seeds of your own improvement. It's where all construction roads cross:
127 readability, design, code quality—you name it. This is where building good code
128 pays off—especially if you do it well enough that you don't have to debug very
129 often.

130 **An Ineffective Approach**

131 Unfortunately, programming classes in colleges and universities hardly ever
132 offer instruction in debugging. If you studied programming in college, you might
133 have had a lecture devoted to debugging. Although my computer-science
134 education was excellent, the extent of the debugging advice I received was to
135 “put print statements in the program to find the defect.” This is not adequate. If
136 other programmers’ educational experiences are like mine, a great many
137 programmers are being forced to reinvent debugging concepts on their own.
138 What a waste!

139 **The Devil’s Guide to Debugging**

140 ***Programmers do not***
141 ***always use available data***
142 ***to constrain their***
143 ***reasoning. They carry out***
144 ***minor and irrational***
145 ***repairs, and they often***
146 ***don’t undo the incorrect***
147 ***repairs.***
148 *—Iris Vessey*

149
150
151

152 ***Don't waste time trying to understand the problem***
153 It's likely that the problem is trivial, and you don't need to understand it
154 completely to fix it. Simply finding it is enough.

155 ***Fix the error with the most obvious fix***
156 It's usually good just to fix the specific problem you see, rather than wasting a
157 lot of time making some big, ambitious correction that's going to affect the
158 whole program. This is a perfect example:

```
159                   x = Compute( y )
160                   if ( y = 17 )
161                    x = $25.15       -- Compute() doesn't work for y = 17, so fix it
162                   Who needs to dig all the way into Compute() for an obscure problem with the
163                   value of 17 when you can just write a special case for it in the obvious place?
```

164 This approach is infinitely extendable. If we later find that *Compute()* returns the
165 wrong value when *y*=18, we just extend our fix:

```
166                   x = Compute( y )
167                   if ( y = 17 )
168                    x = $25.15       -- Compute() doesn't work for y = 17, so fix it
169                   else if ( y = 18 )
170                    x = $27.85       -- Compute() doesn't work for y = 18, so fix it
```

171 **Debugging by Superstition**

172 Satan has leased part of hell to programmers who debug by superstition. Every
173 group has one programmer who has endless problems with demon machines,
174 mysterious compiler defects, hidden language defects that appear when the moon
175 is full, bad data, losing important changes, a vindictive, possessed editor that
176 saves programs incorrectly—you name it. This is “programming by
177 superstition.”

178 If you have a problem with a program you've written, it's your fault. It's not the
179 computer's fault, and it's not the compiler's fault. The program doesn't do
180 something different every time. It didn't write itself; you wrote it, so take
181 responsibility for it.

182 **KEY POINT**

183 Even if an error at first appears not to be your fault, it's strongly in your interest
184 to assume that it is. That assumption helps you debug: It's hard enough to find a
185 defect in your code when you're looking for it; it's even harder when you've
186 assumed your code is error-free. It improves your credibility because when you
187 do claim that an error arose from someone else's code, other programmers will
188 believe that you have checked out the problem carefully. Assuming the error is
 your fault also saves you the embarrassment of claiming that an error is someone

189 else's fault and then having to recant publicly later when you find out that it was
190 your defect after all.

191 23.2 Finding a Defect

192 Debugging consists of finding the defect and fixing it. Finding the defect (and
193 understanding it) is usually 90 percent of the work.

194 Fortunately, you don't have to make a pact with Satan in order to find an
195 approach to debugging that's better than random guessing. Contrary to what the
196 Devil wants you to believe, debugging by thinking about the problem is much
197 more effective and interesting than debugging with an eye of newt and the dust
198 of a frog's ear.

199 Suppose you were asked to solve a murder mystery. Which would be more
200 interesting: going door to door throughout the county, checking every person's
201 alibi for the night of October 17, or finding a few clues and deducing the
202 murderer's identity? Most people would rather deduce the person's identity, and
203 most programmers find the intellectual approach to debugging more satisfying.
204 Even better, the effective programmers who debug in one-twentieth the time of
205 the ineffective programmers aren't randomly guessing about how to fix the
206 program. They're using the scientific method.

207 208 The Scientific Method of Debugging

209 Here are the steps you go through when you use the scientific method:

- 210 211 212 213 214 215 216 217 1. Gather data through repeatable experiments.
2. Form a hypothesis that accounts for the relevant data.
3. Design an experiment to prove or disprove the hypothesis.
4. Prove or disprove the hypothesis.
5. Repeat as needed.

214 **KEY POINT** This process has many parallels in debugging. Here's an effective approach for
215 finding a defect:

1. Stabilize the error.
2. Locate the source of the error (the "fault").

- 218 a. Gather the data that produces the defect.
- 219 b. Analyze the data that has been gathered and form a hypothesis about the
220 defect.
- 221 c. Determine how to prove or disprove the hypothesis, either by testing the
222 program or by examining the code.
- 223 d. Prove or disprove the hypothesis using the procedure identified in 2(c).
- 224 3. Fix the defect.
- 225 4. Test the fix.
- 226 5. Look for similar errors.

227 The first step is similar to the scientific method's first step in that it relies on
228 repeatability. The defect is easier to diagnose if you can make it occur reliably.
229 The second step uses the first four steps of the scientific method. You gather the
230 test data that divulged the defect, analyze the data that has been produced, and
231 form a hypothesis about the source of the error. You design a test case or an
232 inspection to evaluate the hypothesis and then declare success or renew your
233 efforts, as appropriate.

234 Let's look at each of the steps in conjunction with an example.

235 Assume that you have an employee database program that has an intermittent
236 error. The program is supposed to print a list of employees and their income-tax
237 withholdings in alphabetical order. Here's part of the output:

238 Formatting, Fred Freeform \$5,877
239 Goto, Gary \$1,666
240 Modula, Mildred \$10,788
241 Many-Loop, Mavis \$8,889
242 Statement, Sue Switch \$4,000
243 Whileloop, Wendy \$7,860

244 The error is that *Many-Loop*, *Mavis* and *Modula*, *Mildred* are out of order.

245 **Stabilize the Error**

246 If a defect doesn't occur reliably, it's almost impossible to diagnose. Making an
247 intermittent defect occur predictably is one of the most challenging tasks in
248 debugging.

249 **CROSS-REFERENCE** For
250 details on using pointers
251 safely, see Section 13.2,
252 “Pointers.”

253
254
255

An error that doesn't occur predictably is usually an initialization error or a dangling-pointer problem. If the calculation of a sum is right sometimes and wrong sometimes, a variable involved in the calculation probably isn't being initialized properly—most of the time it just happens to start at 0. If the problem is a strange and unpredictable phenomenon and you're using pointers, you almost certainly have an uninitialized pointer or are using a pointer after the memory that it points to has been deallocated.

256
257
258
259
260

Stabilizing an error usually requires more than finding a test case that produces the error. It includes narrowing the test case to the simplest one that still produces the error. If you work in an organization that has an independent test team, sometimes it's the team's job to make the test cases simple. Most of the time, it's your job.

To simplify the test case, you bring the scientific method into play again. Suppose you have 10 factors that, used in combination, produce the error. Form a hypothesis about which factors were irrelevant to producing the error. Change the supposedly irrelevant factors, and rerun the test case. If you still get the error, you can eliminate those factors and you've simplified the test. Then you can try to simplify the test further. If you don't get the error, you've disproved that specific hypothesis, and you know more than you did before. It might be that some subtly different change would still produce the error, but you know at least one specific change that does not.

261
262
263
264
265
266
267
268
269

In the employee withholdings example, when the program is run initially, *Many-Loop, Mavis* is listed after *Modula, Mildred*. When the program is run a second time, however, the list is fine:

270
271
272

273	Formatting, Fred Freeform	\$5,877
274	Goto, Gary	\$1,666
275	Many-Loop, Mavis	\$8,889
276	Modula, Mildred	\$10,788
277	Statement, Sue Switch	\$4,000
278	Whileloop, Wendy	\$7,860

279 It isn't until *Fruit-Loop, Frita* is entered and shows up in an incorrect position
280 that you remember that *Modula, Mildred* had been entered just before she
281 showed up in the wrong spot too. What's odd about both cases is that they were
282 entered singly. Usually, employees are entered in groups.

283 You hypothesize: The problem has something to do with entering a single new
284 employee.

285 If this is true, running the program again should put *Fruit-Loop, Frita* in the right
286 position. Here's the result of a second run:

287 Formatting, Fred Freeform \$5,877
288 Fruit-Loop, Frita \$5,771
289 Goto, Gary \$1,666
290 Many-Loop, Mavis \$8,889
291 Modula, Mildred \$10,788
292 Statement, Sue Switch \$4,000
293 Whileloop, Wendy \$7,860

294 This successful run supports the hypothesis. To confirm it, you want to try
295 adding a few new employees, one at a time, to see whether they show up in the
296 wrong order and whether the order changes on the second run.

297 **Locate the Source of the Error**

298 The goal of simplifying the test case is to make it so simple that changing any
299 aspect of it changes the behavior of the error. Then, by changing the test case
300 carefully and watching the program's behavior under controlled conditions, you
301 can diagnose the problem.

302 Locating the source of the error also calls for using the scientific method. You
303 might suspect that the defect is a result of a specific problem, say an off-by-one
304 error. You could then vary the parameter you suspect is causing the problem—
305 one below the boundary, on the boundary, and one above the boundary—and
306 determine whether your hypothesis is correct.

307 In the running example, the source of the problem could be an off-by-one defect
308 that occurs when you add one new employee but not when you add two or more.
309 Examining the code, you don't find an obvious off-by-one defect. Resorting to
310 Plan B, you run a test case with a single new employee to see whether that's the
311 problem. You add *Hardcase, Henry* as a single employee and hypothesize that
312 his record will be out of order. Here's what you find:

313 Formatting, Fred Freeform \$5,877
314 Fruit-Loop, Frita \$5,771
315 Goto, Gary \$1,666
316 Hardcase, Henry \$493
317 Many-Loop, Mavis \$8,889
318 Modula, Mildred \$10,788
319 Statement, Sue Switch \$4,000
320 Whileloop, Wendy \$7,860

321 The line for *Hardcase, Henry* is exactly where it should be, which means that
322 your first hypothesis is false. The problem isn't caused simply by adding one
323 employee at a time. It's either a more complicated problem or something
324 completely different.

325 Examining the test-run output again, you notice that *Fruit-Loop, Frita* and
326 *Many-Loop, Mavis* are the only names containing hyphens. *Fruit-Loop* was out
327 of order when she was first entered, but *Many-Loop* wasn't, was she? Although

328 you don't have a printout from the original entry, in the original error *Modula*,
329 *Mildred* appeared to be out of order, but she was next to *Many-Loop*. Maybe
330 *Many-Loop* was out of order and *Modula* was all right.

331 You hypothesize: The problem arises from names with hyphens, not names that
332 are entered singly.

333 But how does that account for the fact that the problem shows up only the first
334 time an employee is entered? You look at the code and find that two different
335 sorting routines are used. One is used when an employee is entered, and another
336 is used when the data is saved. A closer look at the routine used when an
337 employee is first entered shows that it isn't supposed to sort the data completely.
338 It only puts the data in approximate order to speed up the save routine's sorting.
339 Thus, the problem is that the data is printed before it's sorted. The problem with
340 hyphenated names arises because the rough-sort routine doesn't handle niceties
341 such as punctuation characters. Now, you can refine the hypothesis even further.

342 You hypothesize: Names with punctuation characters aren't sorted correctly until
343 they're saved.

344 You later confirm this hypothesis with additional test cases.

345 **Tips for Finding Defects**

346 Once you've stabilized an error and refined the test case that produces it, finding
347 its source can be either trivial or challenging, depending on how well you've
348 written your code. If you're having a hard time finding a defect, it could be
349 because the code isn't well written. You might not want to hear that, but it's true.
350 If you're having trouble, consider these tips:

351 ***Use all the data available to make your hypothesis***

352 When creating a hypothesis about the source of a defect, account for as much of
353 the data as you can in your hypothesis. In the example, you might have noticed
354 that *Fruit-Loop*, *Frita* was out of order and created a hypothesis that names
355 beginning with an "F" are sorted incorrectly. That's a poor hypothesis because it
356 doesn't account for the fact that *Modula*, *Mildred* was out of order or that names
357 are sorted correctly the second time around. If the data doesn't fit the hypothesis,
358 don't discard the data—ask why it doesn't fit, and create a new hypothesis.

359 The second hypothesis in the example, that the problem arises from names with
360 hyphens, not names that are entered singly, didn't seem initially to account for
361 the fact that names were sorted correctly the second time around either. In this
362 case, however, the second hypothesis led to a more refined hypothesis that
363 proved to be correct. It's all right that the hypothesis doesn't account for all of

the data at first as long as you keep refining the hypothesis so that it does eventually.

Refine the test cases that produce the error

If you can't find the source of an error, try to refine the test cases further than you already have. You might be able to vary one parameter more than you had assumed, and focusing on one of the parameters might provide the crucial breakthrough.

- 371 **CROSS-REFERENCE** For
- 372 more on unit test
- 373 frameworks, see “Plug unit tests into a test framework”
in Section 22.4.

375

Exercise the code in your unit test suite

Defects tend to be easier to find in small fragments of code than in large integrated programs. Use your unit tests to test the code in isolation.

374 in Section 22.4.
375
376
377
378
379
380
381
382
383

Use available tools

Numerous tools are available to support debugging sessions: interactive debuggers, picky compilers, memory checkers, and so on. The right tool can make a difficult job easy. With one tough-to-find error, for example, one part of the program was overwriting another part's memory. This error was difficult to diagnose using conventional debugging practices because the programmer couldn't determine the specific point at which the program was incorrectly overwriting memory. The programmer used a memory breakpoint to set a watch on a specific memory address. When the program wrote to that memory location, the debugger stopped the code, and the guilty code was exposed.

384

385

This is an example of problem that's difficult to diagnose analytically but which becomes quite simple when the right tool is applied.

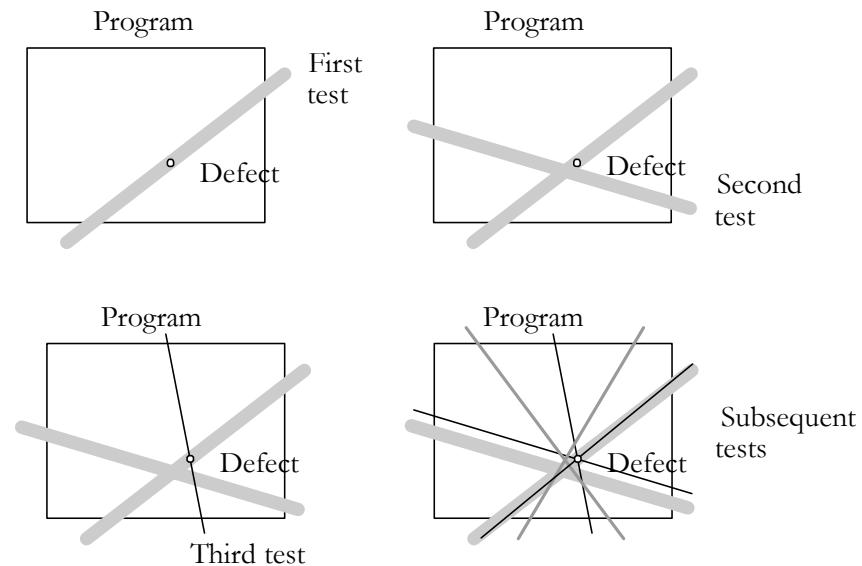
386
387
388
389
390

Reproduce the error several different ways

Sometimes trying cases that are similar to the error-producing case, but not exactly the same, is instructive. Think of this approach as triangulating the defect. If you can get a fix on it from one point and a fix on it from another, you can determine exactly where it is.

391
392
393
394
395
396

Reproducing the error several different ways helps diagnose the cause of the error. Once you think you've identified the defect, run a case that's close to the cases that produce errors but that should not produce an error itself. If it does produce an error, you don't completely understand the problem yet. Errors often arise from combinations of factors, and trying to diagnose the problem with only one test case sometimes doesn't diagnose the root problem.

**F23xx01****Figure 23-1**

Try to reproduce an error several different ways to determine its exact cause.

Generate more data to generate more hypotheses

Choose test cases that are different from the test cases you already know to be erroneous or correct. Run them to generate more data, and use the new data to add to your list of possible hypotheses.

Use the results of negative tests

Suppose you create a hypothesis and run a test case to prove it. Suppose the test case disproves the hypothesis, so that you still don't know the source of the error. You still know something you didn't before—namely, that the defect is not in the area in which you thought it was. That narrows your search field and the set of possible hypotheses.

Brainstorm for possible hypotheses

Rather than limiting yourself to the first hypothesis you think of, try to come up with several. Don't analyze them at first—just come up with as many as you can in a few minutes. Then look at each hypothesis and think about test cases that would prove or disprove it. This mental exercise is helpful in breaking the debugging logjam that results from concentrating too hard on a single line of reasoning.

Narrow the suspicious region of the code

If you've been testing the whole program, or a whole class or routine, test a smaller part instead. Use print statements, logging, or tracing to identify which section of code is producing the error.

397

398

399

400

401

402

403

404

405

406

407

408

409

410

411

412

413

414

415

416

417

418

419

420

421

422 If you need a more powerful technique to narrow the suspicious region of the
423 code, systematically remove parts of the program and see whether the error still
424 occurs. If it doesn't, you know it's in the part you took away. If it does, you
425 know it's in the part you've kept.

426 Rather than removing regions haphazardly, divide and conquer. Use a binary
427 search algorithm to focus your search. Try to remove about half the code the first
428 time. Determine the half the defect is in, and then divide that section. Again,
429 determine which half contains the defect, and again, chop that section in half.
430 Continue until you find the defect.

431 If you use many small routines, you'll be able to chop out sections of code
432 simply by commenting out calls to the routines. Otherwise, you can use
433 comments or preprocessor commands to remove code.

434 If you're using a debugger, you don't necessarily have to remove pieces of code.
435 You can set a breakpoint partway through the program and check for the defect
436 that way instead. If your debugger allows you to skip calls to routines, eliminate
437 suspects by skipping the execution of certain routines and seeing whether the
438 error still occurs. The process with a debugger is otherwise similar to the one in
439 which pieces of a program are physically removed.

440 **CROSS-REFERENCE** For
441 more details on error-prone
442 code, see "Target error-prone
443 modules" in Section 24.6.
444

445 **Check code that's changed recently**
446 If you have a new error that's hard to diagnose, it's usually related to code that's
447 changed recently. It could be in completely new code or in changes to old code.
448 If you can't find a defect, run an old version of the program to see whether the
449 error occurs. If it doesn't, you know the error's in the new version or is caused
450 by an interaction with the new version. Scrutinize the differences between the
451 old and new versions. Check the version control log to see what code has
452 changed recently. If that's not possible, use a diff tool to compare changes in the
453 old, working source code to the new, broken source code.

454 **Expand the suspicious region of the code**
455 It's easy to focus on a small section of code, sure that "the defect *must* be in this
456 section." If you don't find it in the section, consider the possibility that the defect
457 isn't in the section. Expand the area of code you suspect, and then focus on
458 pieces of it using the binary search technique described above.

459 **CROSS-REFERENCE** For
460 a full discussion of
461 integration, see Chapter 29,
“Integration.”

462
463
464
465
466
467

468 **CROSS-REFERENCE** For
469 details on how involving
470 other developers can put a
471 beneficial distance between
you and the problem, see
Section 21.1, “Overview of
472 Collaborative Development
473 Practices.”

474
475
476
477
478
479

480
481

482
483
484
485
486
487
488

489
490
491

492
493
494
495

Integrate incrementally

Debugging is easy if you add pieces to a system one at a time. If you add a piece to a system and encounter a new error, remove the piece and test it separately.

Check for common defects

Use code-quality checklists to stimulate your thinking about possible defects. If you’re following the inspection practices described in Section 21.3, you’ll have your own fine-tuned checklist of the common problems in your environment. You can also use the checklists that appear throughout this book. See the “List of Checklists” following the table of contents.

Talk to someone else about the problem

Some people call this “confessional debugging.” You often discover your own defect in the act of explaining it to another person. For example, if you were explaining the problem in the salary example, you might sound like this:

“Hey, Jennifer, have you got a minute? I’m having a problem. I’ve got this list of employee salaries that’s supposed to be sorted, but some names are out of order. They’re sorted all right the second time I print them out but not the first. I checked to see if it was new names, but I tried some that worked. I know they should be sorted the first time I print them because the program sorts all the names as they’re entered and again when they’re saved—wait a minute—no, it doesn’t sort them when they’re entered. That’s right. It only orders them roughly. Thanks, Jennifer. You’ve been a big help.”

Jennifer didn’t say a word, and you solved your problem. This result is typical, and this approach is perhaps your most potent tool for solving difficult defects.

Take a break from the problem

Sometimes you concentrate so hard you can’t think. How many times have you paused for a cup of coffee and figured out the problem on your way to the coffee machine? Or in the middle of lunch? Or on the way home? Or in the shower the next morning? If you’re debugging and making no progress, once you’ve tried all the options, let it rest. Go for a walk. Work on something else. Go home for the day. Let your subconscious mind tease a solution out of the problem.

The auxiliary benefit of giving up temporarily is that it reduces the anxiety associated with debugging. The onset of anxiety is a clear sign that it’s time to take a break.

Brute Force Debugging

Brute force is an often-overlooked approach to debugging software problems. By “brute force,” I’m referring to a technique that might be tedious, arduous, and time-consuming, but that it is *guaranteed* to solve the problem. Which specific

496 techniques are guaranteed to solve a problem are context dependent, but here are
497 some general candidates:

- 498 • Perform a full design and/or code review on the broken code
- 499 • Throw away the section of code and redesign/recode it from scratch
- 500 • Throw away the whole program and redesign/recode it from scratch
- 501 • Compile code with full debugging information
- 502 • Compile code at pickiest warning level and fix all the picky compiler
503 warnings
- 504 • Strap on a unit test harness and test the new code in isolation
- 505 • Create an automated test suite and run it all night
- 506 • Step through a big loop in the debugger manually until you get to the error
507 condition
- 508 • Instrument the code with print, display, or other logging statements
- 509 • Replicate the end-user's full machine configuration
- 510 • Integrate new code in small pieces, fully testing each piece as its integrated

511 ***Set a maximum time for quick and dirty debugging***

512 For each brute force technique, your reaction might very well be, "I can't do
513 that; it's too much work!" The point is that it's only too much work if it takes
514 more time than what I call "quick and dirty debugging." It's always tempting to
515 try for a quick guess rather than systematically instrumenting the code and
516 giving the defect no place to hide. The gambler in each of us would rather use a
517 risky approach that might find the defect in five minutes than the surefire
518 approach that will find the defect in half an hour. The risk is that, if the five-
519 minute approach doesn't work, you get stubborn. Finding the defect the "easy"
520 way becomes a matter of principle, and hours pass unproductively, as do days,
521 weeks, months, ... How often have you spent two hours debugging code that took
522 only 30 minutes to write? That's a bad distribution of labor, and you would have
523 been better off simply to rewrite the code than to debug bad code.

524 When you decide to go for the quick victory, set a maximum time limit for trying
525 the quick way. If you go past the time limit, resign yourself to the idea that the
526 defect is going to be harder to diagnose than you originally thought, and flush it
527 out the hard way. This approach allows you to get the easy defects right away
528 and the hard defects after a bit longer.

529 ***Make a list of brute force techniques***

530 Before you begin debugging a difficult error, ask yourself, "If I get stuck
531 debugging this problem, is there some way that I am *guaranteed* to be able to fix

532 the problem?" If you can identify at least one brute force technique that will fix
533 the problem—including rewriting the code in question—it's less likely that
534 you'll waste hours or days when there's a quicker alternative.

535 Syntax Errors

536 Syntax-error problems are going the way of the woolly mammoth and the saber-
537 toothed tiger. Compilers are getting better at diagnostic messages, and the days
538 when you had to spend two hours finding a misplaced semicolon in a Pascal
539 listing are almost gone. Here's a list of guidelines you can use to hasten the
540 extinction of this endangered species:

541 ***Don't trust line numbers in compiler messages***

542 When your compiler reports a mysterious syntax error, look immediately before
543 and immediately after the error—the compiler could have misunderstood the
544 problem or simply have poor diagnostics. Once you find the real defect, try to
545 determine the reason the compiler put the message on the wrong statement.
546 Understanding your compiler better can help you find future defects.

547 ***Don't trust compiler messages***

548 Compilers try to tell you exactly what's wrong, but compilers are dissembling
549 little rascals, and you often have to read between the lines to know what one
550 really means. For example, in UNIX C, you can get a message that says "floating
551 exception" for an integer divide-by-0. With C++'s Standard Template Library,
552 you can get a pair of error messages: the first message is the real error in the use
553 of the STL; the second message is a message from the compiler saying, "Error
554 message too long for printer to print; message truncated." You can probably
555 come up with many examples of your own.

556 ***Don't trust the compiler's second message***

557 Some compilers are better than others at detecting multiple errors. Some
558 compilers get so excited after detecting the first error that they become giddy and
559 overconfident; they prattle on with dozens of error messages that don't mean
560 anything. Other compilers are more levelheaded, and although they must feel a
561 sense of accomplishment when they detect an error, they refrain from spewing
562 out inaccurate messages. If you can't quickly find the source of the second or
563 third error message, don't worry about it. Fix the first one and recompile.

564 ***Divide and conquer***

565 The idea of dividing the program into sections to help detect defects works
566 especially well for syntax errors. If you have a troublesome syntax error, remove
567 part of the code and compile again. You'll either get no error (because the error's
568 in the part you removed), get the same error (meaning you need to remove a

569
570

571 **CROSS-REFERENCE** Many
572 programming text editors
573 can automatically find
574 matching braces or *begin-end*
575 pairs. For details on
programming editors, see
“Editing” in Section 30.2.

576

577
578
579
580
581

582 **KEY POINT**

583
584
585
586
587
588
589

590 **HARD DATA**

591
592
593
594
595
596
597
598
599
600

601
602
603
604

different part), or get a different error (because you’ll have tricked the compiler into producing a message that makes more sense).

Find extra comments and quotation marks

If your code is tripping up the compiler because it contains an extra quotation mark or beginning comment somewhere, insert the following sequence systematically into your code to help locate the defect:

C/C++/Java `/*"/**/`

23.3 Fixing a Defect

The hard part is finding the defect. Fixing the defect is the easy part. But as with many easy tasks, the fact that it’s easy makes it especially error-prone. At least one study found that defect corrections have more than a 50 percent chance of being wrong the first time (Yourdon 1986b). Here are a few guidelines for reducing the chance of error:

Understand the problem before you fix it

“The Devil’s Guide to Debugging” is right: The best way to make your life difficult and corrode the quality of your program is to fix problems without really understanding them. Before you fix a problem, make sure you understand it to the core. Triangulate the defect both with cases that should reproduce the error and with cases that shouldn’t reproduce the error. Keep at it until you understand the problem well enough to predict its occurrence correctly every time.

Understand the program, not just the problem

If you understand the context in which a problem occurs, you’re more likely to solve the problem completely rather than only one aspect of it. A study done with short programs found that programmers who achieve a global understanding of program behavior have a better chance of modifying it successfully than programmers who focus on local behavior, learning about the program only as they need to (Littman et al. 1986). Because the program in this study was small (280 lines), it doesn’t prove that you should try to understand a 50,000-line program completely before you fix a defect. It does suggest that you should understand at least the code in the vicinity of the defect correction—the “vicinity” being not a few lines but a few hundred.

Confirm the defect diagnosis

Before you rush to fix a defect, make sure that you’ve diagnosed the problem correctly. Take the time to run test cases that prove your hypothesis and disprove competing hypotheses. If you’ve proven only that the error could be the result of

605 one of several causes, you don't yet have enough evidence to work on the one
606 cause; rule out the others first.

607 ***Relax***
608 A programmer was ready for a ski trip. His product was ready to ship, he was
609 already late, and he had only one more defect to correct. He changed the source
610 file and checked it into version control. He didn't recompile the program and
611 didn't verify that the change was correct.

612 ***Never debug standing up.***
613 —Gerald Weinberg
614

615 If this isn't the height of recklessness, it's close, and it's common. Hurrying to
616 solve a problem is one of the most time-ineffective things you can do. It leads to
617 rushed judgments, incomplete defect diagnosis, and incomplete corrections.
618 Wishful thinking can lead you to see solutions where there are none. The
619 pressure—often self-imposed—encourages haphazard trial-and-error solutions,
620 sometimes assuming that a solution works without verifying that it does.

621 In striking contrast, during the final days of Microsoft Windows 2000
622 development, a developer needed to fix a defect that was the last remaining
623 defect before a Release Candidate could be created. The developer changed the
624 code, checked his fix, and tested his fix on his local build. But he didn't check
625 the fix into version control at that point. Instead, he went to play basketball. He
626 said, “I'm feeling too stressed right now to be sure that I've considered
627 everything I should consider. I'm going to clear my mind for an hour, and then
628 I'll come back and check in the code—once I've convinced myself that the fix is
629 really correct.”

630 Relax long enough to make sure your solution is right. Don't be tempted to take
631 shortcuts. It might take more time, but it'll probably take less. If nothing else,
632 you'll fix the problem correctly and your manager won't call you back from your
633 ski trip.

634 **CROSS-REFERENCE** Gen
635 eral issues involved in
636 changing code are discussed
637 in depth in Chapter 24,
638 “Refactoring.”

639 ***Fix the problem, not the symptom***
640 You should fix the symptom too, but the focus should be on fixing the
641 underlying problem rather than wrapping it in programming duct tape. If you

642
643 don't thoroughly understand the problem, you're not fixing the code. You're
fixing the symptom and making the code worse. Suppose you have this code:

644 Java Example of Code That Needs to Be Fixed

```
645 for ( claimNumber = 0; claimNumber < numClaims[ client ]; claimNumber++ ) {  
646     sum[ client ] = sum[ client ] + claimAmount[ claimNumber ];  
647 }
```

Further suppose that when *client* equals 45, *sum* turns out to be wrong by \$3.45.
Here's the wrong way to fix the problem:

650 CODING HORROR

```
651  
652  
653  
654  
655     Here's the "fix."  
656  
657  
658  
659
```

660 CODING HORROR

```
661  
662  
663  
664  
665  
666     Here's the second "fix."  
667  
668  
669  
670  
671  
672  
673  
674
```

- The fixes won't work most of the time. The problems look as though they're the result of initialization defects. Initialization defects are, by definition, unpredictable, so the fact that the sum for client 45 is off by \$3.45 today doesn't tell you anything about tomorrow. It could be off by \$10,000.02, or it could be correct. That's the nature of initialization defects.

- 680 ● It's unmaintainable. When code is special-cased to work around errors, the
681 special cases become the code's most prominent feature. The \$3.45 won't
682 always be \$3.45, and another error will show up later. The code will be
683 modified again to handle the new special case, and the special case for \$3.45
684 won't be removed. The code will become increasingly barnacled with
685 special cases. Eventually the barnacles will be too heavy for the code to
686 support, and the code will sink to the bottom of the ocean—a fitting place
687 for it.
- 688 ● It uses the computer for something that's better done by hand. Computers
689 are good at predictable, systematic calculations, but humans are better at
690 fudging data creatively. You'd be wiser to treat the output with Whiteout
691 and a typewriter than to monkey with the code.

692 ***Change the code only for good reason***

693 Related to fixing symptoms is the technique of changing code at random until it
694 seems to work. The typical line of reasoning goes like this: "This loop seems to
695 contain a defect. It's probably an off-by-one error, so I'll just put a *-1* here and
696 try it. OK. That didn't work, so I'll just put a *+1* in instead. OK. That seems to
697 work. I'll say it's fixed."

698 As popular as this practice is, it isn't effective. Making changes to code
699 randomly is like poking a Pontiac Aztek with a stick to see if it moves. You're
700 not learning anything; you're just goofing around. By changing the program
701 randomly, you say in effect, "I don't know what's happening here, but I'll try
702 this change and hope it works." Don't change code randomly. That's voodoo
703 programming. The more different you make it without understanding it, the less
704 confidence you'll have that it works correctly.

705 Before you make a change, be confident that it will work. Being wrong about a
706 change should leave you astonished. It should cause self-doubt, personal
707 reevaluation, and deep soul-searching. It should happen rarely.

708 ***Make one change at a time***

709 Changes are tricky enough when they're done one at a time. When done two at a
710 time, they can introduce subtle errors that look like the original errors. Then
711 you're in the awkward position of not knowing whether (1) you didn't correct
712 the error, (2) you corrected the error but introduced a new one that looks similar,
713 or (3) you didn't correct the error and you introduced a similar new error. Keep it
714 simple: Make just one change at a time.

715 **CROSS-REFERENCE** For
716 details on automated
717 regression testing, see
718 “Retesting (Regression
719 Testing)” in Section 22.6.
720

721
722
723

724
725
726
727
728
729
730

731
732

733 **FURTHER READING** For an
734 excellent discussion of
735 psychological issues in
736 debugging, as well as many
737 other areas of software
738 development, see *The
739 Psychology of Computer
740 Programming* (Weinberg
741 1998).

742

743

744
745
746
747
748

Check your fix

Check the program yourself, have someone else check it for you, or walk through it with someone else. Run the same triangulation test cases you used to diagnose the problem to make sure that all aspects of the problem have been resolved. If you’ve solved only part of the problem, you’ll find out that you still have work to do.

Rerun the whole program to check for side effects of your changes. The easiest and most effective way to check for side effects is to run the program through an automated suite of regression tests in JUnit, CppUnit, or equivalent.

Look for similar defects

When you find one defect, look for others that are similar. Defects tend to occur in groups, and one of the values of paying attention to the kinds of defects you make is that you can correct all the defects of that kind. Looking for similar defects requires you to have a thorough understanding of the problem. Watch for the warning sign: If you can’t figure out how to look for similar defects, that’s a sign that you don’t yet completely understand the problem.

23.4 Psychological Considerations in Debugging

Debugging is as intellectually demanding as any other software-development activity. Your ego tells you that your code is good and doesn’t have a defect even when you have seen that it has one. You have to think precisely—formulating hypotheses, collecting data, analyzing hypotheses, and methodically rejecting them—with a formality that’s unnatural to many people. If you’re both building code and debugging it, you have to switch quickly between the fluid, creative thinking that goes with design and the rigidly critical thinking that goes with debugging. As you read your code, you have to battle the code’s familiarity and guard against seeing what you expect to see.

How “Psychological Set” Contributes to Debugging Blindness

When you see a token in a program that says *Num*, what do you see? Do you see a misspelling of the word “Numb”? Or do you see the abbreviation for “Number”? Most likely, you see the abbreviation for “Number.” This is the phenomenon of “psychological set”—seeing what you expect to see. What does this sign say?



749

750

751

752

753

754

755

756

HARD DATA

757

758

759

760

761

762

763

764

765

766

G23xx03

In this classic puzzle, people often see only one “the.” People see what they expect to see. Consider the following:

- Students learning *while* loops often expect a loop to be continuously evaluated; that is, they expect the loop to terminate as soon as the *while* condition becomes false, rather than only at the top or bottom (Curtis et al. 1986). They expect a *while* loop to act as “while” does in natural language.
- A programmer who unintentionally used both the variable *SYSTSTS* and the variable *SYSSTSTS* thought he was using a single variable. He didn’t discover the problem until the program had been run hundreds of times, and a book was written containing the erroneous results (Weinberg 1998).
- A programmer looking at code like this code:

```
if ( x < y )
    swap = x
    x = y
    y = swap
```

sometimes sees code like this code:

```
if ( x < y ) {
    swap = x
    x = y
    y swap
}
```

People expect a new phenomenon to resemble similar phenomena they’ve seen before. They expect a new control construct to work the same as old constructs; programming-langauge *while* statements to work the same as real-life “while” statements; and variable names to be the same as they’ve been before. You see what you expect to see and thus overlook differences, like the misspelling of the word “language” in the previous sentence.

What does psychological set have to do with debugging? First, it speaks to the importance of good programming practices. Good formatting, commenting, variable names, routine names, and other elements of programming style help structure the programming background so that likely defects appear as variations and stand out.

783
784
785
786
787
788
789
790

The second impact of psychological set is in selecting parts of the program to
examine when an error is found. Research has shown that the programmers who
debug most effectively mentally slice away parts of the program that aren't
relevant during debugging (Basili, Selby, and Hutchens 1986). In general, the
practice allows excellent programmers to narrow their search fields and find
defects more quickly. Sometimes, however, the part of the program that contains
the defect is mistakenly sliced away. You spend time scouring a section of code
for a defect, and you ignore the section that contains the defect.

791
792
793

You took a wrong turn at the fork in the road and need to back up before you can
go forward again. Some of the suggestions in Section 23.2's discussion of tips
for finding defects are designed to overcome this "debugging blindness."

794 How "Psychological Distance" Can Help

795 **CROSS-REFERENCE** For
796 details on creating variable
names that won't be
797 confusing, see Section 11.7,
798 "Kinds of Names to Avoid."
799
800
801
802

Psychological distance can be defined as the ease with which two items can be
differentiated. If you are looking at a long list of words and have been told that
they're all about ducks, you could easily mistake "Queck" for "Quack" because
the two words look similar. The psychological distance between the words is
small. You would be much less likely to mistake "Tuack" for "Quack" even
though the difference is only one letter again. "Tuack" is less like "Quack" than
"Queck" is because the first letter in a word is more prominent than the one in
the middle.

803

Here are examples of psychological distances between variable names:

804
805

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

First Variable	Second Variable	Psychological Distance
stoppt	stcppt	Almost invisible
shiftrn	shiftrm	Almost none
dcount	bcount	Small
claims1	claims2	Small
product	sum	Large

806
807
808
809

As you debug, be ready for the problems caused by insufficient psychological
distance between similar variable names and between similar routine names. As
you construct code, choose names with large differences so that you avoid the
problem.

810

811

812 **CROSS-REFERENCE** The
813 line between testing tools and
814 debugging tools is fuzzy. For
815 details on testing tools, see
Section 22.5, "Test-Support
Tools." For details on tools
816 for other software-
development activities, see
817 Chapter 30, "Programming
818 Tools."

819

820

821

822

823

824 **KEY POINT**

825

826

827

828

829

830

831

832

833

834

835

836

837

838

839

840

841

842

843

844

845

23.5 Debugging Tools—Obvious and Not-So-Obvious

You can do much of the detailed, brain-busting work of debugging with debugging tools that are readily available. The tool that will drive the final stake through the heart of the defect vampire isn't yet available, but each year brings an incremental improvement in available capabilities.

Diff

A source-code comparator such as Diff is useful when you're modifying a program in response to errors. If you make several changes and need to remove some that you can't quite remember, a comparator can pinpoint the differences and jog your memory. If you discover a defect in a new version that you don't remember in an older version, you can compare the files to determine exactly what changed.

Compiler Warning Messages

One of the simplest and most effective debugging tools is your own compiler.

Set your compiler's warning level to the highest, pickiest level possible and fix the code so that it doesn't produce any compiler warnings

It's sloppy to ignore compiler errors. It's even sloppier to turn off the warnings so that you can't even see them. Children sometimes think that if they close their eyes and can't see you, they've made you go away. Setting a switch on the compiler to turn off warnings just means you can't see the errors. It doesn't make them go away any more than closing your eyes makes an adult go away.

Assume that the people who wrote the compiler know a great deal more about your language than you do. If they're warning you about something, it usually means you have an opportunity to learn something new about your language. Make the effort to understand what the warning really means.

Treat warnings as errors

Some compilers let you treat warnings as errors. One reason to use the feature is that it elevates the apparent importance of a warning. Just as setting your watch five minutes fast tricks you into thinking it's five minutes later than it is, setting your compiler to treat warnings as errors tricks you into taking them more seriously. Another reason to treat warnings as errors is that they often affect how your program compiles. When you compile and link a program, warnings typically won't stop the program from linking but errors typically will. If you want to check warnings before you link, set the compiler switch that treats warnings as errors.

846 ***Initiate project wide standards for compile-time settings***
847 Set a standard that requires everyone on your team to compile code using the
848 same compiler settings. Otherwise, when you try to integrate code compiled by
849 different people with different settings, you'll get a flood of error messages and
850 an integration nightmare.

851 **Extended Syntax and Logic Checking**

852 You can use additional tools to check your code more thoroughly than your
853 compiler does. For example, for C programmers, the lint utility painstakingly
854 checks for use of uninitialized variables, writing = when you mean ==, and
855 similarly subtle problems.

856 **Execution Profiler**

857 You might not think of an execution profiler as a debugging tool, but a few
858 minutes spent studying a program profile can uncover some surprising (and
859 hidden) defects.

860 For example, I had suspected that a memory-management routine in one of my
861 programs was a performance bottleneck. Memory management had originally
862 been a small component using a linearly ordered array of pointers to memory. I
863 replaced the linearly ordered array with a hash table in the expectation that
864 execution time would drop by at least half. But after profiling the code, I found
865 no change in performance at all. I examined the code more closely and found a
866 defect that was wasting a huge amount of time in the allocation algorithm. The
867 bottleneck hadn't been the linear-search technique; it was the defect. I hadn't
868 needed to optimize the search after all. Examine the output of an execution
869 profiler to satisfy yourself that your program spends a reasonable amount of time
870 in each area.

871 **Test Frameworks/Scaffolding**

872 **CROSS-REFERENCE** For
873 details on scaffolding, see
874 "Building Scaffolding to Test
Individual Classes" in
Section 22.5.

875 **Debugger**

876 Commercially available debuggers have advanced steadily over the years, and
877 the capabilities available today can change the way you program.

878 Good debuggers allow you to set breakpoints to break when execution reaches a
879 specific line, or the *n*th time it reaches a specific line, or when a global variable
880 changes, or when a variable is assigned a specific value. They allow you to step

881 through code line by line, stepping through or over routines. They allow the
882 program to be executed backwards, stepping back to the point where a defect
883 originated. They allow you to log the execution of specific statements—similar
884 to scattering “I’m here!” print statements throughout a program.

885 Good debuggers allow full examination of data, including structured and
886 dynamically allocated data. They make it easy to view the contents of a linked
887 list of pointers or a dynamically allocated array. They’re intelligent about user-
888 defined data types. They allow you to make ad hoc queries about data, assign
889 new values, and continue program execution.

890 You can look at the high-level language or the assembly language generated by
891 your compiler. If you’re using several languages, the debugger automatically
892 displays the correct language for each section of code. You can look at a chain of
893 calls to routines and quickly view the source code of any routine. You can
894 change parameters to a program within the debugger environment.

895 The best of today’s debuggers also remember debugging parameters
896 (breakpoints, variables being watched, and so on) for each individual program so
897 that you don’t have to re-create them for each program you debug.

898 System debuggers operate at the systems level rather than the applications level
899 so that they don’t interfere with the execution of the program being debugged.
900 They’re essential when you are debugging programs that are sensitive to timing
901 or the amount of memory available.

902 *An interactive debugger
903 is an outstanding
904 example of what is not
905 needed—it encourages
906 trial-and-error hacking
907 rather than systematic
908 design, and also hides
909 marginal people barely
910 qualified for precision
911 programming.*
912 —Harlan Mills
913
914

Given the enormous power offered by modern debuggers, you might be surprised that anyone would criticize them. But some of the most respected people in computer science recommend not using them. They recommend using your brain and avoiding debugging tools altogether. Their argument is that debugging tools are a crutch and that you find problems faster by thinking about them than by relying on tools. They argue that you, rather than the debugger, should mentally execute the program to flush out defects.

915 **KEY POINT**
916
917

Regardless of the empirical evidence, the basic argument against debuggers isn’t valid. The fact that a tool can be misused doesn’t imply that it should be rejected. You wouldn’t avoid taking aspirin merely because it’s possible to overdose. You wouldn’t avoid mowing your lawn with a power mower just because it’s possible to cut yourself. Any other powerful tool can be used or abused, and so can a debugger.

CC2E.COM/2368
918

CHECKLIST: Debugging Reminders

919 **Techniques for Finding Defects**

- 920 Use all the data available to make your hypothesis
- 921 Refine the test cases that produce the error
- 922 Exercise the code in your unit test suite
- 923 Use available tools
- 924 Reproduce the error several different ways
- 925 Generate more data to generate more hypotheses
- 926 Use the results of negative tests
- 927 Brainstorm for possible hypotheses
- 928 Narrow the suspicious region of the code
- 929 Be suspicious of classes and routines that have had defects before
- 930 Check code that's changed recently
- 931 Expand the suspicious region of the code
- 932 Integrate incrementally
- 933 Check for common defects
- 934 Talk to someone else about the problem
- 935 Take a break from the problem
- 936 Set a maximum time for quick and dirty debugging
- 937 Make a list of brute force techniques, and use them

938 **Techniques for Syntax Errors**

- 939 Don't trust line numbers in compiler messages
- 940 Don't trust compiler messages
- 941 Don't trust the compiler's second message
- 942 Divide and conquer
- 943 Find extra comments and quotation marks

944 **Techniques for Fixing Defects**

- 945 Understand the problem before you fix it
- 946 Understand the program, not just the problem
- 947 Confirm the defect diagnosis
- 948 Relax
- 949 Save the original source code
- 950 Fix the problem, not the symptom

- 951 Change the code only for good reason
952 Make one change at a time
953 Check your fix
954 Look for similar defects

955 **General Approach to Debugging**

- 956 Do you use debugging as an opportunity to learn more about your program,
957 mistakes, code quality, and problem-solving approach?
958 Do you avoid the trial-and-error, superstitious approach to debugging?
959 Do you assume that errors are your fault?
960 Do you use the scientific method to stabilize intermittent errors?
961 Do you use the scientific method to find defects?
962 Rather than using the same approach every time, do you use several different
963 techniques to find defects?
964 Do you verify that the fix is correct?
965 Do you use compiler warning messages, execution profiling, a test
966 framework, scaffolding, and interactive debugging?

967 CC2E.COM/2375

968 **Additional Resources**

969 Agans, David J. *Debugging: The Nine Indispensable Rules for Finding Even the*
970 *Most Elusive Software and Hardware Problems*. Amacom, 2003. This book
971 provides general debugging principles that can be applied in any language or
972 environment.

973 Myers, Glenford J. *The Art of Software Testing*. New York: John Wiley, 1979.
974 Chapter 7 of this classic book is devoted to debugging.

975 Allen, Eric. *Bug Patterns In Java*. Berkeley, Ca.: Apress, 2002. This book lays
976 out an approach to debugging Java programs that is conceptually very similar to
977 what is described in this chapter, including “The Scientific Method of
978 Debugging,” distinguishing between debugging and testing, and identifying
979 common bug patterns.

980 The following two books are similar in that their titles suggest they are
981 applicable only to Microsoft Windows and .NET programs, but they both
982 contain discussions of debugging in general, use of assertions, and coding
983 practices that help to avoid bugs in the first place.

984 Robbins, John. *Debugging Applications for Microsoft .NET and Microsoft*
985 *Windows*. Redmond, Wa.: Microsoft Press, 2003.

986 McKay, Everett N. and Mike Woodring, *Debugging Windows Programs:*
987 *Strategies, Tools, and Techniques for Visual C++ Programmers*. Boston, Mass.:
988 Addison Wesley, 2000.

989 Key Points

- 990 • Debugging is a make-or-break aspect of software development. The best
991 approach is to use other techniques described in this book to avoid defects in
992 the first place. It's still worth your time to improve your debugging skills,
993 however, because the difference between good and poor debugging
994 performance is at least 10 to 1.
- 995 • A systematic approach to finding and fixing errors is critical to success.
996 Focus your debugging so that each test moves you a step forward. Use the
997 Scientific Method of Debugging.
- 998 • Understand the root problem before you fix the program. Random guesses
999 about the sources of errors and random corrections will leave the program in
1000 worse condition than when you started.
- 1001 • Set your compiler warning to the pickiest level possible, and fix the errors it
1002 reports. It's hard to fix subtle errors if you ignore the obvious ones.
- 1003 • Debugging tools are powerful aids to software development. Find them and
1004 use them. Remember to use your brain at the same time.

24

Refactoring

3 CC2E.COM/2436

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19 *All successful software
20 gets changed.*

21 —Fred Brooks

22

23

24

25

26

Contents

- 24.1 Kinds of Software Evolution
- 24.2 Introduction to Refactoring
- 24.3 Reasons to Refactor
- 24.4 Specific Refactorings
- 24.5 Refactoring Safely
- 24.6 Refactoring Strategies

Related Topics

- Tips for fixing defects: Section 23.3
- Code tuning approach: Section 25.6
- High-level design: Chapter 5
- High-quality classes: Chapter 6
- High-quality routines: Chapter 7
- Collaborative construction: Chapter 21
- Developer testing: Chapter 22
- Areas likely to change: “Identify Areas Likely to Change” in Section 5.3

MYTH: A WELL-MANAGED SOFTWARE PROJECT conducts methodical requirements development and defines a stable list of the program’s responsibilities. Design follows requirements, and it is done carefully so that coding can proceed linearly, from start to finish, implying that most of the code can be written once, tested, and forgotten. According to the myth, the only time that the code is significantly modified is during the software-maintenance phase, something that happens only after the initial version of a system has been delivered.

HARD DATA

27 Reality: Code evolves substantially during its initial development. Many of the
28 changes seen during initial coding are at least as dramatic as changes seen during
29 maintenance. Coding, debugging, and unit testing consume between 30 to 65
30 percent of the effort on a typical project, depending on the project's size. (See
31 Chapter 27, "How Program Size Affects Construction," for details.) If coding
32 and unit testing were straightforward processes, they would consume no more
33 than 20-30 percent of the total effort on a project. Even on well-managed
34 projects, however, requirements change by about one to four percent per month
35 (Jones 2000). Requirements changes invariably cause corresponding code
36 changes—sometimes substantial code changes.

37 **KEY POINT**
38 Another Reality: Modern development practices increase the potential for code
39 changes during construction. In older life cycles, the focus—successful or not—
40 was on avoiding code changes. More modern approaches move away from
41 coding predictability. Current approaches are more code-centered, and over the
life of a project, you can expect code to evolve more than ever.

42 24.1 Kinds of Software Evolution

43 Software evolution is like biological evolution in that some mutations are
44 beneficial and many mutations are not. Good software evolution produces code
45 whose development mimics the ascent from monkeys to Neanderthals to our
46 current exalted state as software developers. Evolutionary forces sometimes beat
47 on a program the other way, however, knocking the program into a de-
48 evolutionary spiral.

49 **KEY POINT**
50 The key distinction between kinds of software evolution is whether the
51 program's quality improves or degrades under modification. If you fix errors
52 with logical duct tape and superstition, quality degrades. If you treat
53 modifications as opportunities to tighten up the original design of the program,
54 quality improves. If you see that program quality is degrading, that's like a
55 canary in a mine shaft that has stopped singing. It's a warning that the program is
evolving in the wrong direction.

56 A second distinction in the kinds of software evolution is the one between
57 changes made during construction and those made during maintenance. These
58 two kinds of evolution differ in several ways. Construction changes are usually
59 made by the original developers, usually before the program has been completely
60 forgotten. The system isn't yet on line, so the pressure to finish changes is only
61 schedule pressure—it's not 500 angry users wondering why their system is
62 down. For the same reason, changes during construction can be more
63 freewheeling—the system is in a more dynamic state, and the penalty for making

64
65

mistakes is low. These circumstances imply a style of software evolution that's different from what you'd find during software maintenance.

66

67 ***There is no code so big,***
68 ***twisted, or complex that***
69 ***maintenance can't make***
70 ***it worse.***

—Gerald Weinberg

71
72
73
74
75
76

A common weakness in programmers' approaches to software evolution is that it goes on as an un-self-conscious process. If you recognize that evolution during development is an inevitable and important phenomenon and plan for it, you can use it to your advantage.

Evolution is at once hazardous and an opportunity to approach perfection. When you have to make a change, strive to improve the code so that future changes are easier. You never know as much when you begin writing a program as you do afterward. When you have a chance to revise a program, use what you've learned to improve it. Make both your initial code and your changes with future changes in mind.

77 **KEY POINT**
78
79

The Cardinal Rule of Software Evolution is that evolution should improve the internal quality of the program. The following sections describe how to accomplish this.

80

24.2 Introduction to Refactoring

81
82
83
84
85
86
87
88

The key strategy in achieving The Cardinal Rule of Software Evolution is refactoring, which Martin Fowler defines as "a change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its observable behavior" (Fowler 1999). The word "refactoring" in modern programming grew out of Larry Constantine's original use of the word "factoring" in structured programming, which referred to decomposing a program into its constituent parts as much as possible (Yourdon and Constantine 1979).

89

24.3 Reasons to Refactor

90
91
92
93

Sometimes code degenerates under maintenance, and sometimes the code just wasn't very good in the first place. In either case, here are some warning signs —sometimes called "smells" (Fowler 1999)—that indicate where refactorings are needed.

94
95
96

Code is duplicated

Duplicated code almost always represents a failure to fully factor the design in the first place. Duplicate code sets you up to make parallel modifications—

97 whenever you have to make changes in one place, you have to make parallel
98 changes in another place. It also violates what Andrew Hunt and Dave Thomas
99 refer to as the “DRY principle”—Don’t Repeat Yourself (2000). I think David
100 Parnas said it best: “Copy and Paste is a design error” (McConnell 1998b).

101 ***A routine is too long***

102 In object-oriented programming, routines longer than a screen are rarely needed,
103 and usually represent the attempt to force-fit a structured programming foot into
104 an object-oriented shoe.

105 One of my clients was assigned the task of breaking up a legacy system’s longest
106 routine, which was more than 12,000 lines long. With effort, he was able to
107 reduce the size of the largest routine to only about 4,000 lines.

108 One way to improve a system is to increase its modularity—increase the number
109 of well-defined, well-named routines that do one thing and do it well. When
110 changes lead you to revisit a section of code, take the opportunity to check the
111 modularity of the routines in that section. If a routine would be cleaner if part of
112 it were made into a separate routine, create a separate routine.

113 ***A loop is too long or too deeply nested***

114 Loop innards tend to be good candidates for being converted into routines, which
115 helps to better factor the code and to reduce the complexity of the loop.

116 ***A class has poor cohesion***

117 If you find a class that takes ownership for a hodge-podge of unrelated
118 responsibilities, that class should be broken up into multiple classes, each of
119 which has responsibility for a cohesive set of responsibilities.

120 ***A class interface does not provide a consistent level of abstraction***

121 Even classes that begin life with a cohesive interface can lose their original
122 consistency. Class interfaces tend to morph over time as a result of modifications
123 that are made in the heat of the moment and that favor expediency to interface
124 integrity. Eventually the class interface becomes a Frankensteinian maintenance
125 monster that does little to improve the intellectual manageability of the program.

126 ***A parameter list has too many parameters***

127 Well-factored programs tend to have many small, well-defined routines that
128 don’t need large parameter lists. A long parameter list is a warning that the
129 abstraction of the routine interface has not been well thought out.

130 ***Changes within a class tend to be compartmentalized***

131 Sometimes a class has two or more distinct responsibilities. When that happens
132 you find yourself changing either one part of the class or another part of the
133 class—but few changes affect both parts of the class. That’s a sign that the class

134 should be cleaved into multiple classes along the lines of the separate
135 responsibilities.

136 ***Changes require parallel modifications to multiple classes***
137 I saw one project that had a checklist of about 15 classes that had to be modified
138 whenever a new kind of output was added. When you find yourself routinely
139 making changes to the same set of classes, that suggests the code in those classes
140 could be rearranged so that changes affect only one class. In my experience, this
141 is a hard ideal to accomplish, but it is nonetheless a good goal.

142 ***Inheritance hierarchies have to be modified in parallel***
143 Finding yourself making a subclass of one class every time you make a subclass
144 of another class is a special kind of parallel modification.

145 ***case statements have to be modified in parallel***
146 Case statements are not inherently bad, but if you find yourself making parallel
147 modifications to similar *case* statements in multiple parts of the program, you
148 should ask whether inheritance might be a better approach.

149 ***Related data items that are used together are not organized into classes***
150 If you find yourself repeatedly manipulating the same set of data items, you
151 should ask whether those manipulations should be combined into a class of their
152 own.

153 ***A routine uses more features of another class than of its own class***
154 This suggests that the routine should be moved into the other class and then
155 invoked by its old class.

156 ***A primitive data type is overloaded***
157 Primitive data types can be used to represent an infinite number of real-world
158 entities. If your program uses a primitive data type like an integer to represent a
159 common entity such as money, consider creating a simple *Money* class so that
160 the compiler can perform type checking on *Money* variables, so that you can add
161 safety checks on the values assigned to money, and so on. If both *Money* and
162 *Temperature* are integers, the compiler won't warn you about erroneous
163 assignments like *bankBalance = recordLowTemperature*.

164 ***A class doesn't do very much***
165 Sometimes the result of refactoring code is that an old class doesn't have much
166 to do. If a class doesn't seem to be carrying its weight, ask if you should assign
167 all of that class's responsibilities to other classes and eliminate the class
168 altogether.

169 *A chain of routines passes tramp data*

170 Finding yourself passing data to one routine just so that routine can pass it to
171 another routine is called “tramp data” (Page-Jones 1988). This might be OK, or
172 it might not. Ask whether passing the specific data in question is consistent with
173 the abstraction presented by each of the routine interfaces. If the abstraction for
174 each routine is OK, passing the data is OK. If not, find some way to make each
175 routine’s interface more consistent.

176 *A middle man object isn’t doing anything*

177 If you find that most of the code in a class is just passing off calls to routines in
178 other classes, consider whether you should eliminate the middleman and call
179 those other classes directly.

180 *One class is overly intimate with another*

181 Encapsulation (information hiding) is probably the strongest tool you have to
182 make your program intellectually manageable and to minimize ripple effects of
183 code changes. Anytime you see one class that knows more about another class
184 than it should (including derived classes knowing too much about their parents),
185 err on the side of stronger encapsulation rather than weaker.

186 *A routine has a poor name*

187 If a routine has a poor name, change the name of the routine where it’s defined,
188 change the name in all places it’s called, and then recompile. As hard as it might
189 be to do this now, it will be even harder later, so do it as soon as you notice it’s a
190 problem.

191 *Data members are public*

192 Public data members are, in my view, always a bad idea. They blur the line
193 between interface and implementation. They inherently violate encapsulation and
194 limit future flexibility. Strongly consider hiding public data members behind
195 access routines.

196 *A subclass uses only a small percentage of its parents’ routines*

197 Typically this indicates that that subclass has been created because a parent class
198 happened to contain the routines it needed, not because the subclass is logically a
199 descendent of the superclass. Consider achieving better encapsulation by
200 switching the subclass’s relationship to its superclass from an is-a relationship to
201 a has-a relationship; convert the superclass to member data of the former
202 subclass and expose only the routines in the former subclass that are really
203 needed.

204 *Comments are used to explain difficult code*

205 Comments have an important role to play, but they should not be used as a
206 crutch to explain bad code. The age-old wisdom is dead on: “Don’t document
207 bad code—rewrite it” (Kernighan and Plauger 1978).

208
209
210
211
212
213
214
215
216
217
218
219
220

Global variables are used

When you revisit a section of code that uses global variables, take time to re-examine them. You might have thought of a way to avoid using global variables since the last time you visited that part of the code. Because you're less familiar with the code than when you first wrote it, you might now find the global variables sufficiently confusing that you're willing to develop a cleaner approach. You might also have a better sense of how to isolate global variables in access routines and a keener sense of the pain caused by not doing so. Bite the bullet and make the beneficial modifications. The initial coding will be far enough in the past that you can be objective about your work yet close enough that you will still remember most of what you need in order to make the revisions correctly. The time during early revisions is the perfect time to improve the code.

221
222
223

A routine uses setup code before a routine call or takedown code after a routine call

Code like this is a warning:

224
225
226
227
228
229
230
231
232
233
234
235
236
237
238

```
WithdrawalTransaction withdrawal;
withdrawal.SetCustomerId( customerId );
withdrawal.SetBalance( balance );
withdrawal.SetWithdrawalAmount( withdrawalAmount );
withdrawal.SetWithdrawalDate( withdrawalDate );

ProcessWithdrawal( withdrawal );

customerId = withdrawal.GetCustomerId();
balance = withdrawal.GetBalance();
withdrawalAmount = withdrawal.GetWithdrawalAmount();
withdrawalDate = withdrawal.GetWithdrawalDate();
```

A similar warning sign is when you find yourself creating a special constructor for the *WithdrawalTransaction* class that takes a subset of its normal initialization data so that you can write code like this:

239
240
241
242
243
244
245
246
247

```
withdrawal = new WithdrawalTransaction( customerId, balance,
                                         withdrawalAmount, withdrawalDate );
ProcessWithdrawal( withdrawal );
delete withdrawal;
```

Anytime you see code that sets up for a call to a routine or takes down after a call to a routine, ask whether the routine interface is presenting the right abstraction. In this case, perhaps the *ProcessWithdrawal()* routine should be added to the *WithdrawalTransaction* class, or perhaps the parameter list of *ProcessWithdrawal* should be modified to support code like this:

248

```
ProcessWithdrawal( balance, withdrawalAmount, withdrawalDate );
```

249
250
251

Note that the converse of this example presents a similar problem. If you find
yourself usually having a *WithdrawalTransaction* object in hand, but needing to
pass several of its values to a routine like this:

252 `ProcessWithdrawal(withdrawal.GetCustomerId(), withdrawal.GetBalance(),`
253 `withdrawal.GetWithdrawalAmount(), withdrawal.GetWithdrawalDate());`
254
255
256
257
258
259

you should also consider refactoring the *ProcessWithdrawal* interface so that it
requires the *WithdrawalTransaction* object rather than its individual fields. Any
of these approaches can be right and any can be wrong; it depends on whether
the abstraction of the *ProcessWithdrawal()* interface is that it expects to have
four distinct pieces of data or that it expects to have a *WithdrawalTransaction*
object.

260 **CROSS-REFERENCE** For
261 guidelines on the use of
262 global variables, see Section
263 13.3, “Global Data.” For an
264 explanation of the differences
between global data and class
265 data, see “Class Data
266 Mistaken For Global Data” in
Section 5.3.
267
268
269
270

271
272
273
274

275
276
277
278
279

280
281
282

283
284
285
286
287

Experts agree that the best way to prepare for future requirements is not to write
speculative code; it’s to make the *currently required* code as clear and
straightforward as possible so that future programmers will know what it does
and does not do, and can make their changes accordingly (Fowler 1999, Beck
2000).

CC2E.COM/2443

288

289

290

291

292

293

294

295

296

297

298

299

300

301

302

303

304

305

306

307

308

309

310

311

312

313

314

CHECKLIST: Reasons to Refactor

- Code is duplicated
 - A routine is too long
 - A loop is too long or too deeply nested
 - A class has poor cohesion
 - A class interface does not provide a consistent level of abstraction
 - A parameter list has too many parameters
 - Changes within a class tend to be compartmentalized
 - Changes require parallel modifications to multiple classes
 - Inheritance hierarchies have to be modified in parallel
 - Related data items that are used together are not organized into classes
 - A routine uses more features of another class than of its own class
 - A primitive data type is overloaded
 - A class doesn't do very much
 - A chain of routines passes tramp data
 - A middle man object isn't doing anything
 - One class is overly intimate with another
 - A routine has a poor name
 - Data members are public
 - A subclass uses only a small percentage of its parents' routines
 - Comments are used to explain difficult code
 - Global variables are used
 - A routine uses setup code before a routine call or takedown code after a routine call
 - A program contains code that seems like it might be needed someday
-

Reasons Not To Refactor

In common parlance, “refactoring” is used loosely to refer to fixing defects, adding functionality, modifying the design—essentially as a synonym for making any change to the code whatsoever. This common dilution of the meaning of the term is unfortunate. Change in itself is not a virtue. But purposeful change, applied with a teaspoonful of discipline, can be the key strategy that supports steady improvement in a program’s quality under maintenance and prevents the all-too-familiar software-entropy death spiral.

322

24.4 Specific Refactorings

323

324

325

326

327

328

329

In this section, I present a catalog of refactorings. Many of them are summaries of the more detailed descriptions presented in *Refactoring* (Fowler 1999). I have not, however, attempted to make this catalog exhaustive. In a sense, every example in this book that shows a “bad code” example and a “good code” example is a candidate for becoming a refactoring. In the interest of not repeating the entire 900 page book in this section, I’ve tried to focus on the refactorings I personally have found most useful.

330

Data Level Refactorings

331

332

333

Replace a magic number with a named constant

If you’re using a numeric or string literal like *3.14*, replace that literal with a named constant like *PI*.

334

335

336

Rename a variable with a clearer or more informative name

If a variable’s name isn’t clear, change it to a better name. The same advice applies to renaming constants, classes and routines, of course.

337

338

339

Move an expression inline

Replace an intermediate variable that was assigned the result of an expression with the expression itself.

340

341

342

Replace an expression with a routine

Replace an expression with a routine (usually so that the expression isn’t duplicated in the code).

343

344

345

Introduce an intermediate variable

Assign an expression to an intermediate variable whose name summarizes the purpose of the expression.

346

347

348

349

Convert a multi-use variable to multiple single-use variables

If a variable is used for more than one purpose (common culprits are *i*, *j*, *temp*, and *x*), create separate variables for each usage, each of which has a more specific name.

350

351

352

Use a local variable for local purposes rather than a parameter

If an input-only routine parameter is being used as a local variable, create a local variable and use that instead.

353

354

355

Convert a data primitive to a class

If a data primitive needs additional behavior (including stricter type checking) or additional data, convert the data to an object and add the behavior you need. This

356 can apply to simple numeric types like *Money* and *Temperature*. It can also
357 apply to enumerated types like *Color*, *Shape*, *Country*, or *OutputType*.

358 ***Convert a set of type codes to a class***

359 In older programs, it's common to see associations like

```
360     const int SCREEN = 0;  
361     const int PRINTER = 1;  
362     const int FILE = 2;
```

363 Rather than defining standalone constants, create a class so that you can receive
364 the benefits of stricter type checking and set yourself up to provide richer
365 semantics for *OutputType* if you ever need to.

366 ***Convert a set of type codes to a class with subclasses***

367 If the different elements associated with different types might have different
368 behavior, then consider creating a base class for the type with subclasses for each
369 type code. For the *OutputType* base class, you might create subclasses like
370 *Screen*, *Printer*, and *File*.

371 ***Change an array to an object***

372 If you're using an array in which different elements are different types, create an
373 object that has a field for each former element of the array.

374 ***Encapsulate a collection***

375 If a class returns a collection, having multiple instances of the collection floating
376 around can create synchronization difficulties. Consider having the class return a
377 read-only collection and provide routines to add and remove elements from the
378 collection.

379 ***Replace a traditional record with a data class***

380 Create a class that contains the members of the record. Creating a class allows
381 you to centralize error checking, persistence, and other operations that concern
382 the record.

383 Statement Level Refactorings

384 ***Decompose a boolean expression***

385 Simplify a boolean expression by introducing well-named intermediate variables
386 that help document the meaning of the expression.

387 ***Move a complex boolean expression into a well-named boolean function***

388 If the expression is complicated enough, this can improve readability. If the
389 expression is used more than once, it eliminates the need for parallel
390 modifications and reduces the chance of error in using the expression.

391 ***Consolidate fragments that are duplicated within different parts of a***
392 ***conditional***

393 If you have the same lines of code repeated at the end of an *else* block that you
394 have at the end of the *if* block, move those lines of code so that they occur after
395 the entire *if-then-else* block.

396 ***Use break or return instead of a loop control variable***

397 If you have a variable within a loop like *Done* that's used to control the loop, use
398 *break* or *return* to exit the loop instead.

399 ***Return as soon as you know the answer instead of assigning a return value***
400 ***within nested if-then-else statements***

401 Code is often easiest to read and least error prone if you exit a routine as soon as
402 you know the return value. The alternative of setting a return value and then
403 unwinding your way through a lot of logic can be harder to follow.

404 ***Replace conditionals with polymorphism (especially repeated case***
405 ***statements)***

406 Much of the logic that used to be contained in *case* statements in structured
407 programs can instead be baked into the inheritance hierarchy and accomplished
408 through polymorphic routine calls instead.

409 ***Create and use null objects instead of testing for null values***

410 Sometimes a null object will have generic behavior or data associated with it,
411 such as referring to a resident whose name is not known as "occupant." In this
412 case, consider moving the responsibility for handling null values out of the client
413 code and into the class—that is, have the *Customer* class define the unknown
414 resident as "occupant" instead of having *Customer*'s client code repeatedly test
415 for whether the customer's name is known and substitute "occupant" if not.

416

Routine Level Refactorings

417 ***Extract a routine***

418 Remove inline code from one routine and turn it into its own routine.

419 ***Move a routine's code inline***

420 Take code from a routine whose body is simple and self-explanatory and move
421 that routine's code inline where it is used.

422 ***Convert a long routine to a class***

423 If a routine is too long, sometimes turning it into a class and then further
424 factoring the former routine into multiple routines will improve readability.

425 ***Substitute a simple algorithm for a complex algorithm***

426 Replace a complicated algorithm with a simpler algorithm.

427

428

429

430

431

432

433

434

435

436

437

438

439

440

441

442

443

444

445

446

447

448

449

450

451

452

453

454

455

456

457

458

459

460

Add a parameter

If a routine needs more information from its caller, add a parameter so that that information can be provided.

Remove a parameter

If a routine no longer uses a parameter, remove it.

Separate query operations from modification operations

Normally, query operations don't change an object's state. If an operation like *GetTotals()* changes an object's state, separate the query functionality from the state-changing functionality and provide two separate routines.

Combine similar routines by parameterizing them

Two similar routines might differ only with respect to a constant value that's used within the routine. Combine the routines into one routine and pass in the value to be used as a parameter.

Separate routines whose behavior depends on parameters passed in

If a routine executes different code depending on the value of an input parameter, consider breaking the routine into separate routines that can be called separately, without passing in that particular input parameter.

Pass a whole object rather than specific fields

If you find yourself passing several values from the same object into a routine, consider changing the routine's interface so that it takes the whole object instead.

Pass specific fields rather than a whole object

If you find yourself creating an object just so that you can pass it to a routine, consider modifying the routine so that it takes specific fields rather than a whole object.

Encapsulate downcasting

If a routine returns an object, it normally should return the most specific type of object it knows about. This is particularly applicable to routines that return iterators, collections, elements of collections, and so on.

Class Implementation Refactorings

Change value objects to reference objects

If you find yourself creating and maintaining numerous copies of large or complex objects, change your usage of those objects so that only one master copy exists (the value object) and the rest of the code uses references to that object (reference objects).

461 *Change reference objects to value objects*

462 If you find yourself performing a lot of reference housekeeping for small or
463 simple objects, change your usage of those objects so that all objects are value
464 objects.

465 *Replace virtual routines with data initialization*

466 If you have a set of subclasses that vary only according to constant values they
467 return, rather than overriding member routines in the derived classes, have the
468 derived classes initialize the class with appropriate constant values, and then
469 have generic code in the base class that works with those values.

470 *Change member routine or data placement*

471 There are several general changes to consider making in an inheritance
472 hierarchy. These changes are normally performed to eliminate duplication in
473 derived classes:

- 474 • Pull a routine up into its superclass
- 475 • Pull a field up into its superclass
- 476 • Pull a constructor body up into its superclass

477 Several other changes are normally made to support specialization in derived
478 classes:

- 479 • Push a routine down into its derived classes
- 480 • Push a field down into its derived classes
- 481 • Push a constructor body down into its derived classes

482 *Extract specialized code into a subclass*

483 If a class has code that's used by only a subset of its instances, move that
484 specialized code into its own subclass.

485 *Combine similar code into a superclass*

486 If two subclasses have similar code, combine that code and move it into the
487 superclass.

488 Class Interface Refactorings

489 *Move a routine to another class*

490 Create a new routine in the target class and move the body of the routine from
491 the source class into the target class. You can either call the new routine from the
492 old routine, or change surrounding code to use the new routine exclusively.

493 *Convert one class to two*

494 If a class has two or more distinct areas of responsibility, break the class into
495 multiple classes, each of which has a clearly defined responsibility.

496 *Eliminate a class*

497 If a class isn't doing very much, move its code into other classes that are more
498 cohesive and eliminate the class.

499 *Hide a delegate*

500 Sometimes Class A calls Class B and Class C, when really Class A should call
501 only Class B, and Class B should call Class C. Ask yourself what the right
502 abstraction is for A's interaction with B. If B should be responsible for calling C,
503 then have B call C.

504 *Replace inheritance with delegation*

505 If a class needs to use another class but wants more control over its interface,
506 make the superclass a field of the former subclass and then expose a set of
507 routines that will provide a cohesive abstraction.

508 *Replace delegation with inheritance*

509 If a class exposes every public routine of a delegate class (member class), inherit
510 from the delegate class instead of just using the class.

511 *Remove a middle man*

512 If Class A calls B, and Class B calls Class C, sometimes it works better to have
513 Class A call Class C directly. The question of whether you should delegate to
514 Class B or not depends on what will best maintain the integrity of Class B's
515 interface.

516 *Introduce a foreign routine*

517 If a class needs an additional routine and you can't modify the class to provide it,
518 you can create a new routine within the client class that provides that
519 functionality.

520 *Introduce an extension class*

521 If a class needs several additional routines and you can't modify the class, you
522 can create a new class that combines the unmodifiable class's functionality with
523 the additional functionality. You can do that either by subclassing the original
524 class and adding new routines or by wrapping the class and exposing the routines
525 you need.

526 *Encapsulate an exposed member variable*

527 If member data is public, change the member data to private and expose the
528 member data's value through a routine instead.

529 ***Remove Set() routines for fields that cannot be changed***
530 If a field is supposed to be set at object creation time and not changed afterward,
531 initialize that field in the object's constructor rather than providing a misleading
532 *Set()* routine.

533 ***Hide routines that are not intended to be used outside the class***
534 If the class interface would be more coherent without a routine, hide the routine.

535 ***Encapsulate unused routines***
536 If you find yourself routinely using only a portion of a class's interface, create a
537 new interface to the class that exposes only those necessary routines. Be sure that
538 the new interface provides a coherent abstraction.

539 ***Collapse a superclass and subclass if their implementations are very***
540 ***similar***
541 If the subclass doesn't provide much specialization, combine it into its
542 superclass.

543 **System Level Refactorings**

544 ***Create a definitive reference source for data you can't control***
545 Sometimes you have data maintained by the system that you can't conveniently
546 or consistently access from other objects that need to know about that data. A
547 common example is data maintained in a GUI control. In such a case, you can
548 create a class that mirrors the data in the GUI control, and then have both the
549 GUI control and the other code treat that class as the definitive source of that
550 data.

551 ***Change unidirectional class association to bidirectional class association***
552 If you have two classes that need to use each other's features, but only one class
553 can know about the other class, then change the classes so that they both know
554 about each other.

555 ***Change bidirectional class association to unidirectional class association***
556 If you have two classes that know about each other's features, but only one class
557 that really needs to know about the other, change the classes so that one knows
558 about the other, but not vice versa.

559 ***Provide a factory method rather than a simple constructor***
560 Use a factory method (routine) when you need to create objects based on a type
561 code or when you want to work with reference objects rather than value objects.

562 ***Replace error codes with exceptions or vice versa***
563 Depending on your error-handling strategy, make sure the code is using the
564 standard approach.

CHECKLIST: Summary of Refactorings

Data Level Refactorings

- Replace a magic number with a named constant.
- Rename a variable with a clearer or more informative name.
- Move an expression inline.
- Replace an expression with a routine.
- Introduce an intermediate variable.
- Convert a multi-use variable to a multiple single-use variables.
- Use a local variable for local purposes rather than a parameter.
- Convert a data primitive to a class.
- Convert a set of type codes to a class.
- Convert a set of type codes to a class with subclasses.
- Change an array to an object.
- Encapsulate a collection.
- Replace a traditional record with a data class.

Statement Level Refactorings

- Decompose a boolean expression.
- Move a complex boolean expression into a well-named boolean function.
- Consolidate fragments that are duplicated within different parts of a conditional.
- Use break or return instead of a loop control variable.
- Return as soon as you know the answer instead of assigning a return value within nested if-then-else statements.
- Replace conditionals with polymorphism (especially repeated case statements).
- Create and use null objects instead of testing for null values.

Routine Level Refactorings

- Extract a routine.
- Move a routine's code inline.
- Convert a long routine to a class.
- Substitute a simple algorithm for a complex algorithm.
- Add a parameter.
- Remove a parameter.
- Separate query operations from modification operations.

- 599 Combine similar routines by parameterizing them.
600 Separate routines whose behavior depends on parameters passed in.
601 Pass a whole object rather than specific fields.
602 Pass specific fields rather than a whole object.
603 Encapsulate downcasting.

604 **Class Implementation Refactorings**

- 605 Change value objects to reference objects.
606 Change reference objects to value objects.
607 Replace virtual routines with data initialization.
608 Change member routine or data placement.
609 Extract specialized code into a subclass.
610 Combine similar code into a superclass.

611 **Class Interface Refactorings**

- 612 Move a routine to another class.
613 Convert one class to two.
614 Eliminate a class.
615 Hide a delegate.
616 Replace inheritance with delegation.
617 Replace delegation with inheritance.
618 Remove a middle man.
619 Introduce a foreign routine.
620 Introduce a class extension.
621 Encapsulate an exposed member variable.
622 Remove *Set()* routines for fields that cannot be changed.
623 Hide routines that are not intended to be used outside the class.
624 Encapsulate unused routines.
625 Collapse a superclass and subclass if their implementations are very similar.

626 **System Level Refactorings**

- 627 Duplicate data you can't control.
628 Change unidirectional class association to bidirectional class association.
629 Change bidirectional class association to unidirectional class association.
630 Provide a factory routine rather than a simple constructor.
631 Replace error codes with exceptions or vice versa.
-

633 *Opening up a working system is more like*
634 *opening up a human*
635 *brain and replacing a nerve than opening up a*
636 *sink and replacing a washer. Would*
637 *maintenance be easier if*
638 *it was called “Software*
639 *Brain Surgery?”*
640 —Gerald Weinberg

641
642
643
644
645
646

647
648
649
650

651
652
653
654

655
656
657
658
659
660

661
662
663
664
665

24.5 Refactoring Safely

Refactoring is a powerful technique for improving code quality. A few simple guidelines can make this powerful technique even more effective.

Keys to Refactoring Safely

Save the code you start with

Before you begin refactoring, make sure you can get back to the code you started with. Save a version in your revision control system, or copy the correct files to a backup directory.

Keep refactorings small

Some refactorings are larger than others, and exactly what constitutes “one refactoring” can be a little fuzzy. Keep the refactorings small so that you fully understand all the impacts of the changes you make. The detailed refactorings described in *Refactoring* (Fowler 1999) provide many good examples of how to do this.

Do refactorings one at a time

Some refactorings are more complicated than others. For all but the simplest refactorings, do the refactorings one at a time, recompile, and retest, then do the next refactoring.

Make a list of steps you intend to take

A natural extension of the Pseudocode Programming Process is to make a list of the refactorings that will get you from Point A to Point B. Making a list helps you keep each change in context.

Make a parking lot

When you’re midway through one refactoring, you’ll sometimes find that you need another refactoring. Midway through that refactoring, you find a third refactoring that would be beneficial. For changes that aren’t needed immediately, make a “parking lot”—a list of the changes that you’d like to make at some point, but that don’t need to be made right now.

Make frequent checkpoints

It’s easy to suddenly find the code going sideways when refactoring. In addition to saving the code you started with, save checkpoints at various steps in a refactoring session so that you can get back to a working program if you code yourself into a dead end.

666
667
668
669

Use your compiler warnings

It's easy to make small errors that slip past the compiler. Setting your compiler to the pickiest warning level possible will help catch many errors almost as soon as you type them.

670
671
672

Retest

Reviews of changed code should be complemented by retests. Regression testing is described in more detail in Chapter TBD, "Developer Testing."

673
674
675

Add test cases

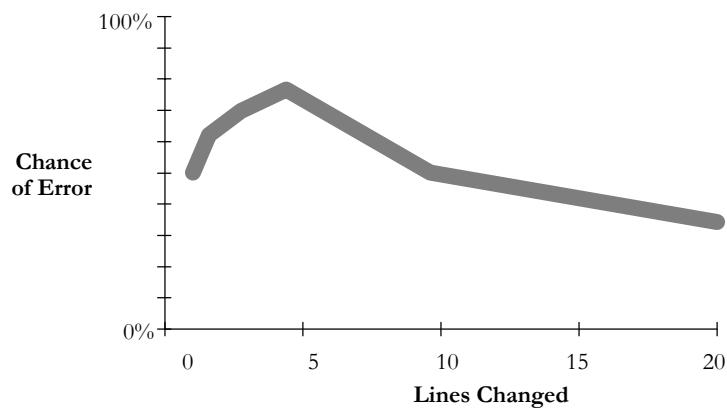
In addition to retesting with your old tests, add new unit tests to exercise the new code.

676 **CROSS-REFERENCE** For details on reviews, see
677 Chapter 21, "Collaborative
678 Construction."
679

680
681
682
683
684

Review the changes

If reviews are important the first time through, they are even more important during subsequent modifications. Ed Yourdon reports that when programmers make changes to a program, they typically have more than a 50 percent chance of making an error the first time (Yourdon 1986b). Interestingly, if programmers work with a substantial portion of the code, rather than just a few lines, the chance of making a correct modification improves. Specifically, as the number of lines changed increases from one to five lines, the chance of making a bad change increases. After that, the chance of making a bad change decreases.



685
686
687
688

F24xx01

Figure 24-1

Small changes tend to be more error prone than larger changes (Weinberg 1983).

689
690
691

Programmers treat small changes casually. They don't desk-check them, they don't have others review them, and they sometimes don't even run the code to verify that the fix works properly.

692 HARD DATA693
694
695696
697
698
699
700
701
702703
704
705706
707
708

709 **Do not partially write a
710 feature with the intent of
711 refactoring to get it
712 complete later.**
713 —John Manzo
714

715 **A big refactoring is a
716 recipe for disaster.**
717 —Kent Beck
718
719

720

The moral is simple. Treat simple changes as if they were complicated. One organization that introduced reviews for one-line changes found that its error rate went from 55 percent before reviews to 2 percent afterward (Freedman and Weinberg 1982).

Adjust your approach depending on the risk level of the refactoring

Some refactorings are riskier than others. A refactoring like “Replace a magic number with a named constant” is relatively risk free. Refactorings that involve class or routine interface changes, database schema changes, changes to boolean tests, among others, tend to be more risky. For easier refactorings, you might streamline your refactoring process to do more than one refactoring at a time and to simply retest, without going through an official review.

For riskier refactorings, err on the side of caution. Do the refactorings one at a time. Have someone else review the refactoring or use pair programming for that refactoring, in addition to the normal compiler checking and unit tests.

Bad Times to Refactor

Refactoring is a powerful technique, but it isn’t a panacea, and it is subject to a few specific kinds of abuse.

Don’t use refactoring as a cover for code and fix

The worst problem with refactoring is how it’s misused. Programmers will sometimes say they’re refactoring, when all they’re really doing is tweaking the code, hoping to find a way to make it work. Refactoring refers to *changes in working code* that do not affect the program’s behavior. Programmers who are tweaking broken code aren’t refactoring; they’re hacking.

Avoid refactoring instead of rewriting

Sometimes code doesn’t need small changes—it needs to be tossed out so you can start over. If you find yourself in a major refactoring session, ask if you should just be redesigning and reimplementing that section of code from the ground up instead.

24.6 Refactoring Strategies

The number of refactorings that would be beneficial to any specific program is essentially infinite. Refactoring is subject to the same law of diminishing returns as other programming activities, and the 80/20 rule applies. Spend your time on the 20 percent of the refactorings that provide 80 percent of the benefit. Here are some ways you can decide which refactorings are most important.

726 ***Refactor when you add a routine***

727 When you add a routine, check whether related routines are well organized. If
728 not, refactor them.

729 ***Refactor when you add a class***

730 Adding a class often brings issues with existing code to the fore. Use this time as
731 an opportunity to refactor other classes that are closely related to the class you're
732 adding.

733 ***Refactor when you fix a defect***

734 Use the understanding you gain from fixing a bug to improve other code that
735 might be prone to similar defects.

736 **CROSS-REFERENCE** For
737 more on error-prone code,
738 see "Which Classes Contain
739 the Most Errors?" in Section
22.4.

740
741

742 ***Target error-prone modules***

743 Some modules are more error prone and brittle than others. Is there a section of
744 code that you and everyone else on your team is afraid of? That's probably an
745 error prone module. Although most people's natural tendency is to avoid these
746 challenging sections of code, targeting these sections for refactoring can be one
747 of the more effective strategies (Jones 2000).

742 ***Target high complexity modules***

743 Another approach is to focus on modules that have the highest complexity
744 ratings. (See "How to Measure Complexity" in Section 19.6 for details on these
745 metrics.) One classic study found that program quality improved dramatically
746 when maintenance programmers focused their improvement efforts on the
747 modules that had the highest complexity (Henry and Kafura 1984).

748 ***In a maintenance environment, improve the parts you touch***

749 Code that is never modified doesn't need to be refactored. But when you do
750 touch a section of code, be sure you leave it better than you found it.

751 ***Define an interface between clean code and ugly code, and then move code
752 across the interface***

753 The "real world" is often messier than you'd like. The messiness might come
754 from complicated business rules, hardware interfaces, or software interfaces. A
755 common problem with geriatric systems is poorly written production code that
756 must remain operational at all times.

757 An effective strategy for rejuvenating geriatric production systems is to
758 designate some code as being in the messy real world, some code as being in an
759 idealized new world, and some code as being the interface between the two.
760 Figure 24-2 shows this idea graphically.

761 **Error! Objects cannot be created from editing field codes.**

762 **F24xx02**

763 **Figure 24-2**

764 *Your code doesn't have to be messy just because the real world is messy. Conceive
765 your system as a combination of ideal code, interfaces from the ideal code to the
766 messy real world, and the messy real world.*

767 As you work with the system, you can begin moving code across the "real world
768 interface" into a more organized ideal world. When you begin working with a
769 legacy system, the poorly written legacy code might make up nearly all the
770 system. One policy that works well is that, anytime you touch a section of messy
771 code, you are required to bring it up to current coding standards, give it clear
772 variable names, and so on—effectively moving it into the ideal world. Over time
773 this can provide for a rapid improvement in a code base, as shown in Figure
774 TBD-3.

775 **Error! Objects cannot be created from editing field codes.**

776 **F24xx03**

777 **Figure 24-3**

778 *One strategy for improving production code is to refactor poorly written legacy code
779 as you touch it and move it to the other side of the "interface to the messy real
780 world."*

CC2E.COM/2457

781 **CHECKLIST: Refactoring Safely**

- 782 Is each change part of a systematic change strategy?
- 783 Did you save the code you started with before beginning refactoring?
- 784 Are you keeping each refactoring small?
- 785 Are you doing refactorings one at a time?
- 786 Have you made a list of steps you intend to take during your refactoring?
- 787 Do you have a parking lot so that you can remember ideas that occur to you
788 mid-refactoring?
- 789 Have you retested after each refactoring?
- 790 Have changes been reviewed if they are complicated or if they affect
791 mission-critical code?
- 792 Have you considered the riskiness of the specific refactoring, and adjusted
793 your approach accordingly?
- 794 Does the change enhance the program's internal quality rather than
795 degrading it?

- 796 Have you avoided using refactoring as a cover for code and fix or as an
797 excuse for not rewriting bad code?

798

CC2E.COM/2464

799

Additional Resources

800 Fowler, Martin. *Refactoring: Improving the Design of Existing Code*, Reading,
801 Mass.: Addison Wesley, 1999. This is the definitive guide to refactoring. It
802 contains detailed discussions of many of the specific refactorings that I
803 summarized in this chapter as well as a handful of other refactorings that I didn't
804 summarize in this chapter. Fowler provides numerous code samples to illustrate
805 how each refactoring is performed step by step.

806 The process of refactoring has a lot in common with the process of fixing
807 defects. For more on fixing defects, see Section 23.3, "Fixing a Defect." The
808 risks associated with refactoring are similar to the risks associated with code
809 tuning. For more on managing code-tuning risks, see Section 25.6, "Summary of
810 the Approach to Code Tuning."

811

Key Points

- 812 • Program changes are a fact of life both during initial development and after
813 initial release.
- 814 • Software can either improve or degrade as it's changed. The Cardinal Rule
815 of Software Evolution is that internal quality should improve with age.
- 816 • One key to success in refactoring is learning to pay attention to the
817 numerous warning signs or smells that indicate a need to refactor.
- 818 • Another key to success is learning numerous specific refactorings.
- 819 • A final key to success is having a strategy for refactoring safely. Some
820 approaches to refactoring are better than others.
- 821 • Refactoring during development is the best chance you'll get to improve
822 your program, to make all the changes you'll wish you'd made the first time.
823 Take advantage of it!

25

Code-Tuning Strategies

Contents

- 25.1 Performance Overview
- 25.2 Introduction to Code Tuning
- 25.3 Kinds of Fat and Molasses
- 25.4 Measurement
- 25.5 Iteration
- 25.6 Summary of the Approach to Code Tuning

Related Topics

Code-tuning techniques: Chapter 29

Software architecture: Section 3.5

THIS CHAPTER DISCUSSES THE QUESTION of performance tuning—historically, a controversial issue. Computer resources were severely limited in the 1960s, and efficiency was a paramount concern. As computers became more powerful in the 1970s, programmers realized how much their focus on performance had hurt readability and maintainability, and code tuning received less attention. The return of performance limitations with the microcomputer revolution of the 1980s again brought efficiency to the fore, which then waned throughout the 1990s. In the 2000s, memory limitations in embedded software for devices such as telephones and PDAs, and the execution time of interpreted code have once again made efficiency a key topic.

You can address performance concerns at two levels: strategic and tactical. This chapter addresses strategic performance issues: what performance is, how important it is, and the general approach to achieving it. If you already have a good grip on performance strategies and are looking for specific code-level techniques that improve performance, move on to the next chapter. Before you begin any major performance work, however, at least skim the information in this chapter so that you don't waste time optimizing when you should be doing other kinds of work.

31

32 *More computing sins are
33 committed in the name of
34 efficiency (without neces-
35 sarily achieving it) than
36 for any other single rea-
son—including blind
37 stupidity.*

38 —W.A. Wulf

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61 **KEY POINT**

62

63

64

25.1 Performance Overview

Code tuning is one way of improving a program's performance. You can often find other ways to improve performance more, in less time and with less harm to the code, than by code tuning. This section describes the options.

Quality Characteristics and Performance

Some people look at the world through rose-colored glasses. Programmers like you and me tend to look at the world through code-colored glasses. We assume that the better we make the code, the more our clients and customers will like our software.

This point of view might have a mailing address somewhere in reality, but it doesn't have a street number, and it certainly doesn't own any real estate. Users are more interested in tangible program characteristics than they are in code quality. Sometimes users are interested in raw performance, but only when it affects their work. Users tend to be more interested in program throughput than raw performance. Delivering software on time, providing a clean user interface, and avoiding downtime are often more significant.

Here's an illustration: I take at least 50 pictures a week on my digital camera. To upload the pictures to my computer, the software that came with the camera requires me to select each picture one by one, viewing them in a window that shows only 6 pictures at a time. Uploading 50 pictures is a tedious process that required dozens of mouse clicks and lots of navigation through the 6-picture window. After putting up with this for a few months, I bought a memory-card reader that plugs directly into my computer and that my computer thinks is a disk drive. Now I can use Windows Explorer to copy the pictures to my computer. What used to take dozens of mouse clicks and lots of waiting now requires about two mouse clicks, a CTRL+A, and a drag and drop.

I really don't care whether the memory card reader transfers each file in half the time or twice the time as the other software, because my throughput is faster. Regardless of whether the memory card reader's code is faster or slower, its performance is better.

Performance is only loosely related to code speed. To the extent that you work on your code's speed, you're not working on other quality characteristics. Be wary of sacrificing other characteristics in order to make your code faster. Your work on speed may hurt performance rather than help it.

65 Performance and Code Tuning

66 Once you've chosen efficiency as a priority, whether its emphasis is on speed or
67 on size, you should consider several options before choosing to improve either
68 speed or size at the code level. Think about efficiency from each of these view-
69 points:

- 70 • Program requirements
71 • System design
72 • Class and routine design
73 • Operating-system interactions
74 • Code compilation
75 • Hardware
76 • Code tuning

77 Program Requirements

78 Performance is stated as a requirement far more often than it actually is a re-
79 quirement. Barry Boehm tells the story of a system at TRW that initially required
80 sub-second response time. This requirement led to a highly complex design and
81 an estimated cost of \$100 million. Further analysis determined that users would
82 be satisfied with four-second responses 90 percent of the time. Modifying the
83 response-time requirement reduced overall system cost by about \$70 million.
84 (Boehm 2000b).

85 Before you invest time solving a performance problem, make sure that you're
86 solving a problem that needs to be solved.

87 Program Design

88 This level includes the major strokes of the design for a single program, mainly
89 the way in which a program is divided into classes. Some program designs make
90 it difficult to write a high-performance system. Others make it hard not to.

91 Consider the example of a real-world data-acquisition program for which the
92 high-level design had identified measurement throughput as a key product attrib-
93 ute. Each measurement included time to make an electrical measurement, cali-
94 brate the value, scale the value, and convert it from sensor data units (such as
95 millivolts) into engineering data units (such as degrees).

96 In this case, without addressing the risk in the high-level design, the program-
97 mers would have found themselves trying to optimize the math to evaluate a
98 13th-order polynomial in software—that is, a polynomial with 14 terms includ-

99
100
101
102
103

104 **CROSS-REFERENCE** For
105 details on the way program-
106 mers work toward objectives,
107 see “Setting Objectives” in
Section 20.2.

108
109
110
111

112
113
114

115 **KEY POINT**

116
117
118
119
120

121

122 **CROSS-REFERENCE** For
123 more information about data
124 types and algorithms, see the
125 “Additional Resources” sec-
tion at the end of the chapter.

126

127 **CROSS-REFERENCE** For
128 code-level strategies for deal-
129 ing with slow or fat operat-
130 ing-system routines, see
131 Chapter 26, “Code-Tuning
Techniques.”

132

ing variables raised to the 13th power. Instead, they addressed the problem with different hardware and a high-level design that used dozens of 3rd-order polynomials. This change could not have been effected through code tuning, and it’s unlikely that any amount of code tuning would have solved the problem. This is an example of a problem that had to be addressed at the program-design level.

If you know that a program’s size and speed are important, design the program’s architecture so that you can reasonably meet your size and speed goals. Design a performance-oriented architecture, and then set resource goals for individual subsystems, features, and classes. This will help in several ways:

- Setting individual resource goals makes the system’s ultimate performance predictable. If each feature meets its resource goals, the whole system will meet its goals. You can identify subsystems that have trouble meeting their goals early and target them for redesign or code tuning.
- The mere act of making goals explicit improves the likelihood that they’ll be achieved. Programmers work to objectives when they know what they are; the more explicit the objectives, the easier they are to work to.
- You can set goals that don’t achieve efficiency directly but promote efficiency in the long run. Efficiency is often best treated in the context of other issues. For example, achieving a high degree of modifiability can provide a better basis for meeting efficiency goals than explicitly setting an efficiency target. With a highly modular, modifiable design, you can easily swap less-efficient components for more-efficient ones.

Class and Routine Design

Designing the internals of classes and routines presents another opportunity to design for performance. One key to performance that comes into play at this level is the choice of data types and algorithms, which usually affect both the memory use and the execution speed of a program.

Operating-System Interactions

If your program works with external files, dynamic memory, or output devices, it’s probably interacting with the operating system. If performance isn’t good, it might be because the operating-system routines are slow or fat. You might not be aware that the program is interacting with the operating system; sometimes your compiler generates system calls or your libraries invoke system calls you would never dream of. More on this later.

133

134 **CROSS-REFERENCE** The
135 optimization results reported
136 in Chapter 26, “Code-Tuning
Techniques,” provide numer-
137 ous examples of compiler
optimizations that produce
138 more efficient code than
139 manual code tuning does.

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

Code Compilation

Good compilers turn clear, high-level language code into optimized machine code. If you choose the right compiler, you might not need to think about optimizing speed any further.

Hardware

Sometimes the cheapest and best way to improve a program’s performance is to buy new hardware. If you’re distributing a program for nationwide use by hundreds of thousands of customers, buying new hardware isn’t a realistic option. But if you’re developing custom software for a few in-house users, a hardware upgrade might be the cheapest option. It saves the cost of initial performance work. It saves the cost of future maintenance problems caused by performance work. It improves the performance of every other program that runs on that hardware too.

Code Tuning

Code tuning is the practice of modifying correct code in ways that make it run more efficiently, and it is the subject of the rest of this chapter. “Tuning” refers to small-scale changes that affect a single class, a single routine, or, more commonly, a few lines of code. “Tuning” does not refer to large-scale design changes, or other higher-level means of improving performance.

You can make dramatic improvements at each level from system design through code tuning. Jon Bentley cites an argument that in some systems, the improvements at each level can be multiplied (1982). Since you can achieve a 10-fold improvement in each of six levels, that implies a potential performance improvement of a million fold. Although such a multiplication of improvements requires a program in which gains at one level are independent of gains at other levels, which is rare, the potential is inspiring.

25.2 Introduction to Code Tuning

What is the appeal of code tuning? It’s not the most effective way to improve performance. Program architecture, class design, and algorithm selection usually produce more dramatic improvements. Nor is it the easiest way to improve performance. Buying new hardware or a compiler with a better optimizer is easier. It’s not the cheapest way to improve performance either. It takes more time to hand-tune code initially, and hand-tuned code is harder to maintain later.

Code tuning is appealing for several reasons. One attraction is that it seems to defy the laws of nature. It’s incredibly satisfying to take a routine that executes

168 in 20 microseconds, tweak a few lines, and reduce the execution speed to 2 mi-
169 croseconds.

170 It's also appealing because mastering the art of writing efficient code is a rite of
171 passage to becoming a serious programmer. In tennis, you don't get any points
172 for the way you pick up a tennis ball, but you still need to learn the right way to
173 do it. You can't just lean over and pick it up with your hand. If you're good, you
174 whack it with the head of your racket until it bounces waist high and then you
175 catch it. Whacking it more than three times or not bouncing it the first time are
176 both serious failings. It doesn't really matter how you pick up a tennis ball, but
177 within the tennis culture the way you pick it up carries a certain cachet. Simi-
178 larly, no one but you and other programmers usually cares how tight your code
179 is. Nonetheless, within the programming culture, writing micro-efficient code
180 proves you're cool.

181 The problem with code tuning is that efficient code isn't necessarily "better"
182 code. That's the subject of the next few subsections.

183 **The Pareto Principle**

184 The Pareto Principle, also known as the 80/20 rule, states that you can get 80
185 percent of the result with 20 percent of the effort. The principle applies to a lot of
186 areas other than programming, but it definitely applies to program optimization.

187 **KEY POINT**
188
189
190 Barry Boehm reports that 20 percent of a program's routines consume 80 percent
of its execution time (1987b). In his classic paper "An Empirical Study of For-
tran Programs," Donald Knuth found that less than 4 percent of a program usu-
ally accounts for more than 50 percent of its run time (1971).

191 Knuth used a line-count profiler to discover this surprising relationship, and the
192 implications for optimization are clear. You should measure the code to find the
193 hot spots and then put your resources into optimizing the few percent that are
194 used the most. Knuth profiled his line-count program and found that it was
195 spending half its execution time in two loops. He changed a few lines of code
196 and doubled the speed of the profiler in less than an hour.

197 Jon Bentley describes a case in which a thousand-line program spent 80 percent
198 of its time in a five-line square-root routine. By tripling the speed of the square-
199 root routine, he doubled the speed of the program (1988).

200 Bentley also reports the case of a team who discovered that half an operating
201 system's time was spent in a small loop. They rewrote the loop in microcode and
202 made the loop 10 times faster, but it didn't change the system's performance—
203 they had rewritten the system's idle loop!

204
205
206
207
208

The team who designed the ALGOL language—the granddaddy of most modern languages and one of the most influential languages ever—received the following advice: “The best is the enemy of the good.” Working toward perfection may prevent completion. Complete it first, and then perfect it. The part that needs to be perfect is usually small.

209

Old Wives’ Tales

210
211

Much of what you’ve heard about code tuning is false. Here are some common misapprehensions:

212
213
214
215
216

Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code—false!

Many programmers cling tenaciously to the belief that if they can write code in one or two lines, it will be the most efficient possible. Consider the following code that initializes a 10-element array:

217 **CROSS-REFERENCE** Both
218 these code fragments violate
219 several rules of good pro-
220 gramming. Readability and
221 maintenance are usually more
important than execution
222 speed or size, but in this
223 chapter the topic is perform-
224 ance, and that implies a trade-
225 off with the other objectives.
226 You’ll see many examples of
227 coding practices here that
228 aren’t recommended in other
229 parts of this book.
230
231

232
233
234
235

If you follow the old “fewer lines are faster” dogma, you’ll guess that the first code is faster because it has four fewer lines. Hah! Tests in Visual Basic and Java have shown that the second fragment is at least 60 percent faster than the first. Here are the numbers:

236
237
238
239

Language	for-Loop Time	Straight-Code Time	Time Savings	Performance Ratio
Visual Basic	8.47	3.16	63%	2.5:1
Java	12.6	3.23	74%	4:1

Note: (1) Times in this and the following tables in this chapter are given in seconds and are meaningful only for comparisons across rows in each table. Actual times will vary according to the compiler and compiler options used and the environment in which each test is run. (2) Benchmark results are typically made up of several

240 *thousand to many million executions of the code fragments to smooth out sample-to-*
241 *sample fluctuations in the results.* (3) *Specific brands and versions of compilers*
242 *aren't indicated. Performance characteristics vary significantly from brand to brand*
243 *and from version to version.* (4) *Comparisons among results from different lan-*
244 *guages aren't always meaningful because compilers for different languages don't*
245 *always offer comparable code-generation options.* (5) *The results shown for inter-*
246 *preted languages (PHP and Python) are typically based on less than 1% of the test*
247 *runs used for the other languages.* (6) *Some of the "time savings" percentages might*
248 *not be exactly reproducible from the data in these tables due to rounding of the*
249 *"straight time" and "code-tuned time" entries.*

250 This certainly doesn't imply the conclusion that increasing the number of lines of
251 high-level language code always improves speed or reduces size. It does imply
252 that regardless of the aesthetic appeal of writing something with the fewest lines
253 of code, there's no predictable relationship between the number of lines of code
254 in a high-level language and a program's ultimate size and speed.

255 ***Certain operations are probably faster or smaller than others—false!***
256 There's no room for "probably" when you're talking about performance. You
257 must always measure performance to know whether your changes helped or hurt
258 your program. The rules of the game change every time you change languages,
259 compilers, versions of compilers, libraries, versions of libraries, processor,
260 amount of memory on the machine, color of shirt you're wearing and so. (These
261 are all serious except the last one.) What was true on one machine with one set
262 of tools can easily be false on another machine with a different set of tools.

263 This phenomenon suggests several reasons not to improve performance by code
264 tuning. If you want your program to be portable, techniques that improve per-
265 formance in one environment can degrade it in others. If you change compilers
266 or upgrade, the new compiler might automatically optimize code the way you
267 were hand-tuning it, and your work will have been wasted. Even worse, your
268 code tuning might defeat more powerful compiler optimizations that have been
269 designed to work with straightforward code.

270 When you tune code, you're implicitly signing up to reprofile each optimization
271 every time you change your compiler brand, compiler version, library version,
272 and so. If you don't reprofile, an optimization that improves performance under
273 one version of a compiler or library might well degrade performance when you
274 change the build environment.

275 **We should forget about
276 small efficiencies, say
277 about 97% of the time:
278 premature optimization is
279 the root of all evil.**
280 —Donald Knuth

281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300

301
302
303

304
305
306
307

308
309
310
311
312

You should optimize as you go—false!

One theory is that if you strive to write the fastest and smallest possible code as you write each routine, your program will be fast and small. This approach creates a forest-for-the-trees situation in which programmers ignore significant global optimizations because they're too busy with micro-optimizations. Here are the main problems with optimizing as you go along:

- It's almost impossible to identify performance bottlenecks before a program is working completely. Programmers are very bad at guessing which 4 percent of the code accounts for 50 percent of the execution time, and so programmers who optimize as they go will, on average, spend 96 percent of their time optimizing code that doesn't need to be optimized. That leaves very little time to optimize the 4 percent that really counts.
- In the rare case in which developers identify the bottlenecks correctly, they overkill the bottlenecks they've identified and allow others to become critical. Again, the ultimate effect is a reduction in performance. Optimizations done after a system is complete can identify each problem area and its relative importance so that optimization time is allocated effectively.
- Focusing on optimization during initial development detracts from achieving other program objectives. Developers immerse themselves in algorithm analysis and arcane debates that in the end don't contribute much value to the user. Concerns such as correctness, information hiding, and readability become secondary goals, even though performance is easier to improve later than these other concerns are. Post-hoc performance work typically affects less than 5 percent of a program's code. Would you rather go back and do performance work on 5 percent of the code or readability work on 100 percent?

In short, premature optimization's primary drawback is its lack of perspective. Its victims include final code speed, performance attributes that are more important than code speed, program quality, and ultimately the software's users.

If the development time saved by implementing the simplest program is devoted to optimizing the running program, the result will always be a faster-running program than one in which optimization efforts have been exerted indiscriminately as the program was developed (Stevens 1981).

In an occasional project, post-hoc optimization won't be sufficient to meet performance goals, and you'll have to make major changes in the completed code. In those cases, small, localized optimizations wouldn't have provided the gains needed anyway. The problem in such cases isn't inadequate code quality—it's inadequate software architecture.

313
314
315
316
317

318 **FURTHER READING** For
319 many other entertaining and
320 enlightening anecdotes, see
321 Gerald Weinberg's *Psychol-*
322 *ogy of Computer Program-*
323 *ming* (1998).

324
325
326
327
328
329

330
331
332

333

334
335
336

337
338

339
340
341
342
343

If you need to optimize before a program is complete, minimize the risks by building perspective into your process. One way is to specify size and speed goals for features and then optimize to meet the goals as you go along. Setting such goals in a specification is a way to keep one eye on the forest while you figure out how big your particular tree is.

A fast program is just as important as a correct one—false!

It's hardly ever true that programs need to be fast or small before they need to be correct. Gerald Weinberg tells the story of a programmer who was flown to Detroit to help debug a troubled program. The programmer worked with the team who had developed the program and concluded after several days that the situation was hopeless.

On the flight home, he mulled over the situation and realized what the problem was. By the end of the flight, he had an outline for the new code. He tested the code for several days and was about to return to Detroit when he got a telegram saying that the project had been cancelled because the program was impossible to write. He headed back to Detroit anyway and convinced the executives that the project could be completed.

Then he had to convince the project's original programmers. They listened to his presentation, and when he'd finished, the creator of the old system asked, "And how long does your program take?"

"That varies, but about ten seconds per input."

"Aha! But my program takes only one second per input." The veteran leaned back, satisfied that he'd stumped the upstart. The other programmers seemed to agree, but the new programmer wasn't intimidated.

"Yes, but your program *doesn't work*. If mine doesn't have to work, I can make it run instantly."

For a certain class of projects, speed or size is a major concern. This class is the minority, is much smaller than most people think, and is getting smaller all the time. For these projects, the performance risks must be addressed by up-front design. For other projects, early optimization poses a significant threat to overall software quality, *including performance*.

344

345 **Jackson's Rules of Opti-**
346 **mization: Rule 1. Don't**
347 **do it. Rule 2 (for experts**
348 **only). Don't do it yet—**
349 **that is, not until you have**
350 **a perfectly clear and un-**
351 **optimized solution.**

351 —M. A. Jackson

352

353

354

355

356

357

358

359

360

361

362

363

364

365

366

367

368

369

370

371

372

373

374

375

376

377

378

379

380

When to Tune

Use a high-quality design. Make the program right. Make it modular and easily modifiable so that it's easy to work on later. When it's complete and correct, check the performance. If the program lumbers, make it fast and small. Don't optimize until you know you need to.

A few years ago I worked on a C++ project that produced graphical outputs to analyze investment data. After my team got the first graph working, testing reported that the program took about 45 minutes to draw the graph, which was clearly not acceptable. We held a team meeting to decide what to do about it. One of the developers became irate and shouted, "If we want to have any chance of releasing an acceptable product we've got to start rewriting the whole code base in assembler *right now*." I responded that I didn't think so—that 4 percent of the code probably accounted for 50 percent or more of the performance bottleneck. It would be best to address that 4 percent toward the end of the project. After a bit more shouting, our manager assigned me to do some initial performance work (which was really a case of "Oh no! Please don't throw me into that briar patch!").

As is often the case, a day's work identified a couple of glaring bottlenecks in the code, and a small number of code-tuning changes reduced the drawing time from 45 minutes to less than 30 seconds. Far less than 1 percent of the code accounted for 90 percent of the run time. By the time we released the software months later, several additional code-tuning changes reduced that drawing time to a little more than 1 second.

Compiler Optimizations

Modern compiler optimizations might be more powerful than you expect. In the case I described earlier, my compiler did as good a job of optimizing a nested loop as I was able to do by rewriting the code in a supposedly more efficient style.

When shopping for a compiler, compare the performance of each compiler on your program. Each compiler has different strengths and weaknesses, and some will be better suited to your program than others.

Optimizing compilers are better at optimizing straightforward code than they are at optimizing tricky code. If you do "clever" things like fooling around with loop indexes, your compiler has a harder time doing its job and your program suffers. See "Using Only One Statement per Line" in Section 31.5, for an example in which a straightforward approach resulted in compiler-optimized code that was 11 percent faster than comparable "tricky" code.

381 With a good optimizing compiler, your code speed can improve 40 percent or
 382 more across the board. Many of the techniques described in the next chapter pro-
 383 duce gains of only 15-30 percent. Why not just write clear code and let the com-
 384 piler do the work? Here are the results of a few tests to check how much an
 385 optimizer speeded up an insertion-sort routine:

Language	Time Without Compiler Optimizations	Time with Compiler Optimizations	Time Savings	Perform-ance Ra-tio
C++ compiler 1	2.21	1.05	52%	2:1
C++ compiler 2	2.78	1.15	59%	2.5:1
C++ compiler 3	2.43	1.25	49%	2:1
C# compiler	1.55	1.55	0%	1:1
Visual Basic	1.78	1.78	0%	1:1
Java VM 1	2.77	2.77	0%	1:1
Java VM 2	1.39	1.38	<1%	1:1
Java VM 3	2.63	2.63	0%	1:1

386 The only difference between versions of the routine was that compiler optimiza-
 387 tions were turned off for the first compile, on for the second. Clearly, some com-
 388 pilers optimize better than others, and some are better without optimizations in
 389 the first place. Some JVMs are also clearly better than others. You'll have to
 390 check your own compiler, JVM, or both to measure its effect.

391 25.3 Kinds of Fat and Molasses

392 In code tuning you find the parts of a program that are as slow as molasses in
 393 winter and as big as Godzilla and change them so that they run like greased
 394 lightning and are so skinny they can hide in the cracks between the other bytes in
 395 RAM. You always have to profile the program to know with any confidence
 396 which parts are slow and fat, but some operations have a long history of laziness
 397 and obesity, and you can start by investigating them.

398 Common Sources of Inefficiency

399 Here are several common sources of inefficiency:

400 *Input/output operations*

401 One of the most significant sources of inefficiency is unnecessary I/O. If you
 402 have a choice of working with a file in memory vs. on disk, in a database, or
 403 across a network, use an in-memory data unless space is critical.

404
405
406

Here's a performance comparison between code that accesses random elements
in a 100-element in-memory array and code that accesses random elements of
the same size in a 100-record disk file:

Language	External File Time	In-Memory Data Time	Time Savings	Performance Ratio
C++	6.04	0.000	100%	n/a
C#	12.8	0.010	100%	1000:1

407
408
409

According to this data, in-memory access is on the order of 1000 times faster
than accessing data in an external file. Indeed with the C++ compiler I used, the
time required for in-memory access wasn't measurable.

410
411

The performance comparison for a similar test of sequential access times is similar:

Language	External File Time	In-Memory Data Time	Time Savings	Performance Ratio
C++	3.29	0.021	99%	150:1
C#	2.60	0.030	99%	85:1

412
413

*The tests for sequential access were run with 13 times the data volume of the tests for
random access, so the results are not comparable across the two types of tests.*

414
415
416
417

If the test had used a slower medium for external access—hard disk across a
network connection—the difference would have been even greater. Here is what
the performance looks like when a similar random-access test is performed on a
network location instead of on the local machine:

Language	Local File Time	Network File Time	Time Savings
C++	6.04	6.64	-10%
C#	12.8	14.1	-10%

418
419
420
421

Of course these results can vary dramatically depending on the speed of your
network, network loading, distance of the local machine from the networked disk
drive, speed of the networked disk drive compared to the speed of the local
drive, current phase of the moon, and other factors.

422
423

Overall, the effect of in-memory access is significant enough to make you think
twice about having I/O in a speed-critical part of a program.

Paging

424
425
426
427

An operation that causes the operating system to swap pages of memory is much
slower than an operation that works on only one page of memory. Sometimes a
simple change makes a huge difference. In the next example, one programmer

428 wrote an initialization loop that produced many page faults on a system that used
429 4K pages.

430 Java Example of an Initialization Loop That Causes Many Page Faults

```
431 for ( column = 0; column < MAX_COLUMNS; column++ ) {  
432     for ( row = 0; row < MAX_ROWS; row++ ) {  
433         table[ row ][ column ] = BlankTableElement();  
434     }  
435 }
```

436 This is a nicely formatted loop with good variable names, so what's the problem?
437 The problem is that each element of *table* is about 4000 bytes long. If *table*
438 has too many rows, every time the program accesses a different row, the operating
439 system will have to switch memory pages. The way the loop is structured,
440 every single array access switches rows, which means that every single array
441 access causes paging to disk.

442 The programmer restructured the loop this way:

443 Java Example of an Initialization Loop That Causes Few Page Faults

```
444 for ( row = 0; row < MAX_ROWS; row++ ) {  
445     for ( column = 0; column < MAX_COLUMNS; column++ ) {  
446         table[ row ][ column ] = BlankTableElement();  
447     }  
448 }
```

449 This code still causes a page fault every time it switches rows, but it switches
450 rows only *MAX_ROWS* times instead of *MAX_ROWS * MAX_COLUMNS* times.

451 The specific performance penalty varies significantly. On a machine with limited
452 memory, I measured the second code sample to be about 1000 times faster than
453 the first code sample. On machines with more memory, I've measured the difference
454 to be as small as a factor of 2, and it doesn't show up at all except for very
455 large values of *MAX_ROWS* and *MAX_COLUMNS*.

456 *System calls*

457 Calls to system routines are often expensive. System routines include input/output
458 operations to disk, keyboard, screen, printer, or other device; memory-management
459 routines; and certain utility routines. If performance is an issue, find out how
460 expensive your system calls are. If they're expensive, consider these options:

- 462
- 463 • Write your own services. Sometimes you need only a small part of the functionality offered by a system routine and can build your own from lower-level system routines. Writing your own replacement gives you something that's faster, smaller, and better suited to your needs.

464

465

- 466 • Avoid going to the system.
467
468
469
470 • Work with the system vendor to make the call faster. Most vendors want to
improve their products and are glad to learn about parts of their systems with
weak performance. (They may seem a little grouchy about it at first, but they
really are interested.)

471 In the code tuning initiative I describe in the “When to Tune” Section, the pro-
472 gram used an *AppTime* class that was derived from a commercially available
473 *BaseTime* class. (These names have been changed to protect the guilty.) The
474 *AppTime* object was the most common object in this application, and we instan-
475 tiated tens of thousands of *AppTime* objects. After several months, we discov-
476 ered that *BaseTime* was initializing itself to the system time in its constructor.
477 For our purposes, the system time was irrelevant, which meant we were need-
478 lessly generating thousands of system-level calls. Simply overriding *BaseTime*’s
479 constructor and initializing the *time* field to 0 instead of to the system time gave
480 us about as much performance improvement as all the other changes we made
481 put together.

482 ***Interpreted languages***

483 Interpreted languages tend to exact significant performance penalties because
484 they must process each programming-language instruction before creating and
485 executing machine code. In the performance benchmarking I performed for this
486 chapter and Chapter 26, I observed the following approximate relationships in
487 performance among different languages:

488 **Table 25-1. Relative execution time of programming languages**

Language	Type of Lan-guage	Execution time relative to C++
C++	Compiled	1:1
Visual Basic	Compiled	1:1
C#	Compiled	1:1
Java	Byte code	1.5:1
PHP	Interpreted	>100:1
Python	Interpreted	>100:1

489 As you can see from the table, C++, Visual Basic, and C# are all comparable.
490 Java is close, but tends to be slower than the other languages. PHP and Python
491 are interpreted languages, and code in those languages tended to run a factor of
492 100 or more slower than code in C++, VB, C#, and Java.

493 The general numbers presented in this table must be viewed cautiously. For any
494 particular piece of code, C++, VB, C#, or Java might be twice as fast or half as
495 fast as the other languages. (You can see this for yourself in the detailed exam-
496 ples in Chapter 26.)

497
498
499
500
501**Errors**

A final source of performance problems is errors in the code. Errors can include leaving debugging code turned on (such as logging trace information to a file), forgetting to deallocate memory, improperly designing database tables, and so on.

502 A version 1.0 application I worked on had a particular operation that was much
 503 slower than other similar operations. A great deal of project mythology grew up
 504 to explain the slowness of this operation. We released version 1.0 without ever
 505 fully understanding why this particular operation was so slow. While working on
 506 the version 1.1 release, however, I discovered that the database table used by the
 507 operation wasn't indexed! Simply indexing the table improved performance by a
 508 factor of 30 for some operations. Defining an index on a commonly-used table is
 509 not optimization; it's just good programming practice.

510 **Relative Performance Costs of Common Operations**

511 Although you can't count on some operations being more expensive than others
 512 without measuring them, certain operations tend to be more expensive. When
 513 you look for the molasses in your program, use Table 25-2 to help make some
 514 initial guesses about the sticky parts of your program.

515 **Table 25-2. Costs of Common Operations**

Operation	Example	Relative Time Consumed	
		C++	Java
<i>Baseline (integer assignment)</i>	<i>i = j</i>	1	1
Routine Calls			
Call routine with no parameters	<i>foo()</i>	1	n/a
Call private routine with no parameters	<i>this.foo()</i>	1	0.5
Call private routine with 1 parameter	<i>this.foo(i)</i>	1.5	0.5
Call private routine with 2 parameters	<i>this.foo(i, j)</i>	1.7	0.5
Object routine call	<i>bar.foo()</i>	2	1
Derived routine call	<i>derivedBar.foo()</i>	2	1
Polymorphic routine call	<i>abstractBar.foo()</i>	2.5	2
Object References			
Level 1 object dereference	<i>i = obj.num</i>	1	1

Operation	Example	Relative Time Consumed	
		C++	Java
Level 2 object dereference	$i = obj1.obj2.num$	1	1
Each additional dereference	$i = obj1.obj2.obj3\dots$	not measurable	not measurable
Integer Operations			
Integer assignment (local)	$i = j$	1	1
Integer assignment (inherited)	$i = j$	1	1
Integer addition	$i = j + k$	1	1
Integer subtraction	$i = j - k$	1	1
Integer multiplication	$i = j * k$	1	1
Integer division	$i = j \% k$	5	1.5
Floating Point Operations			
Floating-point assignment	$x = y$	1	1
Floating-point addition	$x = y + z$	1	1
Floating-point subtraction	$x = y - z$	1	1
Floating-point multiplication	$x = y * z$	1	1
Floating-point division	$x = y / z$	4	1
Transcendental Functions			
Floating-point square root	$x = sqrt(y)$	15	4
Floating-point sine	$x = sin(y)$	25	20
Floating-point logarithm	$x = log(y)$	25	20
Floating-point e^x	$x = exp(y)$	50	20
Arrays			
Access integer array with constant subscript	$i = a[5]$	1	1
Access integer array with variable subscript	$i = a[j]$	1	1
Access two-dimensional integer array with constant subscripts	$i = a[3, 5]$	1	1
Access two-dimensional integer array with variable subscripts	$i = a[j, k]$	1	1
Access floating-point array with constant subscript	$x = z[5]$	1	1

Operation	Example	Relative Time Consumed	
		C++	Java
Access floating-point array with integer-variable subscript	$x = z[j]$	1	1
Access two-dimensional floating-point array with constant subscripts	$x = z[3, 5]$	1	1
Access two-dimensional floating-point array with integer-variable subscripts	$x = z[j, k]$	1	1

516
517
518

*Note: Measurements in this table are highly sensitive to local machine environment,
compiler optimizations, and code generated by specific compilers. Measurements
between C++ and Java are not directly comparable.*

519
520
521

Relative performance of these operations has changed significantly since the first edition of *Code Complete*, so if you're approaching code tuning with 10-year-old ideas about performance, you might need to update your thinking.

522
523
524
525

Most of the common operations are about the same price—routine calls, assignments, integer arithmetic, and floating-point arithmetic are all roughly equal. Transcendental math functions are extremely expensive. Polymorphic routine calls are a bit more expensive than other kinds of routine calls.

526
527
528
529

This table, or a similar one that you make, is the key that unlocks all the speed improvements described in Chapter 26, “Code-Tuning Techniques.” In every case, improving speed comes from replacing an expensive operation with a cheaper one. The next chapter provides examples of how to do so.

530 25.4 Measurement

531
532

Since small parts of a program usually consume a disproportionate share of the run time, measure your code to find the hot spots.

533
534
535
536
537

Once you've found the hot spots and optimized them, measure the code again to assess how much you've improved it. Many aspects of performance are counter-intuitive. The earlier case in this chapter, in which 10 lines of code were significantly faster and smaller than one line, is one example of the ways that code can surprise you.

538 | **KEY POINT**
539

Experience doesn't help much with optimization either. A person's experience might have come from an old machine, language, or compiler—and when any of

540 those things changes, all bets are off. You can never be sure about the effect of
541 an optimization until you measure the effect.

542 A few years ago I wrote a program that summed the elements in a matrix. The
543 original code looked like the next example.

C++ Example of Straightforward Code to Sum the Elements in a Matrix

```
544
545 sum = 0;
546 for ( row = 0; row < rowCount; row++ ) {
547     for ( column = 0; column < columnCount; column++ ) {
548         sum = sum + matrix[ row ][ column ];
549     }
550 }
```

551 This code was straightforward, but performance of the matrix-summation routine
552 was critical, and I knew that all the array accesses and loop tests had to be ex-
553 pensive. I knew from computer-science classes that every time the code accessed
554 a two-dimensional array, it performed expensive multiplications and additions.
555 For a 100-by-100 matrix, that totaled 10,000 multiplications and additions plus
556 the loop overhead. By converting to pointer notation, I reasoned, I could incre-
557 ment a pointer and replace 10,000 expensive multiplications with 10,000 rela-
558 tively cheap increment operations. I carefully converted the code to pointer nota-
559 tion and got this:

C++ Example of an Attempt to Tune Code to sum the Elements in a Ma- trix

```
560
561 sum = 0;
562 elementPointer = matrix;
563 lastElementPointer = matrix[ rowCount - 1 ][ columnCount - 1 ] + 1;
564 while ( elementPointer < lastElementPointer ) {
565     sum = sum + *elementPointer++;
566 }
```

567 Even though the code wasn't as readable as the first code, especially to pro-
568 FURTHER READING Jon
569 Bentley reported a similar
570 experience in which convert-
571 ing to pointers hurt perform-
572 ance by about 10 percent.
573 The same conversion had—in
another setting—improved
574 performance more than 50
percent. See “Software Ex-
ploratorium: Writing Effi-
cient C Programs” (Bentley
1991).

Even though the code wasn't as readable as the first code, especially to programmers who aren't C++ experts, I was magnificently pleased with myself. For a 100-by-100 matrix, I calculated that I had saved 10,000 multiplications and a lot of loop overhead. I was so pleased that I decided to measure the speed improvement, something I didn't always do back then, so that I could pat myself on the back more quantitatively.

574 **No programmer has ever
been able to predict or
analyze where performance
bottlenecks are without data. No matter
where you think it's going, you will be surprised
to discover that it is going somewhere else.**

582 —Joseph M. Newcomer

583

584

585

586 **CROSS-REFERENCE** For
587 a discussion of profiling
588 tools, see "Code Tuning" in
Section 30.3.

589

590

591

592

593

594

595

596

597

598

599

600

601

602

603

604

605

606

607

608

Do you know what I found?

No improvement whatsoever. Not with a 100-by-100 matrix. Not with a 10-by-10 matrix. Not with any size matrix. I was so disappointed that I dug into the assembly code generated by the compiler to see why my optimization hadn't worked. To my surprise, it turned out that I was not the first programmer who ever needed to iterate through the elements of an array—the compiler's optimizer was already converting the array accesses to pointers. I learned that the only result of optimization you can usually be sure of without measuring performance is that you've made your code harder to read. If it's not worth measuring to know that it's more efficient, it's not worth sacrificing clarity for a performance gamble.

Measurements Need to be Precise

Performance measurements need to be precise. Timing your program with a stopwatch or by counting "one elephant, two elephant, three elephant" isn't precise enough. Profiling tools are useful, or you can use your system's clock and routines that record the elapsed times for computing operations.

Whether you use someone else's tool or write your own code to make the measurements, make sure that you're measuring only the execution time of the code you're tuning. Use the number of CPU clock ticks allocated to your program rather than the time of day. Otherwise, when the system switches from your program to another program, one of your routines will be penalized for the time spent executing another program. Likewise, try to factor out measurement overhead so that neither the original code nor the tuning attempt is unfairly penalized.

25.5 Iteration

Once you've identified a performance bottleneck, you'll be amazed at how much you can improve performance by code tuning. You'll rarely get a 10-fold improvement from one technique, but you can effectively combine techniques; so keep trying, even after you find one that works.

I once wrote a software implementation of the Data Encryption Standard, or DES. Actually, I didn't write it once—I wrote it about 30 times. Encryption according to DES encodes digital data so that it can't be unscrambled without a password. The encryption algorithm is so convoluted that it seems like it's been used on itself. The performance goal for my DES implementation was to encrypt an 18K file in 37 seconds on an original IBM PC. My first implementation executed in 21 minutes and 40 seconds, so I had a long row to hoe.

609
 610 Even though most individual optimizations were small, cumulatively they were
 611 significant. To judge from the percentage improvements, no three or even four
 612 optimizations would have met my performance goal. But the final combination
 613 was effective. The moral of the story is that if you dig deep enough, you can
 make some surprising gains.

614 The code tuning I did in this case is the most aggressive code tuning I've ever
 615 done. At the same time, the final code is the most unreadable, unmaintainable
 616 code I've ever written. The initial algorithm is complicated. The code resulting
 617 from the high-level language transformation was barely readable. The translation
 618 to assembler produced a single 500-line routine that I'm afraid to look at. In gen-
 619 eral, this relationship between code tuning and code quality holds true. Here's a
 620 table that shows a history of the optimizations:

CROSS-REFERENCE The techniques listed in this table are described in Chapter 26, "Code-Tuning Techniques."

Optimization	Benchmark Time	Improvement
Implement initially—straightforward	21:40	—
Convert from bit fields to arrays	7:30	65%
Unroll innermost <i>for</i> loop	6:00	20%
Remove final permutation	5:24	10%
Combine two variables	5:06	5%
Use a logical identity to combine the first two steps of the DES algorithm	4:30	12%
Make two variables share the same memory to reduce data shuttling in inner loop	3:36	20%
Make two variables share the same memory to reduce data shuttling in outer loop	3:09	13%
Unfold all loops and use literal array subscripts	1:36	49%
Remove routine calls and put all the code in line	0:45	53%
Rewrite the whole routine in assembler	0:22	51%
Final	0:22	98%

621
 622 *Note: The steady progress of optimizations in this table doesn't imply that all optimi-*
 623 *zations work. I haven't shown all the things I tried that doubled the run time. At least*
two-thirds of the optimizations I tried didn't work.

624 **25.6 Summary of the Approach to Code Tun- 625 ing**

626 Here are the steps you should take as you consider whether code tuning can help
627 you improve the performance of a program:

- 628 1. Develop the software using well-designed code that's easy to understand and
629 modify.
- 630 2. If performance is poor,
 - 631 a. Save a working version of the code so that you can get back to the "last
632 known good state."
 - 633 b. Measure the system to find hot spots.
 - 634 c. Determine whether the weak performance comes from inadequate de-
635 sign, data types, or algorithms and whether code tuning is appropriate. If
636 code tuning isn't appropriate, go back to step 1.
 - 637 d. Tune the bottleneck identified in step (c).
 - 638 e. Measure each improvement one at a time.
 - 639 f. If an improvement doesn't improve the code, revert to the code saved in
640 step (a). (Typically, more than half the attempted tunings will produce
641 only a negligible improvement in performance or degrade performance.)
- 642 3. Repeat from step 2.

CC2E.COM/2585

643 **Additional Resources**

644 **Performance**

645 Smith, Connie U. and Lloyd G. Williams. *Performance Solutions: A Practical
646 Guide to Creating Responsive, Scalable Software*, Boston, Mass.: Addison
647 Wesley, 2002. This book covers software performance engineering, an approach
648 for building performance into software systems at all stages of development. It
649 makes extensive use of examples and case studies for several kinds of programs.
650 It includes specific recommendations for web applications and pays special at-
651 tention to scalability.

CC2E.COM/2592

652 Newcomer, Joseph M. "Optimization: Your Worst Enemy," May 2000,
653 www.flounder.com/optimization.htm. Newcomer is an experienced systems pro-
654 grammer who describes the various pitfalls of ineffective optimization strategies
655 in graphic detail.

656 **Algorithms and Data Types**

657 CC2E.COM/2599 Knuth, Donald. *The Art of Computer Programming*, vol. 1, *Fundamental Algo-*

658 ritisms, 3d ed. Reading, Mass.: Addison-Wesley, 1997.

659 Knuth, Donald. *The Art of Computer Programming*, vol. 2, *Seminumerical Algo-*

660 ritisms, 3d ed. Reading, Mass.: Addison-Wesley, 1997.

661 Knuth, Donald. *The Art of Computer Programming*, vol. 3, *Sorting and Search-*
662 *ing*, 2d ed. Reading, Mass.: Addison-Wesley, 1998.

663 These are the first three volumes of a series that was originally intended to grow
664 to seven volumes. They can be somewhat intimidating. In addition to the English
665 description of the algorithms, they're described in mathematical notation or
666 MIX, an assembly language for the imaginary MIX computer. The books contain
667 exhaustive details on a huge number of topics, and if you have an intense interest
668 in a particular algorithm, you won't find a better reference.

669 Sedgewick, Robert. *Algorithms in Java, Parts 1-4, 3d ed.* Boston, Mass.: Addi-
670 son-Wesley, 2002. This book's four parts contain a survey of the best methods of
671 solving a wide variety of problems. Its subject areas include fundamentals, sort-
672 ing, searching, abstract data type implementation, and advanced topics. Sedge-
673 wick's *Algorithms in Java, Part 5, 3d ed.* (2003) covers graph algorithms.
674 Sedgewick's *Algorithms in C++, Parts 1-4, 3d ed.* (1998), *Algorithms in C++,*
675 *Part 5, 3d ed.* (2002), *Algorithms in C, Parts 1-4, 3d ed.* (1997), and *Algorithms*
676 *in C, Part 5, 3d ed.* (2001) are similarly organized. Sedgewick was a Ph.D. stu-
677 dent of Knuth's.

678 CC2E.COM/2506

678 **CHECKLIST: Code-Tuning Strategy**

679 **Overall Program Performance**

- 680 Have you considered improving performance by changing the program re-
681 quirements?
- 682 Have you considered improving performance by modifying the program's
683 design?
- 684 Have you considered improving performance by modifying the class design?
- 685 Have you considered improving performance by avoiding operating system
686 interactions?

- 687 Have you considered improving performance by avoiding I/O?
688 Have you considered improving performance by using a compiled language
689 instead of an interpreted language?
690 Have you considered improving performance by using compiler optimiza-
691 tions?
692 Have you considered improving performance by switching to different
693 hardware?
694 Have you considered code tuning only as a last resort?

695 **Code-Tuning Approach**

- 696 Is your program fully correct before you begin code tuning?
697 Have you measured performance bottlenecks before beginning code tuning?
698 Have you measured the effect of each code-tuning change
699 Have you backed out the code-tuning changes that didn't produce the in-
700 tended improvement?
701 Have you tried more than one change to improve performance of each bot-
702 tleneck, i.e., *iterated*?

703

704

Key Points

- 705
 - Performance is only one aspect of overall software quality, and it's usually
706 not the most important. Finely tuned code is only one aspect of overall per-
707 formance, and it's usually not the most significant. Program architecture, de-
708 tailed design, and data-structure and algorithm selection usually have more
709 influence on a program's execution speed and size than the efficiency of its
710 code does.
 - Quantitative measurement is a key to maximizing performance. It's needed
711 to find the areas in which performance improvements will really count, and
712 it's needed again to verify that optimizations improve rather than degrade
713 the software.
 - Most programs spend most of their time in a small fraction of their code.
714 You won't know which code that is until you measure it.
 - Multiple iterations are usually needed to achieve desired performance im-
715 provements through code tuning.
 - The best way to prepare for performance work during initial coding is to
716 write clean code that's easy to understand and modify.
- 717
- 718
- 719
- 720

26

Code-Tuning Techniques

3

CC2E.COM/2665

4

26.1 Logic

5

26.2 Loops

6

26.3 Data Transformations

7

26.4 Expressions

8

26.5 Routines

9

26.6 Recoding in Assembler

10

26.7 The More Things Change, the More They Stay the Same

Related Topics

Code-tuning strategies: Chapter 28

11

12

Refactoring: Chapter 24

13

14

CODE TUNING HAS BEEN a popular topic during most of the history of computer programming. Consequently, once you've decided that you need to improve performance and that you want to do it at the code level, you have a rich set of techniques at your disposal.

15

16

17

This chapter focuses on improving speed and includes a few tips for making code smaller. Performance usually refers to both speed and size, but size reductions tend to come more from redesigning classes and data than from tuning code. Code tuning refers to small-scale changes rather than changes in larger-scale designs.

18

19

20

21

22

23

24

25

26

27

28

29

30

Few of the techniques in this chapter are so generally applicable that you'll be able to copy the example code directly into your programs. The main purpose of the discussion here is to illustrate a handful of code tunings that you can adapt to your situation.

The code-tuning changes described in this chapter might seem cosmetically similar to the refactorings described in Chapter 24. But refactorings are changes that improve a program's internal structure (Fowler 1999). The changes in this chapter might better be called "anti-refactorings." Far from "improving the

31
32
33
34

35 **CROSS-REFERENCE** Cod
36 e tunings are heuristics. For
37 more on heuristics, see
38 Section 5.3, "Design
39 Building Blocks: Heuristics."
40
41

internal structure," these changes degrade the internal structure in exchange for gains in performance. This is true by definition. If they didn't degrade the internal structure, we wouldn't consider them to be optimizations; we would use them by default and consider them to be standard coding practice.

Some books present code tuning techniques as "rules of thumb" or cite research that suggests that a specific tuning will produce the desired effect. As you will soon see, the concept of "rules of thumb" applies poorly to code tuning. The only reliable rule of thumb is to measure the effect of each tuning in your environment. Thus this chapter presents a catalog of "things to try"—many of which won't work in your environment but some of which will work very well indeed.

42

26.1 Logic

43 **CROSS-REFERENCE** For
44 other details on using
45 statement logic, see Chapters
46 14 through 19.

Much of programming consists of manipulating logic. This section describes how to manipulate logical expressions to your advantage.

Stop Testing When You Know the Answer

Suppose you have a statement like

47 **if** (5 < x) **and** (x < 10) **then** ...
48
49

Once you've determined that *x* is less than 5, you don't need to perform the second half of the test.

50 **CROSS-REFERENCE** For
51 more on short-circuit
52 evaluation, see "Knowing
53 How Boolean Expressions
54 Are Evaluated" in "Knowing
55 How Boolean Expressions
56 Are Evaluated" in Section
57 19.1.

Some languages provide a form of expression evaluation known as "short-circuit evaluation," which means that the compiler generates code that automatically stops testing as soon as it knows the answer. Short-circuit evaluation is part of C++'s standard operators and Java's "conditional" operators.

If your language doesn't support short-circuit evaluation natively, you have to avoid using *and* and *or*, adding logic instead. With short-circuit evaluation, the code above changes to this:

57 **if** (5 < x) **then**
58 **if** (x < 10) **then** ...
59
60
61
62
63
64

The principle of not testing after you know the answer is a good one for many other kinds of cases as well. A search loop is a common case. If you're scanning an array of input numbers for a negative value and you simply need to know whether a negative value is present, one approach is to check every value, setting a *negativeFound* variable when you find one. Here's how the search loop would look:

C++ Example of Not Stopping After You Know the Answer

```

65
66     negativeInputFound = False;
67     for ( i = 0; i < iCount; i++ ) {
68         if ( input[ i ] < 0 ) {
69             negativeInputFound = True;
70         }
71     }

```

A better approach would be to stop scanning as soon as you find a negative value. Here are the approaches you could use to solve the problem:

- 74 • Add a *break* statement after the *negativeInputFound = True* line.
- 75 • If your language doesn't have *break*, emulate a *break* with a *goto* that goes
76 to the first statement after the loop.
- 77 • Change the *for* loop to a *while* loop and check for *negativeInputFound* as
78 well as for incrementing the loop counter past *iCount*.
- 79 • Change the *for* loop to a *while* loop, put a sentinel value in the first array
80 element after the last value entry, and simply check for a negative value in
81 the *while* test. After the loop terminates, see whether the position of the first
82 found value is in the array or one past the end. Sentinels are discussed in
83 more detail later in the chapter.

84 Here are the results of using the *break* keyword in C++ and Java:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	4.27	3.68	14%
Java	4.85	3.46	29%

85 Note: (1) Times in these tables are given in seconds and are meaningful only for
86 comparisons across rows of each table. Actual times will vary according to the
87 compiler and compiler options used and the environment in which each test is run.
88 (2) Benchmark results are typically made up of several thousand to many million
89 executions of the code fragments to smooth out the sample-to-sample fluctuations in
90 the results. (3) Specific brands and versions of compilers aren't indicated.
91 Performance characteristics vary significantly from brand to brand and version to
92 version. (4) Comparisons among results from different languages aren't always
93 meaningful because compilers for different languages don't always offer comparable
94 code-generation options. (5) The results shown for interpreted languages (PHP and
95 Python) are typically based on less than 1% of the test runs used for the other
96 languages. (6) Some of the "time savings" percentages might not be exactly
97 reproducible from the data in these tables due to rounding of the "straight time" and
98 "code-tuned time" entries.

99
100
101
102
The impact of this change varies a great deal depending on how many values you
have and how often you expect to find a negative value. This test assumed an
average of 100 values and assumed that a negative value would be found 50
percent of the time.

103 Order Tests by Frequency

104
105
106
107
Arrange tests so that the one that's fastest and most likely to be true is performed
first. It should be easy to drop through the normal case, and if there are
inefficiencies, they should be in processing the uncommon cases. This principle
applies to *case* statements and to chains of *if-then-elses*.

108 Here's a *Select-Case* statement that responds to keyboard input in a word
109 processor:

110 Visual Basic Example of a Poorly Ordered Logical Test

```
111 Select inputCharacter
112     Case "+", "="
113         ProcessMathSymbol( inputCharacter )
114     Case "0" To "9"
115         ProcessDigit( inputCharacter )
116     Case ",", ".", ":" , ";" , "!" , "?"
117         ProcessPunctuation( inputCharacter )
118     Case " "
119         ProcessSpace( inputCharacter )
120     Case "A" To "Z", "a" To "z"
121         ProcessAlpha( inputCharacter )
122     Case Else
123         ProcessError( inputCharacter )
124 End Select
```

125 The cases in this *case* statement are ordered in something close to the ASCII sort
126 order. In a *case* statement, however, the effect is often the same as if you had
127 written a big set of *if-then-elses*, so if you get an < ;\$QS>a< ;\$QS> as an input
128 character, the program tests whether it's a math symbol, a punctuation mark, a
129 digit, or a space before determining that it's an alphabetic character. If you know
130 the likely frequency of your input characters, you can put the most common
131 cases first. Here's the reordered *case* statement:

132 Visual Basic Example of a Well-Ordered Logical Test

```
133 Select inputCharacter
134     Case "A" To "Z", "a" To "z"
135         ProcessAlpha( inputCharacter )
136     Case " "
137         ProcessSpace( inputCharacter )
```

```

138     Case ",", ".", ":" , ";" , "!", "?"
139         ProcessPunctuation( inputCharacter )
140     Case "0" To "9"
141         ProcessDigit( inputCharacter )
142     Case "+", "="
143         ProcessMathSymbol( inputCharacter )
144     Case Else
145         ProcessError( inputCharacter )
146 End Select

```

Since the most common case is usually found sooner in the optimized code, the net effect will be the performance of fewer tests. Here are the results of this optimization with a typical mix of characters:

Language	Straight Time	Code-Tuned Time	Time Savings
C#	0.220	0.260	-18%
Java	2.56	2.56	0%
Visual Basic	0.280	0.260	7%

150
151 *Note: Benchmarked with an input mix of 78 percent alphabetic characters, 17 percent spaces, and 5 percent punctuation symbols.*

152 The Visual Basic results are as expected, but the Java and C# results are not as
153 expected. Apparently that's because of the way *switch-case* statements are
154 structured in C++ and Java—since each value must be enumerated individually
155 rather than in ranges, the C++ and Java code doesn't benefit from the
156 optimization as the Visual Basic code does. This result underscores the
157 importance of not following any optimization advice blindly—specific compiler
158 implementations will significantly affect the results.

159 You might assume that the code generated by the Visual Basic compiler for a set
160 of *if-then-elses* that perform the same test as the *case* statement would be similar.
161 Here are those results:

Language	Straight Time	Code-Tuned Time	Time Savings
C#	0.630	0.330	48%
Java	0.922	0.460	50%
Visual Basic	1.36	1.00	26%

162 The results are quite different. For the same number of tests, the VB compiler
163 takes about 5 times as long in the unoptimized case, 4 times in the optimized
164 case. This suggests that the compiler is generating different code for the *case*
165 approach than for the *if-then-else* approach.

166 The improvement with *if-then-elses* is more consistent than it was with the *case*
167 statements, but that's a mixed blessing. In C# and VB both versions of the *case*
168 statement approach are faster than both versions of the *if-then-else* approach,
169 whereas in Java both versions are slower.

170 This variation in results suggests a third possible optimization, described in the
171 next section.

172 Compare Performance of Similar Logic Structures

173 The test described above could be performed using either a *case* statement or *if-
174 then-elses*. Depending on the environment, either approach might work better.
175 Here is the data from the preceding two tables reformatted to present the "code-
176 tuned" times comparing *if-then-else* and *case* performance:

Language	case	if-then- else	Time Savings	Performance Ratio
C#	0.260	0.330	-27%	1:1
Java	2.56	0.460	82%	6:1
Visual Basic	0.260	1.00	258%	1:4

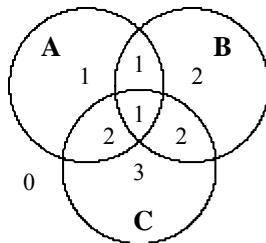
177 These results defy any logical explanation. In one of the languages, *case* is
178 dramatically superior to *if-then-else*, and in another, *if-then-else* is dramatically
179 superior to *case*. In the third language, the difference is relatively small. You
180 might think that because C# and Java share similar syntax for *case* statements,
181 their results would be similar, but in fact their results are opposite each other.

182 This example clearly illustrates the difficulty of performing any sort of "rule of
183 thumb" or "logic" to code tuning—there is simply no reliable substitute for
184 measuring results.

185 Substitute Table Lookups for Complicated 186 Expressions

187 **CROSS-REFERENCE** For
188 details on using table lookups
189 to replace complicated logic,
190 see Chapter 18, "Table-
191 Driven Methods."

In some circumstances, a table lookup may be quicker than traversing a complicated chain of logic. The point of a complicated chain is usually to categorize something and then to take an action based on its category. As an abstract example, suppose you want to assign a category number to something based on which of Groups A, B, and C it falls into:



192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

This table definition is somewhat difficult to understand. Any commenting you can do to make table definitions readable helps.

213

214

215

216

217

218

219

220

221

222

223

224

G26xx01

Here's an example of the complicated logic chain that assigns the category numbers:

C++ Example of a Complicated Chain of Logic

```

197 if ( ( a && !c ) || ( a && b && c ) ) {
198     category = 1;
199 }
200 else if ( ( b && !a ) || ( a && c && !b ) ) {
201     category = 2;
202 }
203 else if ( c && !a && !b ) {
204     category = 3;
205 }
206 else {
207     category = 0;
208 }
```

You can replace this test with a more modifiable and higher-performance lookup table. Here's how:

C++ Example of Using a Table Lookup to Replace Complicated Logic

```

// define categoryTable
static int categoryTable[ 2 ][ 2 ][ 2 ] = {
    // !b!c  !bc  b!c  bc
    0,    3,    2,    2,    //  !a
    1,    2,    1,    1    //   a
};

...
category = categoryTable[ a ][ b ][ c ];
```

Although the definition of the table is hard to read, if it's well documented it won't be any harder to read than the code for the complicated chain of logic was. If the definition changes, the table will be much easier to maintain than the earlier logic would have been. Here are the performance results:

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
C++	5.04	3.39	33%	1.5:1
Visual Basic	5.21	2.60	50%	2:1

225 Use Lazy Evaluation

226 One of my former roommates was a great procrastinator. He justified his laziness
 227 by saying that many of the things people feel rushed to do simply don't need to
 228 be done. If he waited long enough, he claimed, the things that weren't important
 229 would be procrastinated into oblivion, and he wouldn't waste his time doing
 230 them.

231 Lazy evaluation is based on the principle my roommate used. If a program uses
 232 lazy evaluation, it avoids doing any work until the work is needed. Lazy
 233 evaluation is similar to just-in-time strategies that do the work closest to when
 234 it's needed.

235 Suppose, for example, that your program contains a table of 5000 values,
 236 generates the whole table at startup time, and then uses it as the program
 237 executes. If the program uses only a small percentage of the entries in the table,
 238 it might make more sense to compute them as they're needed rather than all at
 239 once. Once an entry is computed, it can still be stored for future reference
 240 ("cached").

241 26.2 Loops

242 **CROSS-REFERENCE** For
 243 other details on loops, see
 Chapter 16, "Controlling
 Loops."

244 Unswitching

245 Switching refers to making a decision inside a loop every time it's executed. If
 246 the decision doesn't change while the loop is executing, you can unswitch the
 247 loop by making the decision outside the loop. Usually this requires turning the
 248 loop inside out, putting loops inside the conditional rather than putting the
 249 conditional inside the loop. Here's an example of a loop before unswitching:

CODING HORROR

250 **CROSS-REFERENCE** As
 251 in the last chapter, this code
 252 fragment violates several
 253 rules of good programming.
 254 Readability and maintenance
 255 are usually more important
 256 than execution speed or size,
 257 but in this chapter the topic is
 258 performance, and that implies
 259 a trade-off with the other
 260 objectives. Like the last
 261 chapter, you'll see many
 examples of coding practices
 here that aren't recommended
 in other parts of this book.

262
 263
 264
 265
 266
 267
 268
 269
 270
 271
 272
 273

274
 275
 276
 277

278 This example also illustrates a key challenge in code tuning—the effect of any
 279 specific code tuning is not predictable. The code tuning produced significant
 280 improvements in three of the four languages, but not in Visual Basic. To perform
 281 this specific optimization in this specific version of VB would produce less
 282 maintainable code without any offsetting gain in performance. The general

C++ Example of a Switched Loop

```
for ( i = 0; i < count; i++ ) {
    if ( sumType == SUMTYPE_NET ) {
        netSum = netSum + amount[ i ];
    }
    else {
        grossSum = grossSum + amount[ i ];
    }
}
```

In this code, the test `if(sumType == SUMTYPE_NET)` is repeated through each iteration even though it'll be the same each time through the loop. You can rewrite the code for a speed gain this way:

C++ Example of an Unswitched Loop

```
if ( sumType == SUMTYPE_NET ) {
    for ( i = 0; i < count; i++ ) {
        netSum = netSum + amount[ i ];
    }
}
else {
    for ( i = 0; i < count; i++ ) {
        grossSum = grossSum + amount[ i ];
    }
}
```

This is good for about a 20 percent time savings:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	2.81	2.27	19%
Java	3.97	3.12	21%
Visual Basic	2.78	2.77	<1%
Python	8.14	5.87	28%

A hazard distinct to this case is that the two loops have to be maintained in parallel. If `count` changes to `clientCount`, you have to remember to change it in both places, which is an annoyance for you and a maintenance headache for anyone else who has to work with the code.

283
284

lesson is that you must measure the effect of each specific optimization to be sure of its effect—no exceptions.

285

Jamming

286
287
288

Jamming, or “fusion,” is the result of combining two loops that operate on the same set of elements. The gain lies in cutting the loop overhead from two loops to one. Here’s a candidate for loop jamming:

289

290
291
292
293
294
295
296
297
298
299

Visual Basic Example of Separate Loops That Could Be Jammed

```
For i = 0 to employeeCount - 1
    employeeName( i ) = ""
Next
...
For i = 0 to employeeCount - 1
    employeeEarnings( i ) = 0
Next
```

When you jam loops, you find code in two loops that you can combine into one. Usually, that means the loop counters have to be the same. In this example, both loops run from 0 to *employeeCount - 1*, so you can jam them. Here’s how:

300 **CODING HORROR**301
302
303
304
305

Visual Basic Example of a Jammed Loop

```
For i = 0 to employeeCount - 1
    employeeName( i ) = ""
    employeeEarnings( i ) = 0
Next
```

Here are the savings:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	3.68	2.65	28%
PHP	3.97	2.42	32%
Visual Basic	3.75	3.56	4%

306

Note: Benchmarked for the case in which employeeCount equals 100.

307

As before, the results vary significantly among languages.

308
309
310
311
312

Loop jamming has two main hazards. First, the indexes for the two parts that have been jammed might change so that they’re no longer compatible. Second, you might not be able to combine the loops easily. Before you combine the loops, make sure they’ll still be in the right order with respect to the rest of the code.

313

314

315

316

317

318

319

320

321

322

323 *Normally, you'd probably use
 324 a for loop for a job like this,
 325 but to optimize, you'd have to
 326 convert to a while loop. For
 327 clarity, a while loop is shown
 328 here.*

329

330

CODING HORROR

332

333

334

335

336

337

338

339 *These lines pick up the case
 340 that might fall through the
 341 cracks if the loop went by
 342 twos instead of by ones.*

343

344

345

346

347

348

349

350

Unrolling

The goal of loop unrolling is to reduce the amount of loop housekeeping. In Chapter 25, a loop was completely unrolled, and 10 lines of code were shown to be faster than 3. In that case, the loop that went from 3 to 10 lines was unrolled so that all 10 array accesses were done individually.

Although completely unrolling a loop is a fast solution and works well when you're dealing with a small number of elements, it's not practical when you have a large number of elements or when you don't know in advance how many elements you'll have. Here's an example of a general loop:

Java Example of a Loop That Can Be Unrolled

```
i = 0;
while ( i < count ) {
    a[ i ] = i;
    i = i + 1;
}
```

To unroll the loop partially, you handle two or more cases in each pass through the loop instead of one. This unrolling hurts readability but doesn't hurt the generality of the loop. Here's the loop unrolled once:

Java Example of a Loop That's Been Unrolled Once

```
i = 0;
while ( i < count - 1 ) {
    a[ i ] = i;
    a[ i + 1 ] = i + 1;
    i = i + 2;
}

if ( i == count ) {
    a[ count - 1 ] = count - 1;
}
```

The technique replaced the original $a[i] = i$ line with two lines, and i is incremented by 2 rather than by 1. The extra code after the *while* loop is needed when *count* is odd and the loop has one iteration left after the loop terminates.

When five lines of straightforward code expand to nine lines of tricky code, the code becomes harder to read and maintain. Except for the gain in speed, its quality is poor. Part of any design discipline, however, is making necessary trade-offs. So, even though a particular technique generally represents poor coding practice, specific circumstances may make it the best one to use.

Here are the results of unrolling the loop:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	1.75	1.15	34%
Java	1.01	0.581	43%
PHP	5.33	4.49	16%
Python	2.51	3.21	-27%

351 *Note: Benchmarked for the case in which count equals 100.*

352 A gain of 16 to 43 percent is respectable, although again you have to watch out
 353 for hurting performance, as the Python benchmark shows. The main hazard of
 354 loop unrolling is an off-by-one error in the code after the loop that picks up the
 355 last case.

356 What if you unroll the loop even further, going for two or more unrollings? Do
 357 you get more benefit? Here's the code for a loop unrolled twice:

358 CODING HORROR

Java Example of a Loop That's Been Unrolled Twice

```

359 i = 0;
360 while ( i < count - 2 ) {
361     a[ i ] = i;
362     a[ i + 1 ] = i+1;
363     a[ i + 2 ] = i+2;
364     i = i + 3;
365 }
366 if ( i <= count - 1 ) {
367     a[ count - 1 ] = count - 1;
368 }
369 if ( i == count - 2 ) {
370     a[ count -2 ] = count - 2;
371 }
372 
```

Here are the results of unrolling the loop the second time:

Language	Straight Time	Single Unrolled Time	Double Unrolled Time	Time Savings
C++	1.75	1.15	1.01	42%
Java	1.01	0.581	0.581	43%
PHP	5.33	4.49	3.70	31%
Python	2.51	3.21	2.79	-12%

373 *Note: Benchmarked for the case in which count equals 100.*

374 The results indicate that further loop unrolling can result in further time savings,
 375 but not necessarily so, as the Java measurement shows. The main concern is how

376 Byzantine your code becomes. When you look at the code above, you might not
 377 think it looks incredibly complicated, but when you realize that it started life a
 378 couple of pages ago as a five-line loop, you can appreciate the trade-off between
 379 performance and readability.

380 Minimizing the Work Inside Loops

381 One key to writing effective loops is to minimize the work done inside a loop. If
 382 you can evaluate a statement or part of a statement outside a loop so that only the
 383 result is used inside the loop, do so. It's good programming practice, and, in
 384 some cases, it improves readability.

385 Suppose you have a complicated pointer expression inside a hot loop that looks
 386 like this:

387 C++ Example of a Complicated Pointer Expression Inside a Loop

```
388           for ( i = 0; i < rateCount; i++ ) {  
389                netRate[ i ] = baseRate[ i ] * rates->discounts->factors->net;  
390           }
```

391 In this case, assigning the complicated pointer expression to a well-named
 392 variable improves readability and often improves performance.

393 C++ Example of Simplifying a Complicated Pointer Expression

```
394           quantityDiscount = rates->discounts->factors->net;  
395           for ( i = 0; i < rateCount; i++ ) {  
396                netRate[ i ] = baseRate[ i ] * quantityDiscount;  
397           }
```

398 The extra variable, *quantityDiscount*, makes it clear that the *baseRate* array is
 399 being multiplied by a quantity-discount factor to compute the net rate. That
 400 wasn't at all clear from the original expression in the loop. Putting the
 401 complicated pointer expression into a variable outside the loop also saves the
 402 pointer from being dereferenced three times for each pass through the loop,
 403 resulting in the following savings:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	3.69	2.97	19%
C#	2.27	1.97	13%
Java	4.13	2.35	43%

404

Note: Benchmarked for the case in which rateCount equals 100.

405 Except for the Java compiler, the savings aren't anything to crow about,
406 implying that during initial coding you can use whichever technique is more
407 readable without worrying about the speed of the code until later.

408 Sentinel Values

409 When you have a loop with a compound test, you can often save time by
410 simplifying the test. If the loop is a search loop, one way to simplify the test is to
411 use a sentinel value, a value that you put just past the end of the search range and
412 that's guaranteed to terminate the search.

413 The classic example of a compound test that can be improved by use of a
414 sentinel is the search loop that checks both whether it has found the value it is
415 seeking and whether it has run out of values. Here's the code:

416 C# Example of Compound Tests in a Search Loop

```
417     found = FALSE;  
418     i = 0;  
419     Here's the compound test.     while ( ( !found ) && ( i < count ) ) {  
420         if ( item[ i ] == testValue ) {  
421             found = TRUE;  
422         }  
423         else {  
424             i++;  
425         }  
426     }  
427  
428     if ( found ) {  
429         ...  
430     }
```

In this code, each iteration of the loop tests for *!found* and for *i < count*. The purpose of the *!found* test is to determine when the desired element has been found. The purpose of the *i < count* test is to avoid running past the end of the array. Inside the loop, each value of *item[]* is tested individually, so the loop really has three tests for each iteration.

435 In this kind of search loop, you can combine the three tests so that you test only
436 once per iteration by putting a "sentinel" at the end of the search range to stop
437 the loop. In this case, you can simply assign the value you're looking for to the
438 element just beyond the end of the search range. (Remember to leave space for
439 that element when you declare the array.) You then check each element, and if
440 you don't find the element until you find the one you stuck at the end, you know
441 that the value you're looking for isn't really there. Here's the code:

442
 443
 444
 445 Remember to allow space for
 446 the sentinel value at the end
 447 of the array.
 448
 449
 450
 451
 452
 453
 454
 455
 456
 457
 458

C# Example of Using a Sentinel Value to Speed Up a Loop

```
// set sentinel value, preserving the original value
initialValue = item[ count ];
item[ count ] = testValue;

i = 0;
while ( item[ i ] != testValue ) {
    i++;
}

// restore the value displaced by the sentinel
item[ count ] = initialValue;

// check if value was found
if ( i < count ) {
    ...
}
```

When *item* is an array of integers, the savings can be dramatic:

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
C#	0.771	0.590	23%	1.3:1
Java	1.63	0.912	44%	2:1
Visual Basic	1.34	0.470	65%	3:1

459 Note: Search is of a 100-element array of integers.

460 The Visual Basic results are particularly dramatic, but all the results are good.
 461 When the kind of array changes, however, the results also change. Here are the
 462 results when *item* is an array of single-precision floating-point numbers:

Language	Straight Time	Code-Tuned Time	Time Savings
C#	1.351	1.021	24%
Java	1.923	1.282	33%
Visual Basic	1.752	1.011	42%

463 Note: Search is of a 100-element array of 4-byte floating-point numbers.

464 As usual, the results vary significantly.

465 The sentinel technique can be applied to virtually any situation in which you use
 466 a linear search—to linked lists as well as arrays. The only caveats are that you
 467 must choose the sentinel value carefully and that you must be careful about how
 468 you put the sentinel value into the array or linked list.

469

470

471

472

473

474

475

476

477

478

479

480

481

482

483

484

485

486

487

Putting the Busiest Loop on the Inside

When you have nested loops, think about which loop you want on the outside and which you want on the inside. Following is an example of a nested loop that can be improved.

Java Example of a Nested Loop That Can Be Improved

```
for ( column = 0; column < 100; column++ ) {
    for ( row = 0; row < 5; row++ ) {
        sum = sum + table[ row ][ column ];
    }
}
```

The key to improving the loop is that the outer loop executes much more often than the inner loop. Each time the loop executes, it has to initialize the loop index, increment it on each pass through the loop, and check it after each pass. The total number of loop executions is 100 for the outer loop and $100 * 5 = 500$ for the inner loop, for a total of 600 iterations. By merely switching the inner and outer loops, you can change the total number of iterations to 5 for the outer loop and $5 * 100 = 500$ for the inner loop, for a total of 505 iterations. Analytically, you'd expect to save about $(600 - 505) / 600 = 16$ percent by switching the loops. Here's the measured difference in performance:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	4.75	3.19	33%
Java	5.39	3.56	34%
PHP	4.16	3.65	12%
Python	3.48	3.33	4%

The results vary significantly, which shows once again that you have to measure the effect in your particular environment before you can be sure your optimization will help.

488

489

490

491

Strength Reduction

Reducing strength means replacing an expensive operation such as multiplication with a cheaper operation such as addition. Sometimes you'll have an expression inside a loop that depends on multiplying the loop index by a factor. Addition is usually faster than multiplication, and if you can compute the same number by adding the amount on each iteration of the loop rather than by multiplying, the code will run faster. Here's an example of code that uses multiplication:

499

Visual Basic Example of Multiplying a Loop Index

```

500   For i = 0 to saleCount - 1
      commission( i ) = (i + 1) * revenue * baseCommission * discount
502   Next
503
504 This code is straightforward but expensive. You can rewrite the loop so that you
505 accumulate multiples rather than computing them each time. This reduces the
506 strength of the operations from multiplication to addition. Here's the code:
```

506

Visual Basic Example of Adding Rather Than Multiplying

```

507   incrementalCommission = revenue * baseCommission * discount
508   cumulativeCommission = incrementalCommission
509   For i = 0 to saleCount - 1
510     commission( i ) = cumulativeCommission
511     cumulativeCommission = cumulativeCommission + incrementalCommission
512   Next
513
514 Multiplication is expensive, and this kind of change is like a manufacturer's
515 coupon that gives you a discount on the cost of the loop. The original code
516 incremented i each time and multiplied it by revenue * baseCommission * discount—first by 1, then by 2, then by 3, and so on. The optimized code sets
517 incrementalCommission equal to revenue * baseCommission * discount. It then
518 adds incrementalCommission to cumulativeCommission on each pass through the
519 loop. On the first pass, it's been added once; on the second pass, it's been added
520 twice; on the third pass, it's been added three times; and so on. The effect is the
521 same as multiplying incrementalCommission by 1, then by 2, then by 3, and so
522 on, but it's cheaper.
```

523

524 The key is that the original multiplication has to depend on the loop index. In
525 this case, the loop index was the only part of the expression that varied, so the
526 expression could be recoded more economically. Here's how much the rewrite
helped in some test cases:

527

528

529

530

531

Language	Straight Time	Code-Tuned Time	Time Savings
C++	4.33	3.80	12%
Visual Basic	3.54	1.80	49%

Note: Benchmark performed with saleCount equals 20. All computed variables are floating point.

26.3 Data Transformations

Changes in data types can be a powerful aid in reducing program size and improving execution speed. Data-structure design is outside the scope of this

532
533

534

535 **CROSS-REFERENCE** For
536 details on using integers and
537 floating point, see Chapter
12, "Fundamental Data
Types."

538

539

540

541

542

543

544

545

546

547

548

549

550

551

552

553

554

555 **CROSS-REFERENCE** For
556 details on arrays, see Section
557 12.8, "Arrays."

558

559

book, but modest changes in the implementation of a specific data type can also benefit performance. Here are a few ways to tune your data types.

Use Integers Rather Than Floating-Point Numbers

Integer addition and multiplication tend to be faster than floating point. Changing a loop index from a floating point to an integer, for example, can save time. Here's an example:

Visual Basic Example of a Loop That Uses a Time-Consuming Floating-Point Loop Index

```
Dim i As Single
For i = 0 to 99
    x( i ) = 0
Next
```

Contrast this with a similar Visual Basic loop that explicitly uses the integer type:

Visual Basic Example of a Loop That Uses a Timesaving Integer Loop Index

```
Dim i As Integer
For i = 0 to 99
    x( i ) = 0
Next
```

How much difference does it make? Here are the results for this Visual Basic code and for similar code in C++ and PHP:

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
C++	2.80	0.801	71%	3.5:1
PHP	5.01	4.65	7%	1:1
Visual Basic	6.84	0.280	96%	25:1

Use the Fewest Array Dimensions Possible

Conventional wisdom maintains that multiple dimensions on arrays are expensive. If you can structure your data so that it's in a one-dimensional array rather than a two-dimensional or three-dimensional array, you might be able to save some time.

Suppose you have initialization code like this:

Java Example of a Standard, Two-Dimensional Array Initialization

```

560
561   for ( row = 0; row < numRows; row++ ) {
562     for ( column = 0; column < numColumns; column++ ) {
563       matrix[ row ][ column ] = 0;
564     }
565   }

```

When this code is run with 50 rows and 20 columns, it takes twice as long with my current Java compiler as when the array is restructured so that it's one-dimensional. Here's how the revised code would look:

Java Example of a One-Dimensional Representation of an Array

```

566
567   for ( entry = 0; entry < numRows * numColumns; entry++ ) {
568     matrix[ entry ] = 0;
569   }

```

Here's a summary of the results, with the addition of comparable results in several other languages:

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
C++	8.75	7.82	11%	1:1
C#	3.28	2.99	9%	1:1
Java	7.78	4.14	47%	2:1
PHP	6.24	4.10	34%	1.5:1
Python	3.31	2.23	32%	1.5:1
Visual Basic	9.43	3.22	66%	3:1

575 *Note: Times for Python and PHP aren't directly comparable to times for the other
576 languages because they were run <1% as many iterations as the other languages.*

577 The results of this optimization are excellent in Visual Basic and Java, good in
578 PHP and Python, but mediocre in C++ and C#. Of course the C++ compiler's
579 unoptimized time was easily the best of the group, so you can't be too hard on it.

580 This wide range of results also show the hazard of following any code-tuning
581 advice blindly. You can never be sure until you try the advice in your specific
582 circumstances.

Minimize Array References

584 In addition to minimizing accesses to doubly or triply dimensioned arrays, it's
585 often advantageous to minimize array accesses, period. A loop that repeatedly
586 uses one element of an array is a good candidate for the application of this
587 technique. Here's an example of an unnecessary array access:

588

C++ Example of Unnecessarily Referencing an Array Inside a Loop

```

589 for ( discountType = 0; discountType < typeCount; discountType++ ) {
590     for ( discountLevel = 0; discountLevel < levelCount; discountLevel++ ) {
591         rate[ discountLevel ] = rate[ discountLevel ] * discount[ discountType ];
592     }
593 }
```

The reference to `discount[discountType]` doesn't change when `discountLevel` changes in the inner loop. Consequently, you can move it out of the inner loop so that you'll have only one array access per execution of the outer loop rather than one for each execution of the inner loop. The next example shows the revised code.

599

C++ Example of Moving an Array Reference Outside a Loop

```

600 for ( discountType = 0; discountType < typeCount; discountType++ ) {
601     thisDiscount = discount[ discountType ];
602     for ( discountLevel = 0; discountLevel < levelCount; discountLevel++ ) {
603         rate[ discountLevel ] = rate[ discountLevel ] * thisDiscount;
604     }
605 }
```

Here are the results:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	32.1	34.5	-7%
C#	18.3	17.0	7%
Visual Basic	23.2	18.4	20%

607
608 *Note: Benchmark times were computed for the case in which typeCount equals 10
and levelCount equals 100.*

609 As usual, the results vary significantly from compiler to compiler.

610 Use Supplementary Indexes

611 Using a supplementary index means adding related data that makes accessing a
612 data type more efficient. You can add the related data to the main data type, or
613 you can store it in a parallel structure.

614 String-Length Index

615 One example of using a supplementary index can be found in the different
616 string-storage strategies. In C, strings are terminated by a byte that's set to 0. In
617 Visual Basic string format, a length byte hidden at the beginning of each string
618 indicates how long the string is. To determine the length of a string in C, a
619 program has to start at the beginning of the string and count each byte until it

620 finds the byte that's set to 0. To determine the length of a Visual Basic string, the
621 program just looks at the length byte. Visual Basic length byte is an example of
622 augmenting a data type with an index to make certain operations—like
623 computing the length of a string—faster.

624 You can apply the idea of indexing for length to any variable-length data type.
625 It's often more efficient to keep track of the length of the structure rather than
626 computing the length each time you need it.

627 **Independent, Parallel Index Structure**

628 Sometimes it's more efficient to manipulate an index to a data type than it is to
629 manipulate the data type itself. If the items in the data type are big or hard to
630 move (on disk, perhaps), sorting and searching index references is faster than
631 working with the data directly. If each data item is large, you can create an
632 auxiliary structure that consists of key values and pointers to the detailed
633 information. If the difference in size between the data-structure item and the
634 auxiliary-structure item is great enough, sometimes you can store the key item in
635 memory even when the data item has to be stored externally. All searching and
636 sorting is done in memory, and you have to access the disk only once, when you
637 know the exact location of the item you want.

638 **Use Caching**

639 Caching means saving a few values in such a way that you can retrieve the most
640 commonly used values more easily than the less commonly used values. If a
641 program randomly reads records from a disk, for example, a routine might use a
642 cache to save the records read most frequently. When the routine receives a
643 request for a record, it checks the cache to see whether it has the record. If it
644 does, the record is returned directly from memory rather than from disk.

645 In addition to caching records on disk, you can apply caching in other areas. In a
646 Microsoft Windows font-proofing program, the performance bottleneck was in
647 retrieving the width of each character as it was displayed. Caching the most
648 recently used character width roughly doubled the display speed.

649 You can cache the results of time-consuming computations too—especially if the
650 parameters to the calculation are simple. Suppose, for example, that you need to
651 compute the length of the hypotenuse of a right triangle, given the lengths of the
652 other two sides. The straightforward implementation of the routine would look
653 like this:

654 **Java Example of a Routine That's Conducive to Caching**

```
655 double Hypotenuse(  
656     double sideA,
```

```

657     double sideB
658   )
659   return Math.sqrt( ( sideA * sideA ) + ( sideB * sideB ) );
660 }

```

If you know that the same values tend to be requested repeatedly, you can cache values this way:

Java Example of Caching to Avoid an Expensive Computation

```

664 private double cachedHypotenuse = 0;
665 private double cachedSideA = 0;
666 private double cachedSideB = 0;
667
668 public double Hypotenuse(
669   double sideA,
670   double sideB
671 ) {
672   // check to see if the triangle is already in the cache
673   if ( ( sideA == cachedSideA ) && ( sideB == cachedSideB ) ) {
674     return cachedHypotenuse;
675   }
676
677   // compute new hypotenuse and cache it
678   cachedHypotenuse = Math.sqrt( ( sideA * sideA ) + ( sideB * sideB ) );
679   cachedSideA = sideA;
680   cachedSideB = sideB;
681
682   return cachedHypotenuse;
683 }

```

The second version of the routine is more complicated than the first and takes up more space, so speed has to be at a premium to justify it. Many caching schemes cache more than one element, so they have even more overhead. Here's the speed difference between these two versions:

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
C++	4.06	1.05	74%	4:1
Java	2.54	1.40	45%	2:1
Python	8.16	4.17	49%	2:1
Visual Basic	24.0	12.9	47%	2:1

688 *Note: The results shown assume that the cache is hit twice for each time it's set.*

689 The success of the cache depends on the relative costs of accessing a cached
690 element, creating an uncached element, and saving a new element in the cache.

691 Success also depends on how often the cached information is requested. In some
692 cases, success might also depend on caching done by the hardware. Generally,
693 the more it costs to generate a new element and the more times the same
694 information is requested, the more valuable a cache is. The cheaper it is to access
695 a cached element and save new elements in the cache, the more valuable a cache
696 is. As with other optimization techniques, caching adds complexity and tends to
697 be error prone.

698

699 **CROSS-REFERENCE** For
700 more information on
701 expressions, see Section 19.1,
“Boolean Expressions.”

702

703
704705
706
707
708709
710
711
712
713
714

26.4 Expressions

Much of the work in a program is done inside mathematical or logical expressions. Complicated expressions tend to be expensive, so this section looks at ways to make them cheaper.

Exploit Algebraic Identities

You can use algebraic identities to replace costly operations with cheaper ones. For example, the following expressions are logically equivalent:

```
not a and not B  
not (a or B)
```

If you choose the second expression instead of the first, you can save a *not* operation.

Although the savings from avoiding a single *not* operation are probably inconsequential, the general principle is powerful. Jon Bentley describes a program that tested whether $\text{sqrt}(x) < \text{sqrt}(y)$ (1982). Since $\text{sqrt}(x)$ is less than $\text{sqrt}(y)$ only when x is less than y , you can replace the first test with $x < y$. Given the cost of the *sqrt()* routine, you’d expect the savings to be dramatic, and they are. Here are the results:

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
C++	7.43	0.010	99.9%	750:1
Visual Basic	4.59	0.220	95%	20:1
Python	4.21	0.401	90%	10:1

Use Strength Reduction

As mentioned earlier, strength reduction means replacing an expensive operation with a cheaper one. Here are some possible substitutions:

715

716
717

- 718 • Replace multiplication with addition.
719 • Replace exponentiation with multiplication.
720 • Replace trigonometric routines with their trigonometric identities.
721 • Replace *longlong* integers with *longs* or *ints* (but watch for performance
722 issues associated with using native-length vs. non-native-length integers)
723 • Replace floating-point numbers with fixed-point numbers or integers.
724 • Replace double-precision floating points with single-precision numbers.
725 • Replace integer multiplication-by-two and division-by-two with shift
726 operations.

727 Here is a detailed example. Suppose you have to evaluate a polynomial. If you're
728 rusty on polynomials, they're the things that look like

729 $Ax^2 + Bx + C$

730 The letters *A*, *B*, and *C* are coefficients, and *x* is a variable. General code to
731 evaluate an *n*th-order polynomial looks like this:

732 Visual Basic Example of Evaluating a Polynomial

```
733       value = coefficient( 0 )
734       For power = 1 To order
735           value = value + coefficient( power ) * x^power
736       Next
```

737 If you're thinking about strength reduction, you'll look at the exponentiation
738 operator with a jaundiced eye. One solution would be to replace the
739 exponentiation with a multiplication on each pass through the loop, which is
740 analogous to the strength-reduction case a few sections ago in which a
741 multiplication was replaced with an addition. Here's how the reduced-strength
742 polynomial evaluation would look:

743 Visual Basic Example of a Reduced-Strength Method of Evaluating a 744 Polynomial

```
745       value = coefficient( 0 )
746       powerOfX = x
747       For power = 1 to order
748           value = value + coefficient( power ) * powerOfX
749           powerOfX = powerOfX * x
750       Next
```

751 This produces a noticeable advantage if you're working with second-order
752 polynomials (polynomials in which the highest-power term is squared) or higher-
753 order polynomials.

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
Python	3.24	2.60	20%	1:1
Visual Basic	6.26	0.160	97%	40:1

754 If you're serious about strength reduction, you still won't care for those two
 755 floating-point multiplications. The strength-reduction principle suggests that you
 756 can further reduce the strength of the operations in the loop by accumulating
 757 powers rather than multiplying them each time. Here's that code:

758 Visual Basic Example of Further Reducing the Strength Required to 759 Evaluate a Polynomial

```
760 value = 0
761 For power = order to 1 Step -1
762     value = ( value + coefficient( power ) ) * x
763 Next
764 value = value + coefficient( 0 )
```

765 This method eliminates the extra *powerOfX* variable and replaces the two
 766 multiplications in each pass through the loop with one.

Language	Straight Time	First Optimization	Second Optimization	Savings over First Optimization
Python	3.24	2.60	2.53	3%
Visual Basic	6.26	0.16	0.31	-94%

767 This is a good example of theory not holding up very well to practice. The code
 768 with reduced strength seems like it should be faster, but it isn't. One possibility
 769 is that decrementing a loop by *-1* instead of incrementing it by *+1* in Visual
 770 Basic hurts performance, but you'd have to measure that hypothesis to be sure.

771 Initialize at Compile Time

772 If you're using a named constant or a magic number in a routine call and it's the
 773 only argument, that's a clue that you could precompute the number, put it into a
 774 constant, and avoid the routine call. The same principle applies to
 775 multiplications, divisions, additions, and other operations.

776 I once needed to compute the base-two logarithm of an integer, truncated to the
 777 nearest integer. The system didn't have a log-base-two routine, so I wrote my
 778 own. The quick and easy approach was to use the fact that

$$779 \log(x)_{\text{base}} = \log(x) / \log(\text{base})$$

780 Given this identity, I could write a routine like this one:

781 **CROSS-REFERENCE** For
782 details on binding variables
783 to their values, see Section
784 10.6, "Binding Time."

785
786
787

788
789

790
791 LOG2 is a named constant
792 equal to 0.69314718.

793
794
795
796

C++ Example of a Log-Base-Two Routine Based on System Routines

```
unsigned int Log2( unsigned int x ) {
    return (unsigned int) ( log( x ) / log( 2 ) );
}
```

This routine was really slow, and since the value of $\log(2)$ never changed, I replaced $\log(2)$ with its computed value, 0.69314718. Then the code looked like this:

C++ Example of a Log-Base-Two Routine Based on a System Routine and a Constant

```
unsigned int Log2( unsigned int x ) {
    return (unsigned int) ( log( x ) / LOG2 );
}
```

Since $\log()$ tends to be an expensive routine, much more expensive than type conversions or division, you'd expect that cutting the calls to the $\log()$ function by half would cut the time required for the routine by about half. Here are the measured results:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	9.66	5.97	38%
Java	17.0	12.3	28%
PHP	2.45	1.50	39%

In this case, the educated guess about the relative importance of the division and type conversions and the estimate of 50 percent were pretty close. Considering the predictability of the results described in this chapter, the accuracy of my prediction in this case proves only that even a blind squirrel finds a nut occasionally.

Be Wary of System Routines

System routines are expensive and provide accuracy that's often wasted. Typical system math routines, for example, are designed to put an astronaut on the moon within ± 2 feet of the target. If you don't need that degree of accuracy, you don't need to spend the time to compute it either.

In the previous example, the $\text{Log2}()$ routine returned an integer value but used a floating-point $\log()$ routine to compute it. That was overkill for an integer result, so after my first attempt, I wrote a series of integer tests that were perfectly accurate for calculating an integer \log_2 . Here's the code:

C++ Example of a Log-Base-Two Routine Based on Integers

```
unsigned int Log2( unsigned int x ) {
```

```

813     if ( x < 2 ) return 0 ;
814     if ( x < 4 ) return 1 ;
815     if ( x < 8 ) return 2 ;
816     if ( x < 16 ) return 3 ;
817     if ( x < 32 ) return 4 ;
818     if ( x < 64 ) return 5 ;
819     if ( x < 128 ) return 6 ;
820     if ( x < 256 ) return 7 ;
821     if ( x < 512 ) return 8 ;
822     if ( x < 1024 ) return 9 ;
823     ...
824     if ( x < 2147483648 ) return 30;
825     return 31 ;
826 }
```

This routine uses integer operations, never converts to floating point, and blows the doors off both floating-point versions. Here are the results:

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
C++	9.66	0.662	93%	15:1
Java	17.0	0.882	95%	20:1
PHP	2.45	3.45	-41%	2:3

Most of the so-called “transcendental” functions are designed for the worst case—that is, they convert to double-precision floating point internally even if you give them an integer argument. If you find one in a tight section of code and don’t need that much accuracy, give it your immediate attention.

Another option is to take advantage of the fact that a right-shift operation is the same as dividing by two. The number of times you can divide a number by two and still have a nonzero value is the same as the \log_2 of that number. Here’s how code based on that observation looks:

CODING HORROR

C++ Example of an Alternative Log-Base-Two Routine Based on the Right-Shift Operator

```

837
838
839     unsigned int Log2( unsigned int x ) {
840         unsigned int i = 0;
841         while ( ( x = ( x >> 1 ) ) != 0 ) {
842             i++;
843         }
844         return i ;
845     }
```

846 To non-C++ programmers, this code is particularly hard to read. The
 847 complicated expression in the *while* condition is an example of a coding practice
 848 you should avoid unless you have a good reason to use it.

849 This routine takes about 350 percent longer than the longer version above,
 850 executing in 2.4 seconds rather than 0.66 seconds. But it's faster than the first
 851 approach, and adapts easily to 32-bit, 64-bit, and other environments.

852 **KEY POINT**

853 This example highlights the value of not stopping after one successful
 854 optimization. The first optimization earned a respectable 30-40 percent savings
 855 but had nowhere near the impact of the second optimization or third
 optimizations.

856 **Use the Correct Type of Constants**

857 Use named constants and literals that are the same type as the variables they're
 858 assigned to. When a constant and its related variable are different types, the
 859 compiler has to do a type conversion to assign the constant to the variable. A
 860 good compiler does the type conversion at compile time so that it doesn't affect -
 861 run-time performance.

862 A less advanced compiler or an interpreter generates code for a runtime
 863 conversion, so you might be stuck. Here are some differences in performance
 864 between the initializations of a floating-point variable *x* and an integer variable *i*
 865 in two cases. In the first case, the initializations look like this:

866 *x* = 5
 867 *i* = 3.14

868 and require type conversions, assuming *x* is a floating point variable and *i* is an
 869 integer. In the second case, they look like this:

870 *x* = 3.14
 871 *i* = 5

872 and don't require type conversions. Here are the results:

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
C++	1.11	0.000	100%	not measurable
C#	1.49	1.48	<1%	1:1
Java	1.66	1.11	33%	1.5:1
Visual Basic	0.721	0.000	100%	not measurable
PHP	0.872	0.847	3%	1:1

873

The variation among compilers is once again notable.

874

875

876

877

878

879

880

881

882

883

884

885

886

887

888

889

890

891

892

893

894

895

896

897

898

899

900

901

902

903

904

905

906

907

908

909

910

911

Precompute Results

A common low-level design decision is the choice of whether to compute results on the fly or compute them once, save them, and look them up as needed. If the results are used many times, it's often cheaper to compute them once and look them up the rest of the time.

This choice manifests itself in several ways. At the simplest level, you might compute part of an expression outside a loop rather than inside. An example of this appeared earlier in the chapter. At a more complicated level, you might compute a lookup table once when program execution begins, using it every time thereafter, or you might store results in a data file or embed them in a program.

In a space-wars video game, for example, the programmers initially computed gravity coefficients for different distances from the sun. The computation for the gravity coefficients was expensive and affected performance. The program recognized relatively few distinct distances from the sun, however, so the programmers were able to precompute the gravity coefficients and store them in a 10-element array. The array lookup was much faster than the expensive computation.

Suppose you have a routine that computes payment amounts on automobile loans. The code for such a routine would look like this:

Java Example of a Complex Computation That Could Be Precomputed

```
double ComputePayment(
    long loanAmount,
    int months,
    double interestRate
) {
    return loanAmount /
        (
            ( 1.0 - Math.pow( ( 1.0 + ( interestRate / 12.0 ) ), -months ) ) /
            ( interestRate / 12.0 )
        );
}
```

The formula for computing loan payments is complicated and fairly expensive. Putting the information into a table instead of computing it each time would probably be cheaper.

How big would the table be? The widest-ranging variable is *loanAmount*. The variable *interestRate* might range from 5 percent through 20 percent by quarter points, but that's only 61 distinct rates. *months* might range from 12 through 72, but that's only 61 distinct periods. *loanAmount* could conceivably range from

912 \$1000 through \$100,000, which is more entries than you'd generally want to
 913 handle in a lookup table.

914 Most of the computation doesn't depend on *loanAmount*, however, so you can
 915 put the really ugly part of the computation (the denominator of the larger
 916 expression) into a table that's indexed by *interestRate* and *months*. You
 917 recompute the *loanAmount* part each time. Here's the revised code:

Java Example of Precomputing a Complex Computation

```
918
919 double ComputePayment(
920   long loanAmount,
921   int months,
922   double interestRate
923   ) {
924     The new variable
925     interestIndex is created to
926     provide a subscript into the
927     loanDivisor array.
928
929   int interestIndex =
930     Math.round( ( interestRate - LOWEST_RATE ) * GRANULARITY * 100.00 );
931   return loanAmount / loanDivisor[ interestIndex ][ months ];
932 }
```

In this code, the hairy calculation has been replaced with the computation of an array index and a single array access. Here are the results of the change:

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
Java	2.97	0.251	92%	10:1
Python	3.86	4.63	-20%	1:1

930 Depending on your circumstances, you would need to precompute the
 931 *loanDivisor* array at program initialization time or read it from a disk file.
 932 Alternatively, you could initialize it to 0, compute each element the first time it's
 933 requested, store it, and look it up each time it's requested subsequently. That
 934 would be a form of caching, discussed earlier.

935 You don't have to create a table to take advantage of the performance gains you
 936 can achieve by precomputing an expression. Code similar to the code in the
 937 previous examples raises the possibility of a different kind of precomputation.
 938 Suppose you have code that computes payments for many loan amounts, as
 939 shown here.

Java Example of a Second Complex Computation That Could Be Precomputed

```
940
941 double ComputePayments(
942   int months,
943   double interestRate
944   ) {
```

```

946
947
948
949
950
951
952 The following code would do
953 something with payment here;
954 for this example's point, it
955 doesn't matter what.
956
957

```

```

        for ( long loanAmount = MIN_LOAN_AMOUNT; loanAmount < MAX_LOAN_AMOUNT;
              loanAmount++ ) {
            payment = loanAmount / (
                ( 1.0 - Math.pow( 1.0+(interestRate/12.0), - months ) ) /
                ( interestRate/12.0 )
            );
            ...
        }
    }

```

Even without precomputing a table, you can precompute the complicated part of the expression outside the loop and use it inside the loop. Here's how it would look:

```

958
959
960
961
962
963
964 Here's the part that's
965 precomputed.
966
967
968
969
970
971
972
973
974
975

```

Java Example of Precomputing the Second Complex Computation

```

double ComputePayments(
    int months,
    double interestRate
) {
    long loanAmount;
    double divisor = ( 1.0 - Math.pow( 1.0+(interestRate/12.0). - months ) ) /
        ( interestRate/12.0 );
    for ( long loanAmount = MIN_LOAN_AMOUNT; loanAmount <= MAX_LOAN_AMOUNT;
          loanAmount++ ) {
        payment = loanAmount / divisor;
        ...
    }
}

```

This is similar to the techniques suggested earlier of putting array references and pointer dereferences outside a loop. The results for Java in this case are comparable to the results of using the precomputed table in the first optimization:

Language	Straight Time	Code-Tuned Time	Time Savings	Performance Ratio
Java	7.43	0.24	97%	30:1
Python	5.00	1.69	66%	3:1

Python improved here, but not in the first optimization attempt. Many times when one optimization does not produce the desired results, a seemingly similar optimization will work as expected.

Optimizing a program by precomputation can take several forms:

- Computing results before the program executes and wiring them into constants that are assigned at compile time

- 982 • Computing results before the program executes and hard-coding them into
983 variables used at run time
- 984 • Computing results before the program executes and putting them into a file
985 that's loaded at run time
- 986 • Computing results once, at program startup, and then referencing them each
987 time they're needed
- 988 • Computing as much as possible before a loop begins, minimizing the work
989 done inside the loop
- 990 • Computing results the first time they're needed and storing them so that you
991 can retrieve them when they're needed again

992 **Eliminate Common Subexpressions**

993 If you find an expression that's repeated several times, assign it to a variable and
994 refer to the variable rather than recomputing the expression in several places.
995 The loan-calculation example has a common subexpression that you could
996 eliminate. Here's the original code:

997 **Java Example of a Common Subexpression**

```
998 payment = loanAmount / (
999     ( 1.0 - Math.pow( 1.0 + ( interestRate / 12.0 ), -months ) ) /
1000    ( interestRate / 12.0 )
1001 );
```

1002 In this sample, you can assign *interestRate/12.0* to a variable that is then
1003 referenced twice rather than computing the expression twice. If you have chosen
1004 the variable name well, this optimization can improve the code's readability at
1005 the same time that it improves performance. The next example shows the revised
1006 code.

1007 **Java Example of Eliminating a Common Subexpression**

```
1008 monthlyInterest = interestRate / 12.0;
1009 payment = loanAmount / (
1010     ( 1.0 - Math.pow( 1.0 + monthlyInterest, -months ) ) /
1011     monthlyInterest
1012 );
```

1013 The savings in this case don't seem impressive:

Language	Straight Time	Code-Tuned Time	Time Savings
Java	2.94	2.83	4%
Python	3.91	3.94	-1%

1014
1015
1016
1017
1018

It appears that the *Math.pow()* routine is so costly that it overshadows the savings from subexpression elimination. Or possibly the subexpression is already being eliminated by the compiler. If the subexpression were a bigger part of the cost of the whole expression or if the compiler optimizer were less effective, the optimization might have more impact.

1019

1020 **CROSS-REFERENCE** For
1021 details on working with
1022 routines, see Chapter 7,
1023 “High-Quality Routines.”
1024
1025
1026

26.5 Routines

One of the most powerful tools in code tuning is a good routine decomposition. Small, well-defined routines save space because they take the place of doing jobs separately in multiple places. They make a program easy to optimize because you can refactor code in one routine and thus improve every routine that calls it. Small routines are relatively easy to rewrite in assembler. Long, tortuous routines are hard enough to understand on their own; in assembler they’re impossible.

1027

1028
1029
1030
1031
1032
1033

Rewrite Routines In Line

In the early days of computer programming, some machines imposed prohibitive performance penalties for calling a routine. A call to a routine meant that the operating system had to swap out the program, swap in a directory of routines, swap in the particular routine, execute the routine, swap out the routine, and swap the calling routine back in. All this swapping chewed up resources and made the program slow.

1034
1035

Modern computers collect a far smaller toll for calling a routine. Here are the results of putting a string-copy routine in line:

Language	Routine Time	Inline-Code Time	Time Savings
C++	0.471	0.431	8%
Java	13.1	14.4	-10%

1036
1037
1038
1039
1040
1041
1042
1043

In some cases, you might be able to save a few nanoseconds by putting the code from a routine into the program directly where it’s needed using a language feature like C++’s *inline* keyword. If you’re working in a language that doesn’t support *inline* directly but that does have a macro preprocessor, you can use a macro to put the code in, switching it in and out as needed. But modern machines—and “modern” means any machine you’re ever likely to work on—impose virtually no penalty for calling a routine. As the example shows, you’re as likely to degrade performance by keeping code inline as to optimize it.

26.6 Recoding in Assembler

One longstanding piece of conventional wisdom that shouldn't be left unmentioned is the advice that when you run into a performance bottleneck, you should recode in assembler. Recoding in assembler tends to improve both speed and code size. Here is a typical approach to optimizing with assembler:

1. Write 100 percent of an application in a high-level language.
2. Fully test the application, and verify that it's correct.
3. If performance improvements are needed after that, profile the application to identify hot spots. Since about 5 percent of a program usually accounts for about 50 percent of the running time, you can usually identify small pieces of the program as hot spots.
4. Recode a few small pieces in assembler to improve overall performance.

Whether you follow this well-beaten path depends on how comfortable you are with assembler, how well-suited the problem is to assembler, and on your level of desperation.

I got my first exposure to assembler on the DES encryption program I mentioned in the previous chapter. I had tried every optimization I'd ever heard of, and the program was still twice as slow as the speed goal. Recoding part of the program in assembler was the only remaining option. As an assembler novice, about all I could do was make a straight translation from a high-level language to assembler, but I got a 50 percent improvement even at that rudimentary level.

Suppose you have a routine that converts binary data to uppercase ASCII characters. The next example shows the Delphi code to do it.

Delphi Example of Code That's Better Suited to Assembler

```
procedure HexExpand(
    var source: ByteArray;
    var target: WordArray;
    byteCount: word
);
var
    index: integer;
    lowerByte: byte;
    upperByte: byte;
    targetIndex: integer;
begin
```

```

1079     targetIndex := 1;
1080     for index := 1 to byteCount do begin
1081         target[ targetIndex ] := ( source[ index ] and $F0) shr 4 ) + $41;
1082         target[ targetIndex+1 ] := ( source[ index ] and $0F ) + $41;
1083         targetIndex := targetIndex + 2;
1084     end;
1085 end;
1086
1087 Although it's hard to see where the fat is in this code, it contains a lot of bit
1088 manipulation, which isn't exactly Delphi's forte. Bit manipulation is assembler's
1089 forte, however, so this code is a good candidate for recoding. Here's the
1090 assembler code:
```

Example of a Routine Recoded in Assembler

```

1091 procedure HexExpand(
1092     var source;
1093     var target;
1094     byteCount : Integer
1095 );
1096
1097     label
1098         EXPAND;
1099
1100     asm
1101         MOV ECX,byteCount      // load number of bytes to expand
1102         MOV ESI,source        // source offset
1103         MOV EDI,target        // target offset
1104         XOR EAX,EAX          // zero out array offset
1105
1106         EXPAND:
1107             MOV EBX,EAX          // array offset
1108             MOV DL,[ESI+EBX]      // get source byte
1109             MOV DH,DL              // copy source byte
1110
1111             AND DH,$F              // get msbs
1112             ADD DH,$41            // add 65 to make upper case
1113
1114             SHR DL,4              // move 1sbs into position
1115             AND DL,$F              // get 1sbs
1116             ADD DL,$41            // add 65 to make upper case
1117
1118             SHL BX,1              // double offset for target array offset
1119             MOV [EDI+EBX],DX        // put target word
1120
1121             INC EAX                // increment array offset
1122             LOOP EXPAND           // repeat until finished
1123     end;
```

1123
1124
1125
1126
Rewriting in assembler in this case was profitable, resulting in a time savings of
41 percent. It's logical to assume that code in a language that's more suited to bit
manipulation—C++, for instance—would have less to gain than Delphi code
would. Here are the results:

Language	High-Level Time	Assembler Time	Time Savings
C++	4.25	3.02	29%
Delphi	5.18	3.04	41%

1127
1128
1129
1130
The “before” picture in this measurements reflects the two languages' strengths
at bit manipulation. The “after” picture looks virtually identical, and it appears
that the assembler code has minimized the initial performance differences
between Delphi and C++.

1131
1132
1133
1134
The assembler routine shows that rewriting in assembler doesn't have to produce
a huge, ugly routine. Such routines are often quite modest, as this one is.
Sometimes assembler code is almost as compact as its high-level-language
equivalent.

1135
1136
1137
1138
1139
1140
1141
1142
A relatively easy and effective strategy for recoding in assembler is to start with
a compiler that generates assembler listings as a by-product of compilation.
Extract the assembler code for the routine you need to tune, and save it in a
separate source file. Using the compiler's assembler code as a base, hand-
optimize the code, checking for correctness and measuring improvements at each
step. Some compilers intersperse the high-level-language statements as
comments in the assembler code. If yours does, you might keep them in the
assembler code as documentation.

CC2E.COM/2672
1143

CHECKLIST: Code-Tuning Techniques

1144
Improve Both Speed and Size

- Substitute table lookups for complicated logic
- Jam loops
- Use integer instead of floating-point variables
- Initialize data at compile time
- Use constants of the correct type
- Precompute results
- Eliminate common subexpressions
- Translate key routines to assembler

1153

1154

1155

1156

1157

1158

1159

1160

1161

1162

1163

1164

1165

1166

1167

1168

1169

1170

1171

1172

Improve Speed Only

- Stop testing when you know the answer
- Order tests in *case* statements and *if-then-else* chains by frequency
- Compare performance of similar logic structures
- Use lazy evaluation
- Unswitch loops that contain *if* tests
- Unroll loops
- Minimize work performed inside loops
- Use sentinels in search loops
- Put the busiest loop on the inside of nested loops
- Reduce the strength of operations performed inside loops
- Change multiple-dimension arrays to a single dimension
- Minimize array references
- Augment data types with indexes
- Cache frequently used values
- Exploit algebraic identities
- Reduce strength in logical and mathematical expressions
- Be wary of system routines
- Rewrite routines in line

1173

1174

26.7 The More Things Change, the More They Stay the Same

You might expect that performance attributes of systems would have changed somewhat in the 10 years since I wrote the first edition of *Code Complete*, and in some ways they have. Computers are dramatically faster and memory is more plentiful. In the first edition, I ran most of the tests in this chapter 10,000 to 50,000 times to get meaningful, measurable results. For this edition I had to run most tests 1 million to 100 million times. When you have to run a test 100 million times to get measurable results, you have to ask whether anyone will ever notice the impact in a real program. Computers have become so powerful that for many common kinds of programs the level of performance optimization discussed in this chapter has become irrelevant.

In other ways, performance issues have hardly changed at all. People writing desktop applications may not need this information, but people writing software

1187 for embedded systems, real-time systems, and other systems with strict speed or
1188 space restrictions can still benefit from this information.

1189 The need to measure the impact of each and every attempt at code tuning has
1190 been a constant since Donald Knuth published his study of Fortran programs in
1191 1971. According to the measurements in this chapter, the effect of any specific
1192 optimization is actually *less predictable* than it was 10 years ago. The effect of
1193 each code tuning is affected by the programming language, compiler, compiler
1194 version, code libraries, library versions, and compiler settings, among other
1195 things.

1196 Code tuning invariably involves tradeoffs among complexity, readability,
1197 simplicity, and maintainability on the one hand and a desire to improve
1198 performance on the other. It introduces a high degree of maintenance overhead
1199 because of all the reprofiling that's required.

1200 I have found that insisting on *measurable improvement* is a good way to resist
1201 the temptation to optimize prematurely and to enforce a bias toward clear,
1202 straightforward code. If an optimization is important enough to haul out the
1203 profiler and measure the optimization's effect, then it's probably important
1204 enough to allow—as long as it works. But if an optimization isn't important
1205 enough to haul out the profiling machinery, then it isn't important enough to
1206 degrade readability, maintainability, and other code characteristics. The impact
1207 of unmeasured code tuning on performance is speculative at best, whereas the
1208 impact on readability is as certain as it is detrimental.

CC2E.COM/2679

1209

Additional Resources

1210 My favorite reference on code tuning is *Writing Efficient Programs* (Bentley,
1211 Englewood Cliffs, N.J.: Prentice Hall, 1982). The book is out of print, but worth
1212 reading if you can find it. It's an expert treatment of code tuning, broadly
1213 considered. Bentley describes techniques that trade time for space and space for
1214 time. He provides several examples of redesigning data types to reduce both
1215 space and time. His approach is a little more anecdotal than the one taken here,
1216 and his anecdotes are interesting. He takes a few routines through several
1217 optimization steps so that you can see the effects of first, second, and third
1218 attempts on a single problem. Bentley strolls through the primary contents of the
1219 book in 135 pages. The book has an unusually high signal-to-noise ratio—it's
1220 one of the rare gems that every practicing programmer should own.

1221 Appendix 4 of Bentley's *Programming Pearls*, 2d Ed. (2000), contains a
1222 summary of the code tuning rules from his earlier book.

- 1223 You can also find a full array of technology-specific optimization books. Several
1224 are listed below, and the web link to the left contains an up-to-date list.
- 1225 CC2E.COM/2686 Booth, Rick. *Inner Loops : A Sourcebook for Fast 32-bit Software Development*,
1226 Boston, Mass.: Addison Wesley, 1997.
- 1227 Gerber, Richard. *Software Optimization Cookbook: High-Performance Recipes*
1228 for the Intel Architecture, Intel Press, 2002.
- 1229 Hasan, Jeffrey and Kenneth Tu. *Performance Tuning and Optimizing ASP.NET*
1230 Applications, Apress, 2003.
- 1231 Killelea, Patrick. *Web Performance Tuning, 2d Ed*, O'Reilly & Associates, 2002.
- 1232 Larman, Craig and Rhett Guthrie. *Java 2 Performance and Idiom Guide*,
1233 Englewood Cliffs, N.J.: Prentice Hall, 2000.
- 1234 Shirazi, Jack. *Java Performance Tuning*, O'Reilly & Associates, 2000.
- 1235 Wilson, Steve and Jeff Kesselman. *Java Platform Performance: Strategies and*
1236 *Tactics*, Boston, Mass.: Addison Wesley, 2000.

1237 Key Points

- 1238 • Results of optimizations vary widely with different languages, compilers,
1239 and environments. Without measuring each specific optimization, you'll
1240 have no idea whether it will help or hurt your program.
- 1241 • The first optimization is often not the best. Even after you find a good one,
1242 keep looking for one that's better.
- 1243 • Code tuning is a little like nuclear energy. It's a controversial, emotional
1244 topic. Some people think it's so detrimental to reliability and maintainability
1245 that they won't do it at all. Others think that with proper safeguards, it's
1246 beneficial. If you decide to use the techniques in this chapter, apply them
1247 with care.

27

How Program Size Affects Construction

CC2E.COM/2761

Contents

- 27.1 Communication and Size
- 27.2 Range of Project Sizes
- 27.3 Effect of Project Size on Errors
- 27.4 Effect of Project Size on Productivity
- 27.5 Effect of Project Size on Development Activities

Related Topics

Prerequisites to construction: Chapter 3

Determining the kind of software you're working on: Section 3.2

Managing construction: Chapter 28

SCALING UP IN SOFTWARE DEVELOPMENT isn't a simple matter of taking a small project and making each part of it bigger. Suppose you wrote the 25,000-line Gigatron software package in 20 staff-months and found 500 errors in field testing. Suppose Gigatron 1.0 is successful as is Gigatron 2.0, and you start work on the Gigatron Deluxe, a greatly enhanced version of the program that's expected to be 250,000 lines of code.

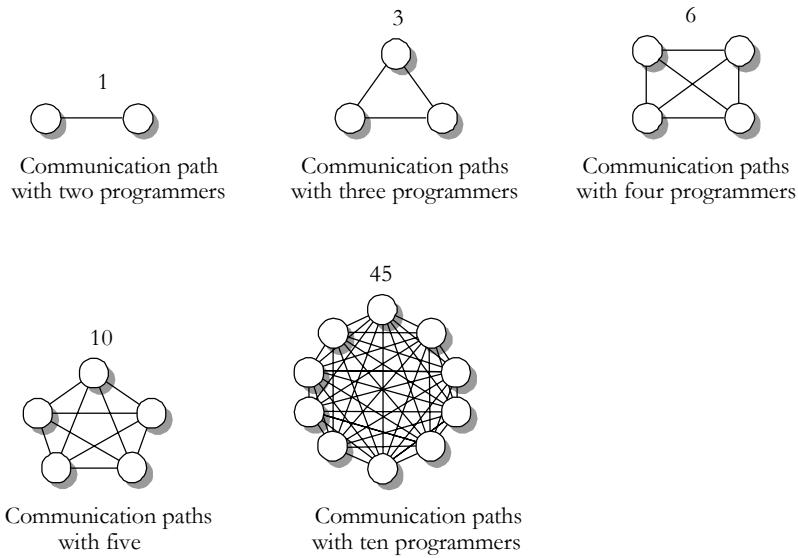
Even though it's 10 times as large as the original Gigatron, the Gigatron Deluxe won't take 10 times the effort to develop; it'll take 30 times the effort. Moreover, 30 times the total effort doesn't imply 30 times as much construction. It probably implies 25 times as much construction and 40 times as much architecture and system testing. You won't have 10 times as many errors either; you'll have 15 times as many—or more.

If you've been accustomed to working on small projects, your first medium-to-large project can rage riotously out of control, becoming an uncontrollable beast instead of the pleasant success you had envisioned. This chapter tells you what kind of beast to expect and where to find the whip and chair to tame it. In

30 contrast, if you're accustomed to working on large projects, you might use
31 approaches that are too formal on a small project. This chapter describes how
32 you can economize to keep the project from toppling under the weight of its own
33 overhead.

34 27.1 Communication and Size

35 If you're the only person on a project, the only communication path is between
36 you and the customer, unless you count the path across your corpus callosum,
37 the path that connects the left side of your brain to the right. As the number of
38 people on a project increases, the number of communication paths increases too.
39 The number doesn't increase additively, as the number of people increases. It
40 increases multiplicatively, proportionally to the square of the number of people.
41 Here's an illustration:



42 F27xx01

43 **Figure 27-1**

44 *45 The number of communication paths increases proportionate to the square of the
46 number of people on the team.*

47 **KEY POINT**
48 As you can see, a two-person project has only one path of communication. A
49 five-person project has 10 paths. A ten-person project has 45 paths, assuming
50 that every person talks to every other person. The 2 percent of projects that have
51 fifty or more programmers have at least 1,200 potential paths. The more
52 communication paths you have, the more time you spend communicating and the
more opportunities are created for communication mistakes. Larger-size projects

53 demand organizational techniques that streamline communication or limit it in a
54 sensible way.

55 The typical approach taken to streamlining communication is to formalize it in documents.
56 Instead of having 50 people talk to each other in every conceivable
57 combination, 50 people read and write documents. Some are text documents;
58 some are graphic. Some are printed on paper; others are kept in electronic form.

59 **27.2 Range of Project Sizes**

60 Is the size of the project you're working on typical? The wide range of project
61 sizes means that you can't consider any single size to be typical. One way of
62 thinking about project size is to think about the size of a project team. Here's a
63 crude estimate of the percentages of all projects that are done by teams of
64 various sizes:

Team Size	Approximate Percentage of Projects
1-3	25%
4-10	30%
11-25	20%
26-50	15%
50+	10%

65 *Source: Adapted from "A Survey of Software Engineering Practice: Tools, Methods,
66 and Results" (Beck and Perkins 1983), Agile Software Development Ecosystems
67 (Highsmith 2002), and Balancing Agility and Discipline (Boehm and Turner 2003).*

68 One aspect of data on project size that might not be immediately apparent is the
69 difference between the percentage of projects of various sizes and the percentage
70 of programmers who work on projects of each size. Since larger projects use
71 more programmers on each project than do small ones, they can make up a small
72 percentage of the number of projects and still employ a large percentage of all
73 programmers. Here's a rough estimate of the percentage of all programmers who
74 work on projects of various sizes:

Team Size	Approximate Percentage of Programmers
1-3	5%
4-10	10%
11-25	15%
26-50	20%
50+	50%

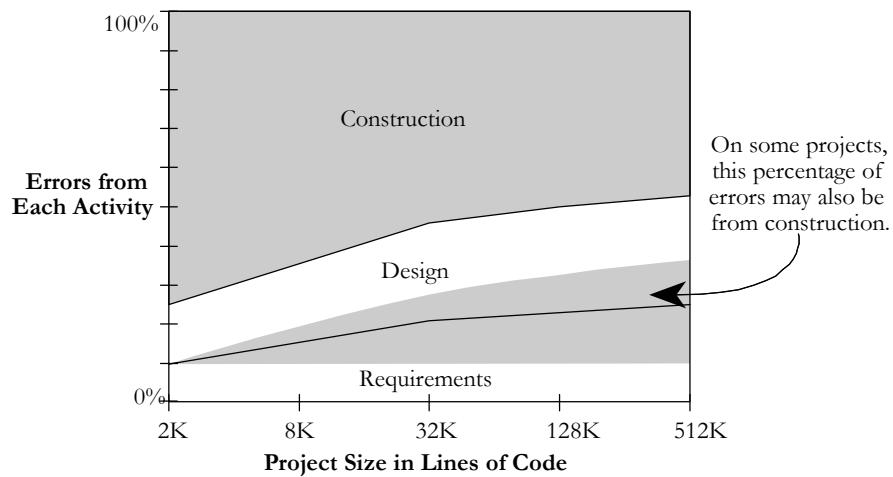
75
76
77
78

Source: Derived from data in “A Survey of Software Engineering Practice: Tools,
Methods, and Results” (Beck and Perkins 1983), Agile Software Development
Ecosystems (Highsmith 2002), and Balancing Agility and Discipline (Boehm and
Turner 2003).

79 27.3 Effect of Project Size on Errors

80 **CROSS-REFERENCE** for
81 more details on errors, see
82 Section 22.4, “Typical
83 Errors.”

Both the quantity and the kinds of errors are affected by project size. You might
not think that the kinds of errors would be affected, but as project size increases,
a larger percentage of errors can usually be attributed to mistakes in
requirements and design. Here’s an illustration:



84 F27xx02

85 Figure 27-2

86 As project size increases, errors usually come more from requirements and design.
87 Sometimes they still come primarily from construction.

88

89 Sources: Software Engineering Economics (Boehm 1981), “Measuring and
90 Managing Software Maintenance” (Grady 1987), and Estimating Software Costs
91 (Jones 1998).

92 On small projects, construction errors make up about 75 percent of all the errors
93 found. Methodology has less influence on code quality, and the biggest influence
94 on program quality is often the skill of the individual writing the program (Jones
95 1998).

96 On larger projects, construction errors can taper off to about 50 percent of the
97 total errors; requirements and architecture errors make up the difference.
98 Presumably this is related to the fact that more requirements development and

99
100
101
102
103

architectural design are required on large projects, so the opportunity for errors
arising out of those activities is proportionally larger. In some very large
projects, however, the proportion of construction errors remains high; sometimes
even with 500,000 lines of code, up to 75 percent of the errors can be attributed
to construction (Grady 1987).

104 **KEY POINT**

105
106
107
108
109

As the kinds of defects change with size, so do the numbers of defects. You
would naturally expect a project that's twice as large as another to have twice as
many errors. But the density of defects, the number of defects per line of code,
increases. The product that's twice as large is likely to have more than twice as
many errors. Table 27-1 shows the range of defect densities you can expect on
projects of various sizes:

110 **Table 27-1. Project Size and Error Density**

Project Size (in Lines of Code)	Error Density
Smaller than 2K	0-25 errors per thousand lines of code (KLOC)
2K-16K	0-40 errors per KLOC
16K-64K	0.5-50 errors per KLOC
64K-512K	2-70 errors per KLOC
512K or more	4-100 errors per KLOC

111
112

CROSS-REFERENCE The
data in this table represents
average performance. A
handful of organizations have
reported better error rates
than the minimums shown
here. For examples, see
“How Many Errors Should
You Expect to Find?” in
Section 22.4.

113
114
115
116
117
118
119
120

*Source: “Program Quality and Programmer Productivity” (Jones 1977), Estimating
Software Costs (Jones 1998).*

121
122
123
124
125
126

The data in this table was derived from specific projects, and the numbers may
bear little resemblance to those for the projects you've worked on. As a snapshot
of the industry, however, the data is illuminating. It indicates that the number of
errors increases dramatically as project size increases, with very large projects
having up to four times as many errors per line of code as small projects. The
data also implies that up to a certain size, it's possible to write error-free code;
above that size, errors creep in regardless of the measures you take to prevent
them.

121 **27.4 Effect of Project Size on Productivity**

122
123
124
125
126

Productivity has a lot in common with software quality when it comes to project
size. At small sizes (2000 lines of code or smaller), the single biggest influence
on productivity is the skill of the individual programmer (Jones 1998). As
project size increases, team size and organization become greater influences on
productivity.

127 **HARD DATA**

128
129
130
131

How big does a project need to be before team size begins to affect productivity? In "Prototyping Versus Specifying: a Multiproject Experiment," Boehm, Gray, and Seewaldt reported that smaller teams completed their projects with 39 percent higher productivity than larger teams. The size of the teams? Two people for the small projects, three for the large (1984).

132
133

Table 27-2 gives the inside scoop on the general relationship between project size and productivity.

134 **Table 27-2. Project Size and Productivity**

Project Size (in Lines of Code)	Lines of Code per Staff-Year (Cocomo II nominal in parentheses)
1K	2,500–25,000 (4,000)
10K	2,000–25,000 (3,200)
100K	1,000–20,000 (2,600)
1,000K	700–10,000 (2,000)
10,000K	300–5,000 (1,600)

135
136
137
138

Source: Derived from data in Measures for Excellence (Putnam and Meyers 1992), Industrial Strength Software (Putnam and Meyers 1997), Software Cost Estimation with Cocomo II (Boehm et al, 2000), and "Software Development Worldwide: The State of the Practice" (Cusumano et al 2003).

139
140
141
142
143
144

Productivity is substantially determined by the kind of software you're working on, personnel quality, programming language, methodology, product complexity, programming environment, tool support, how "lines of code" are counted, how non-programmer support effort is factored into the "lines of code per staff-year" figure, and many other factors, so the specific figures in Table 27-2 vary dramatically.

145
146
147
148

Realize, however, that the general trend the numbers show is significant. Productivity on small projects can be 2-3 times as high as productivity on large projects, and productivity can vary by a factor of 5-10 from the smallest projects to the largest.

149
150

27.5 Effect of Project Size on Development Activities

151
152
153

If you are working on a 1-person project, the biggest influence on the project's success or failure is you. If you're working on a 25-person project, it's conceivable that you're still the biggest influence, but it's more likely that no one

154 person will wear the medal for that distinction; your organization will be a
155 stronger influence on the project's success or failure.

156 **Activity Proportions and Size**

157 As project size increases and the need for formal communications increases, the
158 kinds of activities a project needs change dramatically. Here's a chart that shows
159 the proportions of development activities for projects of different sizes:

160 **Error! Objects cannot be created from editing field codes.**

161 **F27xx03**

162 **Figure 27-3**

163 *Construction activities dominate small projects. Larger projects require more
164 architecture, more integration work, and more system testing to succeed.*

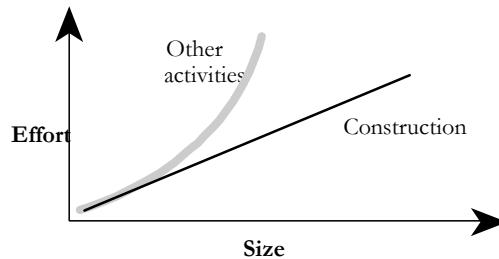
165 *Requirements work is not shown on this diagram because requirements effort is not
166 as directly a function of program size as other activities are.*

167 *Sources: Albrecht 1979; Glass 1982; Boehm, Gray, and Seewaldt 1984; Boddie
168 1987; Card 1987; McGarry, Waligora, and McDermott 1989; Brooks 1995; Jones
169 1998; Jones 2000; Boehm et al. 2000.*

170 **KEY POINT**
171 On a small project, construction is the most prominent activity by far, taking up
172 as much as 65 percent of the total development time. On a medium-size project,
173 construction is still the dominant activity but its share of the total effort falls to
174 about 50 percent. On very large projects, architecture, integration, and system
175 testing take up more time, and construction becomes less dominant. In short, as
176 project size increases, construction becomes a smaller part of the total effort. The
177 chart looks as though you could extend it to the right and make construction
178 disappear altogether, so in the interest of protecting my job, I've cut it off at
512K.

179 Construction becomes less predominant because as project size increases, the
180 construction activities—detailed design, coding, debugging, and unit testing—
181 scale up proportionately but many other activities scale up faster.

182 Here's an illustration:

**F27xx04****Figure 27-4**

The amount of software construction work is a near-linear function of project size. Other kinds of work increase non-linearly as project size increases.

Projects that are close in size will perform similar activities, but as sizes diverge, the kinds of activities will diverge too. As the introduction to this chapter described, when the Gigatron Deluxe comes out at 10 times the size of the original Gigatron, it will need 25 times more construction effort, 25-50 times the planning effort, 30 times the integration effort, and 40 times the architecture and system testing.

Proportions of activities vary because different activities become critical at different project sizes. Barry Boehm and Richard Turner found that spending about 5 percent of total project costs on architecture produced the lowest cost for projects in the 10,000 line-of-code range. But for projects in the 100,000 line-of-code range, spending 15-20 percent of project effort on architecture produced the best results (Boehm and Turner 2004).

Here's a list of activities that grow at a more-than-linear rate as project size increases:

- Communication
- Planning
- Management
- Requirements development
- System functional design
- Interface design and specification
- Architecture
- Integration
- Defect removal
- System testing

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

- 212 • Document production

213 Regardless of the size of a project, a few techniques are always valuable:
214 disciplined coding practices, design and code inspections by other developers,
215 good tool support, and use of high-level languages. These techniques are
216 valuable on small projects and invaluable on large projects.

217 Programs, Products, Systems, and System 218 Products

219 **FURTHER READING** For
220 another explanation of this
221 point, see Chapter 1 in *the*
Mythical Man-Month
222 (Brooks 1995).

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239 **HARD DATA**

240

241

242

243

244

245

246

247

Lines of code and team size aren't the only influences on a project's size. A more subtle influence is the quality and the complexity of the final software. The original Gigatron, the Gigatron Jr., might have taken only a month to write and debug. It was a single program written, tested, and documented by a single person. If the 2,500-line Gigatron Jr. took one month, why did the full-fledged 25,000-line Gigatron take 20?

The simplest kind of software is a single "program" that's used by itself by the person who developed it or, informally, by a few others.

A more sophisticated kind of program is a software "product," a program that's intended for use by people other than the original developer. A software product is used in environments that differ to lesser or greater extents from the environment in which it was created. It's extensively tested before it's released, it's documented, and it's capable of being maintained by others. A software product costs about three times as much to develop as a software program.

Another level of sophistication is required to develop a group of programs that work together. Such a group is called a software "system." Development of a system is more complicated than development of a simple program because of the complexity of developing interfaces among the pieces and the care needed to integrate the pieces. On the whole, a system also costs about three times as much as a simple program.

When a "system product" is developed, it has the polish of a product and the multiple parts of a system. System products cost about nine times as much as simple programs (Brooks 1995, Shull et al 2002).

A failure to appreciate the differences in polish and complexity among programs, products, systems, and system products is one common cause of estimation errors. Programmers who use their experience in building a program to estimate the schedule for building a system product can underestimate by a factor of almost 10. As you consider the following example, refer to the chart on page TBD. If you used your experience in writing 2K lines of code to estimate the

248 time it would take you to develop a 2K program, your estimate would be only 65
249 percent of the total time you'd actually need to perform all the activities that go
250 into developing a program. Writing 2K lines of code doesn't take as long as
251 creating a whole program that contains 2K lines of code. If you don't consider
252 the time it takes to do nonconstruction activities, development will take 50
253 percent more time than you estimate.

254 As you scale up, construction becomes a smaller part of the total effort in a
255 project. If you base your estimates solely on construction experience, the
256 estimation error increases. If you used your own 2K construction experience to
257 estimate the time it would take to develop a 32K program, your estimate would
258 be only 50 percent of the total time required; development would take 100
259 percent more time than you would estimate.

260 The estimation error here would be completely attributable to your not
261 understanding the effect of size on developing larger programs. If in addition
262 you failed to consider the extra degree of polish required for a product rather
263 than a mere program, the error could easily increase by a factor of 3 or more.

264 **Methodology and Size**

265 Methodologies are used on projects of all sizes. On small projects,
266 methodologies tend to be casual and instinctive. On large projects, they tend to
267 be rigorous and carefully planned.

268 Some methodologies can be so loose that programmers aren't even aware that
269 they're using them. A few programmers argue that methodologies are too rigid
270 and say that they won't touch them. While it may be true that a programmer
271 hasn't selected a methodology consciously, any approach to programming
272 constitutes a methodology, no matter how unconscious or primitive the approach
273 is. Merely getting out of bed and going to work in the morning is a rudimentary
274 methodology though not a very creative one. The programmer who insists on
275 avoiding methodologies is really only avoiding choosing one explicitly—no one
276 can avoid using them altogether.

277 **KEY POINT**

278 Formal approaches aren't always fun, and if they are misapplied, their overhead
279 gobbles up their other savings. The greater complexity of larger projects,
280 however, requires a greater conscious attention to methodology. Building a
281 skyscraper requires a different approach than building a doghouse. Different
282 sizes of software projects work the same way. On large projects, unconscious
283 choices are inadequate to the task. Successful project planners choose their
strategies for large projects explicitly.

284 In social settings, the more formal the event, the more uncomfortable your
285 clothes have to be (high heels, neckties, and so on). In software development, the
286 more formal the project, the more paper you have to generate to make sure
287 you've done your homework. Capers Jones points out that a project of 1,000
288 lines of code will average about 7% of its effort on paperwork, whereas a
289 100,000 line of code project will average about 26% of its effort on paperwork
290 (Jones 1998).

291 This paperwork isn't created for the sheer joy of writing documents. It's created
292 as a direct result of the phenomenon illustrated in Figure 27-1—the more
293 people's brains you have to coordinate, the more formal documentation you need
294 to coordinate them.

295 You don't create any of this documentation for its own sake. The point of
296 writing a configuration-management plan, for example, isn't to exercise your
297 writing muscles. The point of your writing the plan is to force you to think
298 carefully about configuration management and to force you to explain your plan
299 to everyone else. The documentation is a tangible side effect of the real work you
300 do as you plan and construct a software system. If you feel as though you're
301 going through the motions and writing generic documents, something is wrong.

302 **KEY POINT**
303 "More" is not better, as far as methodologies are concerned. In their review of
304 agile vs. plan-driven methodologies, Barry Boehm and Richard Turner caution
305 that you'll usually do better if you start your methods small and scale up for a
306 large project than if you start with an all-inclusive method and pare it down for a
307 small project (Boehm and Turner 2004). Some software pundits talk about
308 "lightweight" and "heavyweight" methodologies, but in practice the key is to
309 consider your project's specific size and type and then find the methodology
that's "right-weight."

CC2E.COM/2768

310 Additional Resources

311 Boehm, Barry and Richard Turner. *Balancing Agility and Discipline: A Guide*
312 for the Perplexed, Boston, Mass.: Addison Wesley, 2004. Boehm and Turner
313 describe how project size affects the use of agile and plan-driven methods, along
314 with other agile and plan-driven issues.

315 Boehm, Barry W. 1981. *Software Engineering Economics*. Englewood Cliffs,
316 N.J.: Prentice Hall. Boehm's book is an extensive treatment of the cost and
317 productivity, and quality ramifications of project size and other variables in the
318 software-development process. It includes discussions of the effect of size on
319 construction and other activities. Chapter 11 is an excellent explanation of
320 software's diseconomies of scale. Other information on project size is spread

321 throughout the book. Boehm's 2000 book *Software Cost Estimation with*
322 *Cocomo II* contains much more up-to-date information on Boehm's Cocomo
323 estimating model, but the earlier book provides more in-depth background
324 discussions that are still relevant.

325 Jones, Capers, *Estimating Software Costs*, New York: McGraw-Hill, 1998. This
326 book is packed with tables and graphs the dissect the sources of software
327 development productivity. For the impact of project size specifically, Jones's
328 1986 book *Programming Productivity* contains an excellent discussion in the
329 section titled "The Impact of Program Size" in Chapter 3.

330 Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software*
331 *Engineering, Anniversary Edition (2nd Ed)*, Reading, Mass.: Addison-Wesley,
332 1995. Brooks was the manager of IBM's OS/360 development, a mammoth
333 project that took 5000 staff-years. He discusses management issues pertaining to
334 small and large teams and presents a particularly vivid account of chief-
335 programmer teams in this engaging collection of essays.

336 DeGrace, Peter, and Leslie Stahl. *Wicked Problems, Righteous Solutions: a*
337 *Catalogue of Modern Software Engineering Paradigms*. Englewood Cliffs, N.J.:
338 Yourdon Press, 1990. As the title suggests, this book catalogs approaches to
339 developing software. As noted throughout this chapter, your approach needs to
340 vary as the size of the project varies, and DeGrace and Stahl make that point
341 clearly. The section titled "Attenuating and Truncating" in Chapter 5 discusses
342 customizing software-development processes based on project size and
343 formality. The book includes descriptions of models from NASA and the
344 Department of Defense and a remarkable number of edifying illustrations.

345 Jones, T. Capers. "Program Quality and Programmer Productivity." *IBM*
346 *Technical Report TR 02.764* (January 1977): 42-78. Also available in Jones's
347 *Tutorial: Programming Productivity: Issues for the Eighties*, 2d ed., Los
348 Angeles: IEEE Computer Society Press, 1986. This paper contains the first in-
349 depth analysis of the reasons large projects have different spending patterns than
350 small ones. It's a thorough discussion of the differences between large and small
351 projects, including requirements and quality-assurance measures. It's dated, but
352 still interesting.

353 Key Points

- 354 • As project size increases, communication needs to be supported. The point
355 of most methodologies is to reduce communications problems, and a
356 methodology should live or die on its merits as a communication facilitator.

- 357 • All other things being equal, productivity will be lower on a large project
358 than on a small one.
- 359 • All other things being equal, a large project will have more errors per line of
360 code than a small one.
- 361 • Activities that are taken for granted on small projects must be carefully
362 planned on larger ones. Construction becomes less predominant as project
363 size increases.
- 364 • Scaling-up a light-weight methodology tends to work better than scaling
365 down a heavy-weight methodology. The most effective approach of all is
366 using a “right-weight” methodology.

28

Managing Construction

3 CC2E.COM/2836

Contents

- 4 28.1 Encouraging Good Coding
- 5 28.2 Configuration Management
- 6 28.3 Estimating a Construction Schedule
- 7 28.4 Measurement
- 8 28.5 Treating Programmers as People
- 9 28.6 Managing Your Manager

Related Topics

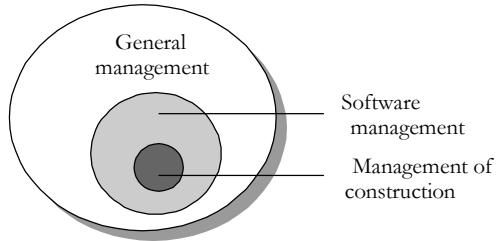
10 Prerequisites to construction: Chapter 3

11 Determining the kind of software you're working on: Section 3.2

12 Program size: Chapter 27

13 Software quality: Chapter 20

14
15 MANAGING SOFTWARE DEVELOPMENT HAS BEEN a formidable
16 challenge for the past several decades. As Figure 28-1 suggests, the general topic
17 of software-project management extends beyond the scope of this book, but this
18 chapter discusses a few specific management topics that apply directly to
19 construction. If you're a developer, this section will help you understand the
20 issues that managers need to consider. If you're a manager, this section will help
21 you understand how management looks to developers as well as how to
22 manage construction effectively. Because the chapter covers a broad collection
23 of topics, several sections also describe where you can go for more information.



F28xx01

Figure 28-1

This chapter covers the software-management topics related to construction.

If you're interested in software management, be sure to read Section 3.2, "Determine the Kind of Software You're Working On," to understand the difference between traditional sequential approaches to development and modern iterative approaches. Be sure also to read Chapter 20, "The Software-Quality Landscape" and Chapter 27, "How Program Size Affects Construction." Quality goals and the size of the project both significantly affect how a specific software project should be managed.

28.1 Encouraging Good Coding

Since code is the primary output of construction, a key question in managing construction is "How do you encourage good coding practices?" In general, mandating a strict set of technical standards from the management position isn't a good idea. Programmers tend to view managers as being at a lower level of technical evolution, somewhere between single-celled organisms and the woolly mammoths that died out during the Ice Age, and if there are going to be programming standards, programmers need to buy into them.

If someone on a project is going to define standards, have a respected architect define the standards rather than the manager. Software projects operate as much on an "expertise hierarchy" as on an "authority hierarchy." If the architect is regarded as the project's thought leader, the project team will generally follow standards set by the architect.

If you choose this approach, be sure the architect really is respected. Sometimes a project architect is just a senior person who has been around too long and is out of touch with production-coding issues. Programmers will resent that kind of "architect" defining standards that are out of touch with the work they're doing.

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52 Considerations in Setting Standards

53 Standards are more useful in some organizations than in others. Some developers
54 welcome standards because they reduce arbitrary variance in the project. If your
55 group resists adopting strict standards, consider a few alternatives: flexible
56 guidelines, a collection of suggestions rather than guidelines, or a set of
57 examples that embody the best practices.

58 Techniques

59 Here are several techniques for achieving good coding practices that are less
60 heavy-handed than laying down rigid coding standards:

61 *Assign two people to every part of the project*

62 **CROSS-REFERENCE** For
63 more details on pair
64 programming, see Section
65 21.2, "Pair Programming."

66 *Review every line of code*

67 A code review typically involves the programmer and at least two reviewers.
68 That means that at least three people read every line of code. Another name for
69 peer review is "peer pressure." In addition to providing a safety net in case the
70 original programmer leaves the project, reviews improve code quality because
71 the programmer knows that the code will be read by others. Even if your shop
72 hasn't created explicit coding standards, reviews provide a subtle way of moving
73 toward a group coding standard—decisions are made by the group during
74 reviews, and, over time, the group will derive its own standards.

75 *Require code sign-offs*

76 In other fields, technical drawings are approved and signed by the managing
77 engineer. The signature means that to the best of the engineer's knowledge, the
78 drawings are technically competent and error-free. Some companies treat code
79 the same way. Before code is considered to be complete, senior technical
80 personnel must sign the code listing.

81 *Route good code examples for review*

82 A big part of good management is communicating your objectives clearly. One
83 way to communicate your objectives is to circulate good code to your
84 programmers or post it for public display. In doing so, you provide a clear
85 example of the quality you're aiming for. Similarly, a coding-standards manual
86 can consist mainly of a set of "best code listings." Identifying certain listings as
87 "best" sets an example for others to follow. Such a manual is easier to update
88 than an English-language standards manual and effortlessly presents subtleties in
89 coding style that are hard to capture point by point in prose descriptions.

90 **CROSS-REFERENCE** A
91 large part of programming is
92 communicating your work to
93 other people. For details, see
94 Section 33.5,
95 “Communication and
96 Cooperation” and Section
97 34.3, “Write Programs for
98 **HARD DATA**

99 Second.”

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

Emphasize that code listings are public assets

Programmers sometimes feel that the code they’ve written is “their code,” as if it were private property. Although it is the result of their work, code is part of the project and should be freely available to anyone else on the project that needs it. It should be seen by others during reviews and maintenance, even if at no other time.

One of the most successful projects ever reported developed 83,000 lines of code in 11 work-years of effort. Only one error that resulted in system failure was detected in the first 13 months of operation. This accomplishment is even more dramatic when you realize that the project was completed in the late 1960s, without online compilation or interactive debugging. Productivity on the project, 7500 lines of code per work-year in the late 1960s, is still impressive by today’s standards. The chief programmer on the project reported that one key to the project’s success was the identification of all computer runs (erroneous and otherwise) as public rather than private assets (Baker and Mills 1973). This idea has extended into modern contexts including Extreme Programming’s idea of collective ownership (Beck 2000), as well as in other contexts.

Reward good code

Use your organization’s reward system to reinforce good coding practices. Keep these considerations in mind as you develop your reinforcement system:

- The reward should be something that the programmer wants. (Many programmers find “attaboy” rewards distasteful, especially when they come from nontechnical managers.)
- Code that receives an award should be exceptionally good. If you give an award to a programmer everyone else knows does bad work, you look like Charlie Chaplin trying to run a cake factory. It doesn’t matter that the programmer has a cooperative attitude or always comes to work on time. You lose credibility if your reward doesn’t match the technical merits of the situation. If you’re not technically skilled enough to make the good-code judgment, don’t! Don’t make the award at all, or let your team choose the recipient.

One easy standard

If you’re managing a programming project and you have a programming background, an easy and effective technique for eliciting good work is to say “I must be able to read and understand any code written for the project.” That the manager isn’t the hottest technical hotshot can be an advantage in that it may discourage “clever” or tricky code.

127

128

129

130

131

132

133

The Role of This Book

Most of this book is a discussion of good programming practices. It isn't intended to be used to justify rigid standards, and it's intended even less to be used as a set of rigid standards. Using it in such a way would contradict some of its most important themes. Use this book as a basis for discussion, as a sourcebook of good programming practices, and for identifying practices that could be beneficial in your environment.

134

28.2 Configuration Management

135

136

137

138

139

A software project is dynamic. The code changes; the design changes; the requirements change. What's more, changes in the requirements lead to more changes in the design; changes in the design lead to even more changes in the code and test cases.

What Is Configuration Management?

140

141

142

143

144

Configuration management is the practice of identifying project artifacts and handling changes systematically so that a system can maintain its integrity over time. Another name for it is "change control." It includes techniques for evaluating proposed changes, tracking changes, and keeping copies of the system as it existed at various points in time.

145

146

147

148

149

If you don't control changes to requirements, you can end up writing code for parts of the system that are eventually eliminated. You can write code that's incompatible with new parts of the system. You might not detect many of the incompatibilities until integration time, which will become finger-pointing time because nobody will really know what's going on.

150

151

152

153

154

155

156

If changes to code aren't controlled, you might change a routine that someone else is changing at the same time; successfully combining your changes with theirs will be problematic. Uncontrolled code changes can make code seem more tested than it is. The version that's been tested will probably be the old, unchanged version; the modified version might not have been tested. Without good change control, you can make changes to a routine, find new errors, and not be able to back up to the old, working routine.

157

158

159

160

161

The problems go on indefinitely. If changes aren't handled systematically, you're taking random steps in the fog rather than moving directly toward a clear destination. Without good change control, rather than developing code you're wasting your time thrashing. Configuration management helps you use your time effectively.

162 | HARD DATA

163
164
165
166
167
168

In spite of the obvious need for configuration management, many programmers have been avoiding it for decades. A survey more than 20 years ago found that over a third of programmers weren't even familiar with the idea (Beck and Perkins 1983), and there's little indication that that has changed. A more recent study by the Software Engineering Institute found that, of organizations using informal software development practices, less than 20% had adequate configuration management (SEI 2003).

169
170
171
172
173

Configuration management wasn't invented by programmers. But because programming projects are so volatile, it's especially useful to programmers. Applied to software projects, configuration management is usually called software configuration management, or SCM (commonly pronounced "scum"). SCM focuses on a program's source code, documentation, and test data.

174
175
176
177
178
179

The systemic problem with SCM is overcontrol. The surest way to stop auto accidents is to prevent everyone from driving, and one sure way to prevent software-development problems is to stop all software development. Although that's one way to control changes, it's a terrible way to develop software. You have to plan SCM carefully so that it's an asset rather than an albatross around your neck.

180 **CROSS-REFERENCE** For details on the effects of project size on construction, see Chapter 27, "How Program Size Affects Construction."

185
186

On a small 1-person project, you can probably do well with no SCM beyond planning for informal periodic backups. Nonetheless, configuration management is still useful (and, in fact, I used configuration management in creating this manuscript). On a large 50-person project, you'll probably need a full-blown SCM scheme including fairly formal procedures for backups, change control for requirements and design, and control over documents, source code, content, test cases, and other project artifacts.

187
188
189

If your project is neither very large nor very small, you'll have to settle on a degree of formality somewhere between the two extremes. The following subsections describe some of the options in implementing SCM.

190

191 **CROSS-REFERENCE** Some development approaches support changes better than others. For details, see Section 3.2, "Determine the Kind of Software You're Working On."

196
197
198

Requirements and Design Changes

During development, you're bound to be bristling with ideas about how to improve the system. If you implement each change as it occurs to you, you'll soon find yourself walking on a software treadmill—for all that the system will be changing, it won't be moving closer to completion. Here are some guidelines for controlling design changes:

Follow a systematic change-control procedure

As Section 3.4 noted, a systematic change-control procedure is a godsend when you have a lot of change requests. By establishing a systematic procedure, you

199
200 make it clear that changes will be considered in the context of what's best for the
project overall.

201
202 ***Handle change requests in groups***
203 It's tempting to implement easy changes as ideas arise. The problem with
204 handling changes in this way is that good changes can get lost. If you think of a
205 simple change 25 percent of the way through the project and you're on schedule,
206 you'll make the change. If you think of another simple change 50 percent of the
207 way through the project and you're already behind, you won't. When you start to
208 run out of time at the end of the project, it won't matter that the second change is
209 10 times as good as the first—you won't be in a position to make any
210 nonessential changes. Some of the best changes can slip through the cracks
merely because you thought of them later rather than sooner.

211
212 The informal solution to this problem is to write down all ideas and suggestions,
213 no matter how easy they would be to implement, and save them until you have
214 time to work on them. Then, viewing them as a group, choose the ones that will
be the most beneficial.

215
216 ***Estimate the cost of each change***
217 Whenever your customer, your boss, or you are tempted to change the system,
218 estimate the time it would take to make the change, including review of the code
219 for the change and retesting the whole system. Include in your estimate time for
220 dealing with the change's ripple effect through requirements to design to code to
221 test to changes in the user documentation. Let all the interested parties know that
222 software is intricately interwoven and that time estimation is necessary even if
the change appears small at first glance.

223
224 Regardless of how optimistic you feel when the change is first suggested, refrain
225 from giving an off-the-cuff estimate. Hand waving estimates are often mistaken
by a factor of 2 or more.

226 **CROSS-REFERENCE** For
227 another angle on handling
228 changes, see "Handling
229 Requirements Changes
230 During Construction" in
Section 3.4.
231
232

While some degree of change is inevitable, a high volume of change requests is a key warning sign that requirements, architecture, or top-level designs weren't done well enough to support effective construction. Backing up to work on requirements or architecture might seem expensive, but it won't be nearly as expensive as constructing the software more than once or as throwing away code for features that you really didn't need.

233
234 ***Establish a change-control board or its equivalent in a way that makes***
sense for your project
235 The job of a change-control board is to separate the wheat from the chaff in
236 change requests. Anyone who wants to propose a change submits the change
237 request to the change-control board. The term "change request" refers to any

238 request that would change the software: an idea for a new feature, a change to an
239 existing feature, an “error report” that might or might not be reporting a real
240 error, and so on. The board meets periodically to review proposed changes. It
241 approves, disapproves, or defers each change. Change control boards are
242 considered a best practice for prioritizing and controlling requirements changes,
243 however they are still fairly uncommon in commercial settings (Jones 1998,
244 Jones 2000).

245 ***Watch for bureaucracy, but don't let the fear of bureaucracy preclude
246 effective change control***

247 Lack of disciplined change control is one of the biggest management problems
248 facing the software industry today. A significant percentage of the projects that
249 are perceived to be late would actually be on time if they accounted for the
250 impact of untracked but agreed-upon changes. Poor change control allows
251 changes to accumulate off the books, which undermines status visibility, long-
252 range predictability, project planning, risk management specifically, and project
253 management generally.

254 Change control tends to drift toward bureaucracy, and so it's important to look
255 for ways to streamline the change control process. If you'd rather not use
256 traditional change requests, set up a simple “ChangeBoard” email alias and have
257 people email change requests to that email address. Or have people present
258 change proposals interactively at a change board meeting. Or log change
259 requests as defects in your defect tracking software (classified as changes rather
260 than defects).

261 You can implement the Change Control Board itself formally. Or you can define
262 a Product Planning Group or War Council that carries the traditional
263 responsibilities of a change control board. You can identify a single person to be
264 the Change Czar. But whatever you call it, do it!

265 **KEY POINT**
266 I occasionally see projects suffering from ham-handed implementations of
267 change control. But 10 times as often I see projects suffering from no meaningful
268 change control at all. The substance of change control is what's important, so
don't let fear of bureaucracy stop you from realizing its many benefits.

269 Software Code Changes

270 Another configuration-management issue is controlling source code. If you
271 change the code and a new error surfaces that seems unrelated to the change you
272 made, you'll probably want to compare the new version of the code to the old in
273 your search for the source of the error. If that doesn't tell you anything, you
274 might want to look at a version that's even older. This kind of excursion through

275
276

history is easy if you have version-control tools that keep track of multiple versions of source code.

277 **KEY POINT**

278
279
280
281
282
283
284
285
286
287
288

Version-control software

Good version-control software works so easily that you barely notice you're using it. It's especially helpful on team projects. One style of version control locks source files so that only one person can modify a file at a time. Typically, when you need to work on source code in a particular file, you check the file out of version control. If someone else has already checked it out, you're notified that you can't check it out. When you can check the file out, you work on it just as you would without version control until you're ready to check it in. Another style allows multiple people to work on files simultaneously, and handles the issue of merging changes when the code is checked in. In either case, when you check the file in, version control asks why you changed it, and you type in a reason.

289

For this modest investment of effort, you get several big benefits:

290
291
292
293
294
295

296
297
298

299
300
301

- You don't step on anyone's toes by working on a file while someone else is working on it (or at least you'll know about it if you do).
- You can easily update your copies of all the project's files to the current versions, usually by issuing a single command.
- You can backtrack to any version of any file that was ever checked into version control.
- You can get a list of the changes made to any version of any file.
- You don't have to worry about personal backups because the version-control copy is a safety net.

302

Version control is indispensable on team projects. It's so effective that the applications division of Microsoft has found source-code version control to be a "major competitive advantage" (Moore 1992).

Tool Versions

303
304
305
306

For some kinds of projects, it may be necessary to be able to reconstruct the exact environment used to create each specific version of the software—including compilers, linkers, code libraries, and so on. In that case, you will want to put all of those tools into version control, too.

307
308
309

Machine Configurations

Many companies (including my company) have experienced good results from creating standardized development machine configurations. A disk image is

310 created of a standard developer workstation, including all the common developer
311 tools, office applications, and so on. That image is loaded onto each developer's
312 machine. Having standardized configurations helps to avoid a raft of problems
313 associated with slightly different configuration settings, different versions of
314 tools used, and so on. A standardized disk image also greatly streamlines setting
315 up new machines compared to having to install each piece of software
316 individually.

317 **Backup Plan**

318 A backup plan isn't a dramatic new concept; it's the idea of backing up your
319 work periodically. If you were writing a book by hand, you wouldn't leave the
320 pages in a pile on your porch. If you did, they might get rained on or blown
321 away, or your neighbor's dog might borrow them for a little bedtime reading.
322 You'd put them somewhere safe. Software is less tangible, so it's easier to forget
323 that you have something of enormous value on one machine.

324 Many things can happen to computerized data. A disk can fail. You or someone
325 else can delete key files accidentally. An angry employee can sabotage your
326 machine. You could lose a machine to theft, flood, or fire.

327 Take steps to safeguard your work. Your backup plan should include making
328 backups on a periodic basis and periodic transfer of backups to off-site storage,
329 and it should encompass all the important materials on your project—documents,
330 graphics, and notes—in addition to source code.

331 One often-overlooked aspect of devising a backup plan is a test of your backup
332 procedure. Try doing a restore at some point to make sure that the backup
333 contains everything you need and that the recovery works.

334 When you finish a project, make a project archive. Save a copy of everything:
335 source code, compilers, tools, requirements, design, documentation—everything
336 you need to re-create the product. Keep it all in a safe place.

CC2E.COM/2843

337 **CHECKLIST: Configuration Management**

338 **General**

- 339 Is your software-configuration-management plan designed to help
340 programmers and minimize overhead?
- 341 Does your SCM approach avoid overcontrolling the project?
- 342 Do you group change requests, either through informal means such as a list
343 of pending changes or through a more systematic approach such as a
344 change-control board?

- 345 Do you systematically estimate the effect of each proposed change?
346 Do you view major changes as a warning that requirements development
347 isn't yet complete?

348 **Tools**

- 349 Do you use version-control software to facilitate configuration management?
350 Do you use version-control software to reduce coordination problems of
351 working in teams?

352 **Backup**

- 353 Do you back up all project materials periodically?
354 Are project backups transferred to off-site storage periodically?
355 Are all materials backed up, including source code, documents, graphics,
356 and important notes?
357 Have you tested the backup-recovery procedure?

358 CC2E.COM/2850

359 **Additional Resources on Configuration
360 Management**

361 Because this book is about construction, this section has focused on change
362 control from a construction point of view. But changes affect projects at all
363 levels, and a comprehensive change-control strategy needs to do the same.

364 Hass, Anne Mette Jonassen, *Configuration Management Principles and*
365 *Practices*, Boston, Mass.: Addison Wesley, 2003. This book provides the big-
366 picture view of software configuration management and practical details on how
367 to incorporate it into your software development process. It focuses on managing
368 and controlling configuration items.

369 Berczuk, Stephen P. and Brad Appleton, *Software Configuration Management*
370 *Patterns: Effective Teamwork, Practical Integration*, Boston, Mass.: Addison
371 Wesley, 2003. Like Hass's book, this book provides a CM overview and
372 practical. It complements Hass' book by focusing on branching strategies that
373 allow teams of developers to isolate and coordinate their work.

374 CC2E.COM/2857 SPMN. *Little Book of Configuration Management*. Arlington, VA; Software
375 Program Managers Network, 1998. This pamphlet is an introduction to
376 configuration management activities and defines critical success factors. It is
377 available as a free download from the SPMN website at
378 www.spmn.com/products_guidebooks.html.

379 Bays, Michael, *Software Release Methodology*, Englewood Cliffs, N.J.: Prentice
380 Hall, 1999. This book discusses software configuration management with an
381 emphasis on releasing software into production.

382 Bersoff, Edward H., and Alan M. Davis. "Impacts of Life Cycle Models on
383 Software Configuration Management." *Communications of the ACM* 34, no. 8
384 (August 1991): 104–118. This article describes how SCM is affected by newer
385 approaches to software development, especially prototyping approaches. The
386 article is especially applicable in environments that are using agile development
387 practices.

388 28.3 Estimating a Construction Schedule

389 HARD DATA

390 Managing a software project is one of the formidable challenges of the twenty-
391 first century, and estimating the size of a project and the effort required to
392 complete it is one of the most challenging aspects of software-project
393 management. The average large software project is one year late and 100 percent
394 over budget (Standish Group 1994, Jones 1997, Johnson 1999). This has as
395 much to do with poor size and effort estimates as with poor development efforts.
396 This section outlines the issues involved in estimating software projects and
 indicates where to look for more information.

397 Estimation Approaches

398 For
399 further reading on schedule-
 estimation techniques, see
400 Chapter 8 of *Rapid*
 Development (McConnell
401 1996) and *Software Cost*
402 *Estimation with Cocomo II*
 (Boehm et al 2000).

- 403
- 404
- 405
- 406
- 407
- 408
- 409
- 410
- 411
- Use scheduling software.
 - Use an algorithmic approach, such as Cocomo II, Barry Boehm's estimation model (Boehm et al 2000).
 - Have outside estimation experts estimate the project.
 - Have a walkthrough meeting for estimates.
 - Estimate pieces of the project, and then add the pieces together.
 - Have people estimate their own pieces, and then add the pieces together.
 - Estimate the time needed for the whole project, and then divide up the time among the pieces.
 - Refer to experience on previous projects.
 - Keep previous estimates and see how accurate they were. Use them to adjust new estimates.

412
413
414

415 **FURTHER READING** This
416 approach is adapted from
417 *Software Engineering
Economics* (Boehm 1981).
418
419

420
421
422
423

424 **CROSS-REFERENCE** For
425 more information on software
426 requirements, see Section 3.4,
427 “Requirements Prerequisite.”
428
429

430
431
432
433
434
435
436

437 **CROSS-REFERENCE** It’s
438 hard to find an area of
439 software development in
440 which iteration is not a
441 valuable technique. This is
442 one case in which iteration is
443 useful. For a summary of
444 iterative techniques, see
445 Section 34.8, “Iterate,
Repeatedly, Again and
Again.”

446
447
448
449

Pointers to more information on these approaches are given in the “Additional Resources” subsection at the end of this section. Here’s a good approach to estimating a project:

Establish objectives

What are you estimating? Why do you need an estimate? How accurate does the estimate need to be to meet your objectives? What degree of certainty needs to be associated with the estimate? Would an optimistic or a pessimistic estimate produce substantially different results?

Allow time for the estimate, and plan it

Rushed estimates are inaccurate estimates. If you’re estimating a large project, treat estimation as a miniproject and take the time to miniplan the estimate so that you can do it well.

Spell out software requirements

Just as an architect can’t estimate how much a “pretty big” house will cost, you can’t reliably estimate a “pretty big” software project. It’s unreasonable for anyone to expect you to be able to estimate the amount of work required to build something when “something” has not yet been defined. Define requirements or plan a preliminary exploration phase before making an estimate.

Estimate at a low level of detail

Depending on the objectives you identified, base the estimate on a detailed examination of project activities. In general, the more detailed your examination is, the more accurate your estimate will be. The Law of Large Numbers says that the error of sums is greater than the sum of errors. In other words, a 10 percent error on one big piece is 10 percent high or 10 percent low. On 50 small pieces, 10 percent errors are both high and low and tend to cancel each other out.

Use several different estimation techniques, and compare the results

The list of estimation approaches at the beginning of the section identified several techniques. They won’t all produce the same results, so try several of them. Study the different results from the different approaches.

Children learn early that if they ask each parent individually for a third bowl of ice cream, they have a better chance of getting at least one “yes” than if they ask only one parent. Sometimes the parents wise up and give the same answer; sometimes they don’t. See what different answers you can get from different estimation techniques.

No approach is best in all circumstances, and the differences among them can be illuminating. For example, on the first edition of this book, my original eyeball estimate for the length of the book was 250-300 pages. When I finally did an in-depth estimate, the estimate came out to 873 pages. “That can’t be right,” I

450
451
452
453
454
thought. So I estimated it using a completely different technique. The second
estimate came out to 828 pages. Considering that these estimates were within
about 5 percent of each other, I concluded that the book was going to be much
closer to 850 pages than to 250 pages, and I was able to adjust my writing plans
accordingly.

455
456
457
458
459
460
Re-estimate periodically
Factors on a software project change after the initial estimate, so plan to update
your estimates periodically. As Figure 28-2 illustrates, the accuracy of your
estimates should improve as you move toward completing the project. From time
to time, compare your actual results to your estimated results, and use that
evaluation to refine estimates for the remainder of the project.

461 CC2E.COM/2864
Error! Objects cannot be created from editing field codes.

F28xx02

Figure 28-2

463
464
465
466
467
468
*Estimates created early in a project are inherently inaccurate. As the project
progresses, estimates can become more accurate. Re-estimate periodically
throughout a project. Use what you learn during each activity to improve your
estimate for the next activity. As the project progresses, the accuracy of your
estimates should improve.*

Estimating the Amount of Construction

469
470 **CROSS-REFERENCE** for
471 details on the amount of
472 coding for projects of various
473 sizes, see “Activity
474 Proportions and Size” in
Section 21.2.

475
476
477
478
479
480
Error! Objects cannot be created from editing field codes.

F28xx03

Figure 28-3

481
482
483
484
*Until your company has project-history data of its own, the proportion of time
devoted to each activity shown in the chart is a good place to start estimates for your
projects.*

The best answer to the question of how much construction a project will call for
is that the proportion will vary from project to project and organization to
organization. Keep records of your organization’s experience on projects and use
them to estimate the time future projects will take.

485

486 **CROSS-REFERENCE** The
 487 effect of a program's size on
 488 productivity and quality isn't
 489 always intuitively apparent.
 490 See Chapter 27, "How
 Program Size Affects
 Construction," for an
 explanation of how size
 affects construction.

Influences on Schedule

The largest influence on a software project's schedule is the size of the program to be produced. But many other factors also influence a software-development schedule. Studies of commercial programs have quantified some of the factors, and they're shown in Table 28-1.

Table 28-1. Factors That Influence Software-Project Effort

Factor	Potential Positive Influence	Potential Negative Influence
Co-located vs. multi-site development	-14%	22%
Database size	-10%	28%
Documentation match to project needs	-19%	23%
Flexibility allowed in interpreting requirements	-9%	10%
How actively risks are addressed	-12%	14%
Language and tools experience	-16%	20%
Personnel continuity (turnover)	-19%	29%
Platform volatility	-13%	30%
Process maturity	-13%	15%
Product complexity	-27%	74%
Programmer capability	-24%	34%
Reliability required	-18%	26%
Requirements analyst capability	-29%	42%
Reuse requirements	-5%	24%
State-of-the-art application	-11%	12%
Storage constraint (how much of available storage will be consumed)	0%	46%
Team cohesion	-10%	11%
Team's experience in the applications area	-19%	22%
Team's experience on the technology platform	-15%	19%
Time constraint (of the application itself)	0%	63%
Use of software tools	-22%	17%

Source: Software Cost Estimation with Cocomo II (*Boehm et al 2000*).

491

492 Here are some of the less easily quantified factors that can influence a software-
493 development schedule. These factors are drawn from Barry Boehm's *Software
494 Cost Estimation with Cocomo II* (2000) and Capers Jones's *Estimating Software
495 Costs* (1998).

- 496 • Requirements developer experience and capability
497 • Programmer experience and capability
498 • Team motivation
499 • Management quality
500 • Amount of code reused
501 • Personnel turnover
502 • Requirements volatility
503 • Quality of relationship with customer
504 • User participation in requirements
505 • Customer experience with the type of application
506 • Extent to which programmers participate in requirements development
507 • Classified security environment for computer, programs, and data
508 • Amount of documentation
509 • Project objectives (schedule vs. quality vs. usability vs. the many other
510 possible objectives)

511 Each of these factors can be significant, so consider them along with the factors
512 shown in Table 28-1 (which includes some of these factors).

513 **Estimation vs. Control**

514 *The important question
515 is, do you want
516 prediction, or do you
517 want control?*
518 —Tom Gilb
519

Estimation is an important part of planning to complete a software project on time. Once you have a delivery date and a product specification, the main problem is how to control the expenditure of human and technical resources for an on-time delivery of the product. In that sense, the accuracy of the initial estimate is much less important than your subsequent success at controlling resources to meet the schedule.

520 **What to do If You're Behind**

521 **HARD DATA**
522 Most software projects fall behind. Surveys of estimated vs. actual schedules
523 have shown that estimates tend to have an optimism factor of 20 to 30 percent
(van Genuchten 1991).

524 When you're behind, increasing the amount of time usually isn't an option. If it
525 is, do it. Otherwise, you can try one or more of these solutions:

526 ***Hope that you'll catch up***

527 **HARD DATA**

528 Hopeful optimism is a common response to a project's falling behind schedule.
529 The rationalization typically goes like this: "Requirements took a little longer
530 than we expected, but now they're solid, so we're bound to save time later. We'll
531 make up the shortfall during coding and testing." This is hardly ever the case.
532 One survey of over 300 software projects concluded that delays and overruns
533 generally increase toward the end of a project (van Genuchten 1991). Projects
don't make up lost time later; they fall further behind.

534 ***Expand the team***

535 According to Fred Brooks's law, adding people to a late software project makes
it later (Brooks 1995). It's like adding gas to a fire. Brooks's explanation is
536 convincing: New people need time to familiarize themselves with a project
537 before they can become productive. Their training takes up the time of the
538 people who have already been trained. And merely increasing the number of
539 people increases the complexity and amount of project communication. Brooks
540 points out that the fact that one woman can have a baby in nine months does not
541 imply that nine women can have a baby in one month.

543 Undoubtedly the warning in Brooks's law should be heeded more often than it is.
544 It's tempting to throw people at a project and hope that they'll bring it in on
545 time. Managers need to understand that developing software isn't like riveting
546 sheet metal: More workers working doesn't necessarily mean more work will get
547 done.

548 The simple statement that adding programmers to a late project makes it later,
549 however, masks the fact that under some circumstances it's possible to add
550 people to a late project and speed it up. As Brooks points out in the analysis of
551 his law, adding people to software projects in which the tasks can't be divided
552 and performed independently doesn't help. But if a project's tasks are
553 partitionable, you can divide them further and assign them to different people,
554 even to people who are added late in the project. Other researchers have formally
555 identified circumstances under which you can add people to a late project
556 without making it later (Abdel-Hamid 1989, McConnell 1999).

557 ***Reduce the scope of the project***

558 The powerful technique of reducing the scope of the project is often overlooked.
559 If you eliminate a feature, you eliminate the design, coding, debugging, testing,
560 and documentation of that feature. You eliminate that feature's interface to other
561 features.

562 When you plan the product initially, partition the product's capabilities into
563 "must haves," "nice to haves," and "optionals." If you fall behind, prioritize the
564 "optionals" and "nice to haves" and drop the ones that are the least important.

565 Short of dropping a feature altogether, you can provide a cheaper version of the
566 same functionality. You might provide a version that's on time but that hasn't
567 been tuned for performance. You might provide a version in which the least
568 important functionality is implemented crudely. You might decide to back off on
569 a speed requirement because it's much easier to provide a slow version. You
570 might back off on a space requirement because it's easier to provide a memory-
571 intensive version.

572 Re-estimate development time for the least important features. What
573 functionality can you provide in two hours, two days, or two weeks? What do
574 you gain by building the two-week version rather than the two-day version, or
575 the two-day version rather than the two-hour version?

576 CC2E.COM/2871

Additional Resources on Software Estimation

577 Boehm, Barry, et al, 2000. *Software Cost Estimation with Cocomo II*, Boston,
578 Mass.: Addison Wesley, 2000. This book describes the ins and outs of the
579 Cocomo II estimating model, which is undoubtedly the most popular model in
580 use today.

581 Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs, N.J.:
582 Prentice Hall, 1981. This older book contains an exhaustive treatment of
583 software-project estimation considered more generally than in Boehm's newer
584 book.

585 Humphrey, Watts S. *A Discipline for Software Engineering*. Reading, Mass:
586 Addison Wesley, 1995. Chapter TBD of this book describes Humphrey's Probe
587 method, which is a technique for estimating work at the individual developer
588 level.

589 Conte, S. D., H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics
590 and Models*. Menlo Park, Calif.: Benjamin/Cummings, 1986. Chapter 6 contains
591 a good survey of estimation techniques including a history of estimation,
592 statistical models, theoretically based models, and composite models. The book
593 also demonstrates the use of each estimation technique on a database of projects
594 and compares the estimates to the projects' actual lengths.

595 Gilb, Tom. *Principles of Software Engineering Management*. Wokingham,
596 England: Addison-Wesley, 1988. The title of Chapter 16, "Ten Principles for
597 Estimating Software Attributes," is somewhat tongue-in-cheek. Gilb argues

598 against project estimation and in favor of project control. Pointing out that
599 people don't really want to predict accurately but do want to control final results,
600 Gilb lays out 10 principles you can use to steer a project to meet a calendar
601 deadline, a cost goal, or another project objective.

602 28.4 Measurement

603 Software projects can be measured in numerous ways. Here are two solid
604 reasons to measure your process:

605 **KEY POINT**
606 *For any project attribute, it's possible to measure that attribute in a way
607 that's superior to not measuring it at all*

608 The measurement may not be perfectly precise; it may be difficult to make; it
609 may need to be refined over time; but measurement will give you a handle on
your software-development process that you don't have without it (Gilb 2004).

610 If data is to be used in a scientific experiment, it must be quantified. Can you
611 imagine an FDA scientist recommending a ban on a new food product because a
612 group of white rats "just seemed to get sicker" than another group? That's
613 absurd. You'd demand a quantified reason, like "Rats that ate the new food
614 product were sick 3.7 more days per month than rats that didn't." To evaluate
615 software-development methods, you must measure them. Statements like "This
616 new method seems more productive" aren't good enough.

617 *To argue against measurement is to argue that it's better not to know
618 what's really happening on your project*

619 When you measure an aspect of a project, you know something about it that you
620 didn't know before. You can see whether the aspect gets bigger or smaller or
621 stays the same. The measurement gives you a window into at least that aspect of
622 your project. The window might be small and cloudy until you refine your
623 measurements, but it will be better than no window at all. To argue against all
624 measurements because some are inconclusive is to argue against windows
625 because some happen to be cloudy.

626 You can measure virtually any aspect of the software-development process.
627 Table 28-2 lists some measurements that other practitioners have found to be
628 useful:

629 **Table 28-2. Useful Measurements**

Size	Overall Quality
Total lines of code written	Total number of defects
Total comment lines	Number of defects in each class or routine
Total number of classes or routines	

Total data declarations
Total blank lines

Average defects per thousand lines of code
Mean time between failures
Compiler-detected errors

Productivity

Work-hours spent on the project
Work-hours spent on each class or routine
Number of times each class or routine changed
Dollars spent on project
Dollars spent per line of code
Dollars spent per defect

Maintainability

Number of public routines on each class
Number of parameters passed to each routine
Number of private routines and/or variables on each class
Number of local variables used by each routine
Number of routines called by each class or routine
Number of decision points in each routine
Control-flow complexity in each routine
Lines of code in each class or routine
Lines of comments in each class or routine
Number of data declarations in each class or routine
Number of blank lines in each class or routine
Number of *gosub*s in each class or routine
Number of input or output statements in each class or routine

Defect Tracking

Severity of each defect
Location of each defect (class or routine)
Origin of each defect (requirements, design, construction, test)
Way in which each defect is corrected
Person responsible for each defect
Number of lines affected by each defect correction
Work hours spent correcting each defect
Average time required to find a defect
Average time required to fix a defect
Number of attempts made to correct each defect
Number of new errors resulting from defect correction

630 You can collect most of these measurements with software tools that are
631 currently available. Discussions throughout the book indicate the reasons that
632 each measurement is useful. At this time, most of the measurements aren't useful
633 for making fine distinctions among programs, classes, and routines (Shepperd
634 and Ince 1989). They're useful mainly for identifying routines that are "outliers";
635 abnormal measurements in a routine are a warning sign that you should re-
636 examine that routine, checking for unusually low quality.

637 Don't start by collecting data on all possible measurements—you'll bury
638 yourself in data so complex that you won't be able to figure out what any of it
639 means. Start with a simple set of measurements such as the number of defects,
640 the number of work-months, the total dollars, and the total lines of code.
641 Standardize the measurements across your projects, and then refine them and add
642 to them as your understanding of what you want to measure improves
643 (Pietrasanta 1990).

644 Make sure you're collecting data for a reason. Set goals; determine the questions
645 you need to ask to meet the goals; and then measure to answer the questions
646 (Basili and Weiss 1984). Be sure that you ask for only as much information as is
647 feasible to obtain and that you keep in mind that data collection will always take
648 a back seat to deadlines (Basili et al 2002).

CC2E.COM/2878

649 Additional Resources on Software Measurement

650 Oman, Paul and Shari Lawrence Pfleeger, eds. *Applying Software Metrics*, Los
651 Alamitos, Ca.: IEEE Computer Society Press, 1996. This volume collects more
652 than 25 key papers on software measurement under one cover.

653 Jones, Capers. *Applied Software Measurement: Assuring Productivity and*
654 *Quality, 2d Ed.* New York: McGraw-Hill, 1997. Jones is a leader in software
655 measurement, and his book is an accumulation of knowledge in this area. It
656 provides the definitive theory and practice of current measurement techniques
657 and describes problems with traditional measurements. It lays out a full program
658 for collecting "function-point metrics." Jones has collected and analyzed a huge
659 amount of quality and productivity data, and this book distills the results in one
660 place—including a fascinating chapter on averages for U.S. software
661 development.

662 Grady, Robert B., and Deborah L. Caswell. *Software Metrics: Establishing a*
663 *Company-Wide Program*, Englewood Cliffs, N.J.: Prentice Hall, 1987. Grady
664 and Caswell describe their experience in establishing a software- measurement
665 program at Hewlett-Packard and tell you how to establish a software-
666 measurement program in your organization.

667 Conte, S. D., H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics*
668 and *Models*. Menlo Park, Calif.: Benjamin/Cummings, 1986. This book catalogs
669 current knowledge of software measurement circa 1986, including commonly
670 used measurements, experimental techniques, and criteria for evaluating
671 experimental results.

672 Basili, Victor R., et al., 2002. "Lessons learned from 25 years of process
673 improvement: The Rise and Fall of the NASA Software Engineering
674 Laboratory," *Proceedings of the 24th International Conference on Software*
675 *Engineering*, Orlando, Florida, 2002. This paper catalogs lessons learned by one
676 of the world's most sophisticated software development organizations. The
677 lessons focus on measurement topics.

678 CC2E.COM/2892 NASA Software Engineering Laboratory, *Software Measurement Guidebook*,
679 June 1995, NASA-GB-001-94. This guidebook of about 100 pages is probably
680 the best source of practical information on how to setup and run a measurement
681 program. It can be downloaded from NASA's website.

682 CC2E.COM/2899 Gilb, Tom, 2004. *Competitive Engineering*, Boston, Mass.: Addison Wesley,
683 2004. This book presents a measurement-focused approach to defining
684 requirements, evaluating designs, measuring quality, and, in general, managing
685 projects. It can be downloaded from Gilb's website.

686 28.5 Treating Programmers as People

687 **KEY POINT** The abstractness of the programming activity calls for an offsetting naturalness
688 in the office environment and rich contacts among coworkers. Highly technical
689 companies offer parklike corporate campuses, organic organizational structures,
690 comfortable offices, and other "high-touch" environmental features to balance
691 the intense, sometimes arid intellectuality of the work itself. The most successful
692 technical companies combine elements of high-tech and high-touch (Naisbitt
693 1982). This section describes ways in which programmers are more than organic
694 reflections of their silicon alter egos.

695 How do Programmers Spend Their Time?

696 Programmers spend their time programming, but they also spend time in
697 meetings, on training, on reading their mail, and on just thinking. A 1964 study
698 at Bell Laboratories found that programmers spent their time this way:

699

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

Activity	Source Code	Business	Personal	Meetings	Training	Mail/Misc. Documents	Technical Manuals	Operating Procedures, Misc.	Time
Talk or listen	4%	17%	7%	3%				1%	
Talk with manager		1%							
Telephone		2%	1%						
Read	14%					2%	2%		
Write/record	13%					1%			
Away or out		4%	1%	4%	6%				
Walking	2%	2%	1%			1%			
Miscellaneous	2%	3%	3%			1%		1%	1%
Totals	35%	29%	13%	7%	6%	5%	2%	2%	1%

700

701

Source: "Research Studies of Programmers and Programming" (Bairdain 1964, reported in Boehm 1981).

702

703

704

705

706

707

708

709

710

711

712

This data is based on a time-and-motion study of 70 programmers. The data is old, and the proportions of time spent in the different activities would vary among programmers, but the results are nonetheless illuminating. About 30 percent of a programmer's time is spent in non-technical activities that don't directly help the project: walking, personal business, and so on. Programmers in this study spent 6 percent of their time walking; that's about 2.5 hours a week, about 125 hours a year. That might not seem like much until you realize that programmers spend as much time each year walking as they spend in training, three times as much time as they spend reading technical manuals, and six times as much as they spend talking with their managers. I personally have not seen much change in this pattern today.

713

Variation in Performance and Quality

714 | HARD DATA

715

716

717

718

719

720

721

722

Talent and effort among individual programmers vary tremendously, as they do in all fields. One study found that in a variety of professions—writing, football, invention, police work, and aircraft piloting—the top 20 percent of the people produced about 50 percent of the output (Augustine 1979). The results of the study are based on an analysis of productivity data such as touchdowns, patents, solved cases, and so on. Since some people make no tangible contribution whatsoever (quarterbacks who make no touchdowns, inventors who own no patents, detectives who don't close cases, and so on), the data probably understates the actual variation in productivity.

723
724
725

726

727 **HARD DATA**

728
729
730
731
732
733
734

735 **HARD DATA**

736
737
738
739
740

741
742
743
744
745

746 **HARD DATA**

747
748
749
750
751
752

753
754
755

756
757
758
759
760

In programming specifically, many studies have shown order-of-magnitude differences in the quality of the programs written, the sizes of the programs written, and the productivity of programmers.

Individual Variation

The original study that showed huge variations in individual programming productivity was conducted in the late 1960s by Sackman, Erikson, and Grant (1968). They studied professional programmers with an average of 7 years' experience and found that the ratio of initial coding time between the best and worst programmers was about 20 to 1; the ratio of debugging times over 25 to 1; of program size 5 to 1; and of program execution speed about 10 to 1. They found no relationship between a programmer's amount of experience and code quality or productivity.

Although specific ratios such as 25 to 1 aren't particularly meaningful, more general statements such as "There are order-of-magnitude differences among programmers" are meaningful and have been confirmed by many other studies of professional programmers (Curtis 1981, Mills 1983, DeMarco and Lister 1985, Curtis et al. 1986, Card 1987, Boehm and Papaccio 1988, Valett and McGarry 1989, Boehm et al 2000).

Team Variation

Programming teams also exhibit sizable differences in software quality and productivity. Good programmers tend to cluster, as do bad programmers, an observation that has been confirmed by a study of 166 professional programmers from 18 organizations (Demarco and Lister 1999).

In one study of seven identical projects, the efforts expended varied by a factor of 3.4 to 1 and program sizes by a factor of 3 to 1 (Boehm, Gray, and Seewaldt 1984). In spite of the productivity range, the programmers in this study were not a diverse group. They were all professional programmers with several years of experience who were enrolled in a computer-science graduate program. It's reasonable to assume that a study of a less homogeneous group would turn up even greater differences.

An earlier study of programming teams observed a 5-to-1 difference in program size and a 2.6-to-1 variation in the time required for a team to complete the same project (Weinberg and Schulman 1974).

After reviewing data more than 20 years of data in constructing the Cocomo II estimation model, Barry Boehm and other researchers concluded that developing a program with a team in the 15th percentile of programmers ranked by ability typically requires about 3.5 times as many work-months as developing a program with a team in the 90th percentile (Boehm et al 2000). Boehm and other

761 researchers have found that 80 percent of the contribution comes from 20 percent
762 of the contributors (Boehm 1987b).

763 The implication for recruiting and hiring is clear. If you have to pay more to get
764 a top-10-percent programmer rather than a bottom-10-percent programmer, jump
765 at the chance. You'll get an immediate payoff in the quality and productivity of
766 the programmer you hire, and a residual effect in the quality and productivity of
767 the other programmers your organization is able to retain because good
768 programmers tend to cluster.

769 Religious Issues

770 Managers of programming projects aren't always aware that certain
771 programming issues are matters of religion. If you're a manager and you try to
772 require compliance with certain programming practices, you're inviting your
773 programmers' ire. Here's a list of religious issues:

- 774 • Programming language
- 775 • Indentation style
- 776 • Placing of braces
- 777 • Choice of IDE
- 778 • Commenting style
- 779 • Efficiency vs. readability trade-offs
- 780 • Choice of methodology—for example, scrum vs. extreme programming vs.
781 evolutionary delivery
- 782 • Programming utilities
- 783 • Naming conventions
- 784 • Use of *gotos*
- 785 • Use of global variables
- 786 • Measurements, especially productivity measures such as lines of code per
787 day

788 The common denominator among these topics is that a programmer's position on
789 each is a reflection of personal style. If you think you need to control a
790 programmer in any of these religious areas, consider these points:

791 ***be aware that you're dealing with a sensitive area***

792 Sound out the programmer on each emotional topic before jumping in with both
793 feet.

794 *Use “suggestions” or “guidelines” with respect to the area*

795 Avoid setting rigid “rules” or “standards.”

796 *Finesse the issues you can by sidestepping explicit mandates*

797 To finesse indentation style or brace placement, require source code to be run
798 through a pretty-printer formatter before it’s declared finished. Let the pretty
799 printer do the formatting. To finesse commenting style, require that all code be
800 reviewed and that unclear code be modified until it’s clear.

801 *Have your programmers develop their own standards*

802 As mentioned elsewhere, the details of a specific standard are often less
803 important than the fact that some standard exists. Don’t set standards for your
804 programmers, but do insist they standardize in the areas that are important to
805 you.

806 Which of the religious topics are important enough to warrant going to the mat?
807 Conformity in minor matters of style in any area probably won’t produce enough
808 benefit to offset the effects of lower morale. If you find indiscriminate use of
809 *gotos* or global variables, unreadable styles, or other practices that affect whole
810 projects, be prepared to put up with some friction in order to improve code
811 quality. If your programmers are conscientious, this is rarely a problem. The
812 biggest battles tend to be over nuances of coding style, and you can stay out of
813 those with no loss to the project.

814 **Physical Environment**

815 Here’s an experiment: Go out to the country. Find a farm. Find a farmer. Ask
816 how much money in equipment the farmer has for each worker. The farmer will
817 look at the barn and see a few tractors, some wagons, a combine for wheat, and a
818 peaviner for peas and will tell you that it’s over \$100,000 per employee.

819 Next go to the city. Find a programming shop. Find a programming manager.
820 Ask how much money in equipment the programming manager has for each
821 worker. The programming manager will look at an office and see a desk, a chair,
822 a few books, and a computer and will tell you that it’s under \$25,000 per
823 employee.

824 Physical environment makes a big difference in productivity. DeMarco and
825 Lister asked 166 programmers from 35 organizations about the quality of their
826 physical environments. Most employees rated their workplaces as not
827 acceptable. In a subsequent programming competition, the programmers who
828 performed in the top 25 percent had bigger, quieter, more private offices and
829 fewer interruptions from people and phone calls. Here’s a summary of the
830 differences in office space between the best and worst performers:

Environmental Factor	Top 25%	Bottom 25%
Dedicated floor space	78 sq. ft.	46 sq. ft.
Acceptably quiet workspace	57% yes	29% yes
Acceptably private workspace	62% yes	19% yes
Ability to silence phone	52% yes	10% yes
Ability to divert calls	76% yes	19% yes
Frequent needless interruptions	38% yes	76% yes
Workspace that makes programmer feel appreciated	57% yes	29% yes

Source: Peopleware (*DeMarco and Lister 1999*).

The data shows a strong correlation between productivity and the quality of the workplace. Programmers in the top 25 percent were 2.6 times more productive than programmers in the bottom 25 percent. DeMarco and Lister thought that the better programmers might naturally have better offices because they had been promoted, but further examination revealed that this wasn't the case. Programmers from the same organizations had similar facilities, regardless of differences in their performance.

Large software-intensive organizations have had similar experiences. Xerox, TRW, IBM, and Bell Labs have indicated that they realize significantly improved productivity with a \$10,000 to \$30,000 capital investment per person, sums that were more than recaptured in improved productivity (Boehm 1987a). With "productivity offices," self-reported estimates ranged from 39 to 47 percent improvement in productivity (Boehm et al. 1984).

In summary, bringing your workplace from a bottom-25-percent to a top-25-percent environment is likely to result in at least a 100 percent improvement in productivity.

Additional Resources on Programmers as Human Beings

Weinberg, Gerald M. *The Psychology of Computer Programming*, 2d Ed. New York: Van Nostrand Reinhold, 1998. This is the first book to explicitly identify programmers as human beings, and it's still the best on programming as a human activity. It's crammed with acute observations about the human nature of programmers and its implications.

DeMarco, Tom and Timothy Lister. *Peopleware: Productive Projects and Teams*, 2d Ed. New York: Dorset House, 1999. As the title suggests, this book also deals with the human factor in the programming equation. It's filled with

831

832 **HARD DATA**

833

834

835

836

837

838

839

840

841

842

843

844

845

846

847

CC2E.COM/2806

848

849

850

851

852

853

854

855

856

857

858 anecdotes about managing people, the office environment, hiring and developing
859 the right people, growing teams, and enjoying work. The authors lean on the
860 anecdotes to support some uncommon viewpoints, and the logic is thin in places,
861 but the people-centered spirit of the book is what's important, and the authors
862 deliver that message without faltering.

863 McCue, Gerald M. "IBM's Santa Teresa Laboratory—Architectural Design for
864 Program Development," *IBM Systems Journal* 17, no. 1 (1978): 4–25. McCue
865 describes the process that IBM used to create its Santa Teresa office complex.
866 IBM studied programmer needs, created architectural guidelines, and designed
867 the facility with programmers in mind. Programmers participated throughout.
868 The result is that in annual opinion surveys each year, the physical facilities at
869 the Santa Teresa facility are rated the highest in the company.

870 McConnell, Steve. *Professional Software Development*, Boston, MA: Addison
871 Wesley, 2004. Chapter 7, "Orphans Preferred," summarizes studies on
872 programmer demographics including personality types, educational
873 backgrounds, and job prospects.

874 Carnegie, Dale. *How to Win Friends and Influence People*, Revised Edition.
875 New York: Pocket Books, 1981. When Dale Carnegie wrote the title for the first
876 edition of this book in 1936, he couldn't have realized the connotation it would
877 carry today. It sounds like a book Machiavelli would have displayed on his shelf.
878 The spirit of the book is diametrically opposed to Machiavellian manipulation,
879 however, and one of Carnegie's key points is the importance of developing a
880 genuine interest in other people. Carnegie has a keen insight into everyday
881 relationships and explains how to work with other people by understanding them
882 better. The book is filled with memorable anecdotes, sometimes two or three to a
883 page. Anyone who works with people should read it at some point, and anyone
884 who manages people should read it now.

885 28.6 Managing Your Manager

886 In software development, nontechnical managers are common, as are managers
887 who have technical experience but who are 10 years behind the times.
888 Technically competent, technically current managers are rare. If you work for
889 one, do whatever you can to keep your job. It's an unusual treat.

890 ***In a hierarchy every***
891 ***employee tends to rise to***
892 ***his level of incompetence.***
893 — ***The Peter Principle***
894

If your manager is more typical, you're faced with the unenviable task of managing your manager. "Managing your manager" means that you need to tell your manager what to do rather than the other way around. The trick is to do it in a way that allows your manager to continue believing that you are the one being managed. Here are some approaches to dealing with your manager:

- 895 • Plant ideas for what you want to do, and then wait for your manager to have
896 a brainstorm (your idea) about doing what you want to do.
897 • Educate your manager about the right way to do things. This is an ongoing
898 job because managers are often promoted, transferred, or fired.
899 • Focus on your manager's interests, doing what he or she really wants you to
900 do, and don't distract your manager with unnecessary implementation
901 details. (Think of it as "encapsulation" of your job.)
902 • Refuse to do what your manager tells you, and insist on doing your job the
903 right way.
904 • Find another job.

905 The best long-term solution is to try to educate your manager. That's not always
906 an easy task, but one way you can prepare for it is by reading Dale Carnegie's
907 *How to Win Friends and Influence People*.

CC2E.COM/2813

908 **Additional Resources on Software Project 909 Management**

910 Here are a few books that cover issues of general concern in managing software
911 projects.

912 Gilb, Tom. *Principles of Software Engineering Management*. Wokingham,
913 England: Addison-Wesley, 1988. Gilb has charted his own course for thirty
914 years, and most of the time he's been ahead of the pack whether the pack realizes
915 it or not. This book is a good example. This was one of the first books to discuss
916 evolutionary development practices, risk management, and the use of formal
917 inspections. Gilb is keenly aware of leading-edge approaches; indeed this book
918 published more than 15 years ago contains most of the good practices currently
919 flying under the "Agile" banner. Gilb is incredibly pragmatic and the book is still
920 one of the best software management books.

921 McConnell, Steve. *Rapid Development*, Redmond, Wa.: Microsoft Press, 1996.
922 This book covers project leadership and project management issues from the
923 perspective of projects that are experiencing significant schedule pressure, which
924 in my experience is most projects.

925 Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software
926 Engineering, Anniversary Edition (2nd Ed)*, Reading, Mass.: Addison-Wesley,
927 1995. This book is a hodgepodge of metaphors and folklore related to managing
928 programming projects. It's entertaining, and it will give you many illuminating
929 insights into your own projects. It's based on Brooks's challenges in developing

930 the OS/360 operating system, which gives me some reservations. It's full of
931 advice along the lines of "We did this and it failed" and "We should have done
932 this because it would have worked." Brooks's observations about techniques that
933 failed are well grounded, but his claims that other techniques would have worked
934 are too speculative. Read the book critically to separate the observations from
935 the speculations. This warning doesn't diminish the book's basic value. It's still
936 cited in computing literature more often than any other book, and even though it
937 was originally published in 1975, it seems fresh today. It's hard to read it without
938 saying "Right on!" every couple of pages.

939 Relevant Standards

940 *IEEE Std 1058-1998, Standard for Software Project Management Plans.*

941 *IEEE Std 12207-1997, Information Technology—Software Life Cycle Processes.*

942 *IEEE Std 1045-1992, Standard for Software Productivity Metrics.*

943 *IEEE Std 1062-1998, Recommended Practice for Software Acquisition.*

944 *IEEE Std 1540-2001, Standard for Software Life Cycle Processes—Risk*
945 *Management.*

946 *IEEE Std 828-1998, Standard for Software Configuration Management Plans*

947 *IEEE Std 1490-1998, Guide—Adoption of PMI Standard—A Guide to the*
948 *Project Management Body of Knowledge.*

949 Key Points

- 950 • Good coding practices can be achieved either through enforced standards or
951 through more light-handed approaches.
- 952 • Configuration management, when properly applied, makes programmers'
953 jobs easier. This especially includes change control.
- 954 • Good software estimation is a significant challenge. Keys to success are
955 using multiple approaches, tightening down your estimates as you work your
956 way into the project, and making use of data to create the estimates.
- 957 • Measurement is a key to successful construction management. You can find
958 ways to measure any aspect of a project that are better than not measuring it
959 at all. Accurate measurement is a key to accurate scheduling, to quality
960 control, and to improving your development process.

- 961
- 962
- Programmers and managers are people, and they work best when treated as such.

29

Integration

3 CC2E.COM/2985

Contents

- 4 29.1 Importance of the Integration Approach
- 5 29.2 Integration Frequency—Phased or Incremental?
- 6 29.3 Incremental Integration Strategies
- 7 29.4 Daily Build and Smoke Test

Related Topics

Developer testing: Chapter 22

Debugging: Chapter 23

Managing construction: Chapter 28

THE TERM “INTEGRATION” REFERS TO the software-development activity in which you combine separate software components into a single system. On small projects, integration might consist of a morning spent hooking a handful of classes together. On large projects, it might consist of weeks or months of hooking sets of programs together. Regardless of the size of the task, common principles apply.

The topic of integration is intertwined with the topic of construction sequence. The order in which you build classes or components affects the order in which you can integrate them—you can’t integrate something that hasn’t been built yet. Both integration and construction sequence are important topics. This chapter addresses both topics from the integration point of view.

29.1 Importance of the Integration Approach

In engineering fields other than software, the importance of proper integration is well known. The Pacific Northwest, where I live, saw a dramatic illustration of the hazards of poor integration when the football stadium at the University of Washington collapsed partway through construction.

**F29xx01****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.

The structure wasn't strong enough to support itself as it was being built. It doesn't matter that it would have been strong enough by the time it was done; it needed to be strong enough at each step. If you construct and integrate software in the wrong order, it's harder to code, harder to test, and harder to debug. If none of it will work until all of it works, it can seem as though it will never be finished. It too can collapse under its own weight during construction—the bug count might seem insurmountable, progress might be invisible, or the complexity might be overwhelming—even though the finished product would have worked.

Because it's done after a developer has finished developer testing and in conjunction with system testing, integration is sometimes thought of as a testing activity. It's complex enough, however, that it should be viewed as an independent activity. Here are some of the benefits you can expect from careful integration:

-
- KEY POINT**
- Easier defect diagnosis
 - Fewer defects
 - Less scaffolding

28
29
30
31
32
33
34

35
36
37
38
39
40
41
42

43
44
45
46
47

48
49
50

- 51 • Shorter time to first working product
- 52 • Shorter overall development schedules
- 53 • Better customer relations
- 54 • Improved morale
- 55 • Improved chance of project completion
- 56 • More reliable schedule estimates
- 57 • More accurate status reporting
- 58 • Improved code quality
- 59 • Less documentation

60 These might seem like elevated claims for system testing's forgotten cousin, but
61 the fact that it's overlooked in spite of its importance is precisely the reason
62 integration has its own chapter in this book.

63 **29.2 Integration Frequency—Phased or 64 Incremental?**

65 Programs are integrated by means of either the phased or the incremental
66 approach.

67 **Phased Integration**

68 Until a few years ago, phased integration was the norm. It follows these well-
69 defined steps:

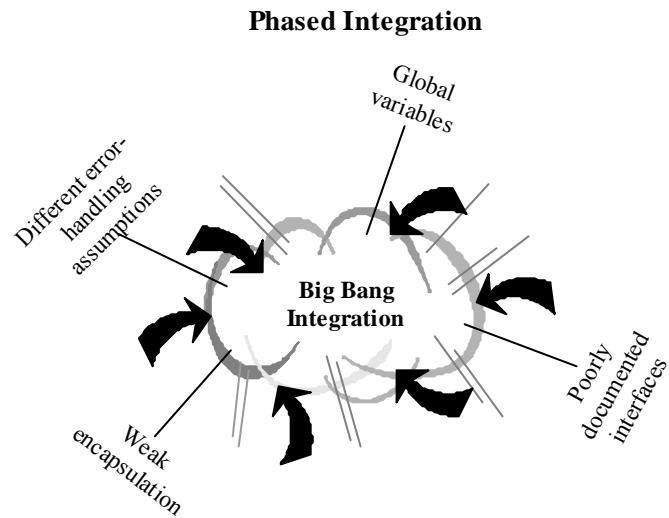
- 70 1. Design, code, test, and debug each class. This step is called "unit
71 development."
- 72 2. Combine the classes into one whopping-big system. This is called "system
73 integration."
- 74 3. Test and debug the whole system. This is called "system dis-integration."
75 (Thanks to Meilir Page-Jones for this witty observation.)

76 **CROSS-REFERENCE** Many integration problems arise
77 from using global data. For
78 techniques on working with
79 global data safely, see
80 Section 13.3, "Global Data."
81

One problem with phased integration is that when the classes in a system are put together for the first time, new problems inevitably surface and the causes of the problems could be anywhere. Since you have a large number of classes that have never worked together before, the culprit might be a poorly tested classes, an error in the interface between two classes, or an error caused by an interaction between two classes. All classes are suspect.

82
83
84
85
86
87

The uncertainty about the location of any of the specific problems is compounded by the fact that all the problems suddenly present themselves at once. This forces you to deal not only with problems caused by interactions between classes but with problems that are hard to diagnose because the problems themselves interact. For this reason, another name for phased integration is “big bang integration.”



88
89
90
91

F29xx02

Figure 29-2

Phased integration is also called “big bang” integration for a good reason!

92
93
94
95

Phased integration can't begin until late in the project, after all the classes have been developer-tested. When the classes are finally combined and errors surface by the score, programmers immediately go into panicky debugging mode rather than methodical error detection and correction.

96
97
98
99

For small programs—no, for tiny programs—phased integration might be the best approach. If the program has only two or three classes, phased integration might save you time, if you're lucky. But in most cases, another approach is better.

100

101 **CROSS-REFERENCE** Met
102 aphors appropriate for
103 incremental integration are
discussed in “Software
104 Oyster Farming: System
Accretion” and “Software
105 Construction: Building
106 Software” in Section 2.3.
107

108

109
110
111
112

113
114
115
116
117
118

119
120
121
122
123

124
125
126

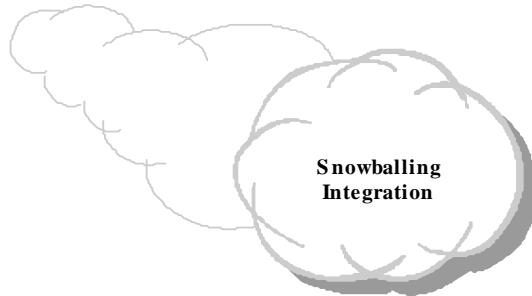
Incremental Integration

In incremental integration, you write and test a program in small pieces and then combine the pieces one at a time. In this one-piece-at-a-time approach to integration, you follow these steps:

1. Develop a small, functional part of the system. It can be the smallest functional part, the hardest part, a key part, or some combination. Thoroughly test and debug it. It will serve as a skeleton on which to hang the muscles, nerves, and skin that make up the remaining parts of the system.
2. Design, code, test, and debug a class.
3. Integrate the new class with the skeleton. Test and debug the combination of skeleton and new class. Make sure the combination works before you add any new classes. If work remains to be done, repeat the process starting at step 2.

Occasionally, you might want to integrate units larger than a single class. If a component has been thoroughly tested, for example, and each of its classes put through a mini-integration, you can integrate the whole component and still be doing incremental integration. The system grows and gains momentum as you add pieces to it in the same way that a snowball grows and gains momentum when it rolls down a hill.

Incremental Integration



F29xx03

Figure 29-3

Incremental integration helps a project build momentum, like a snowball going down a hill.

Benefits of Incremental Integration

The incremental approach offers many advantages over the traditional phased approach regardless of which incremental strategy you use.

127

HARD DATA

129

130

131

132

133

134

135

136

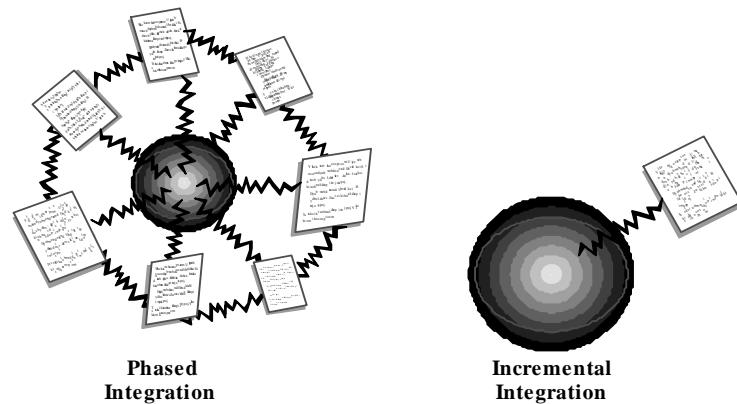
137

138

139

Errors are easy to locate

When new problems surface during incremental integration, the new class is obviously at fault. Either its interface to the rest of the program contains an error or its interaction with a previously integrated class produces an error. Either way, you know exactly where to look. Moreover, simply because you have fewer problems at once, you reduce the risk that multiple problems will interact or that one problem will mask another. The more interface errors you tend to have, the more this benefit of incremental integration will help your projects. An accounting of errors for one project revealed that 39 percent were intermodule interface errors (Basil and Perricone 1984). Since developers on many projects spend up to 50 percent of their time debugging, maximizing debugging effectiveness by making errors easy to locate provides benefits in quality and productivity.



140

141

142

143

144

145

146

F29xx04**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.

147

148

149

150

151

The system succeeds early in the project

When code is integrated and running, even if the system isn't usable, it's apparent that it soon will be. With incremental integration, programmers see early results from their work, so their morale is better than when they suspect that their project will never draw its first breath.

152

153

154

155

156

You get improved progress monitoring

When you integrate frequently, the features that are present and not present are obvious. Management will have a better sense of progress from seeing 50 percent of a system's capability working than from hearing that coding is "99 percent complete."

157 ***You'll improve customer relations***

158 If frequent integration has an effect on developer morale, it also has an effect on
159 customer morale. Customers like signs of progress, and incremental builds
160 provide signs of progress frequently.

161 ***The units of the system are tested more fully***

162 Integration starts early in the project. You integrate each class as it's developed,
163 rather than waiting for one magnificent binge of integration at the end. Classes
164 are developer tested in both cases, but each class is exercised as a part of the
165 overall system more often with incremental integration than it is with phased
166 integration.

167 ***You can build the system with a shorter development schedule***

168 If integration is planned carefully, you can design part of the system while
169 another part is being coded. This doesn't reduce the total number of work-hours
170 required to develop the complete design and code, but it allows some work to be
171 done in parallel, an advantage when calendar time is at a premium.

172 Incremental integration supports and encourages other incremental strategies.
173 The advantages of incrementalism applied to integration are the tip of the
174 iceberg.

175

29.3 Incremental Integration Strategies

176 With phased integration, you don't have to plan the order in which project
177 components are built. All components are integrated at the same time, so you can
178 build them in any order as long as they're all ready by D-day.

179 With incremental integration, you have to plan more carefully. Most systems
180 will call for the integration of some components before the integration of others.
181 Planning for integration thus affects planning for construction; the order in
182 which components are constructed has to support the order in which they will be
183 integrated.

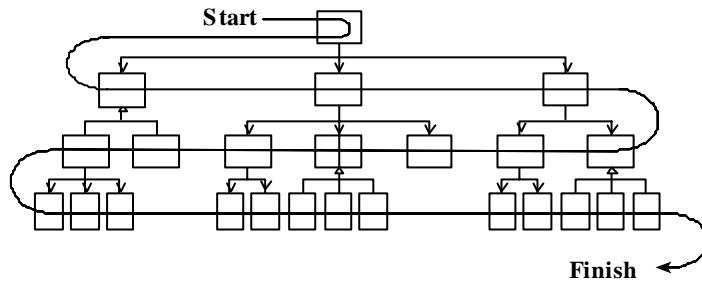
184 Integration-order strategies come in a variety of shapes and sizes, and none is
185 best in every case. The best integration approach varies from project to project,
186 and the best solution is always the one that you create to meet the specific
187 demands of a specific project. Knowing the points on the methodological
188 number line will give you insight into the possible solutions.

189

Top-Down Integration

190 In top-down integration, the class at the top of the hierarchy is written and
191 integrated first. The top is the main window, the applications control loop, the

192 object that contains *main()* in Java, *WinMain()* for Microsoft Windows
193 programming, or similar. Stubs have to be written to exercise the top class. Then,
194 as classes are integrated from the top down, stub classes are replaced with real
195 ones. Here's how this kind of integration proceeds:



F29xx05

Figure 29-5

In top-down integration, you add classes at the top first, at the bottom last.

An important aspect of top-down integration is that the interfaces between classes must be carefully specified. The most troublesome errors to debug are not the ones that affect single classes but those that arise from subtle interactions between classes. Careful interface specification can reduce the problem. Interface specification isn't an integration activity, but making sure that the interfaces have been specified well is.

In addition to the advantages you get from any kind of incremental integration, an advantage of top-down integration is that the control logic of the system is tested relatively early. All the classes at the top of the hierarchy are exercised a lot, so that big, conceptual, design problems are exposed quickly.

Another advantage of top-down integration is that, if you plan it carefully, you can complete a partially working system early in the project. If the user-interface parts are at the top, you can get a basic interface working quickly and flesh out the details later. The morale of both users and programmers benefits from getting something visible working early.

Top-down incremental integration also allows you to begin coding before the low-level design details are complete. Once the design has been driven down to a fairly low level of detail in all areas, you can begin implementing and integrating the classes at the higher levels without waiting to dot every "i" and cross every "t."

In spite of these advantages, pure top-down integration usually involves disadvantages that are more troublesome than you'll want to put up with.

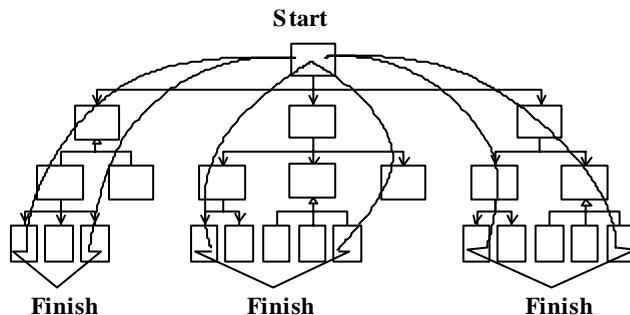
222 Pure top-down integration leaves exercising the tricky system interfaces until
223 last. If system interfaces are buggy or a performance problem, you'd usually like
224 to get to them long before the end of the project. It's not unusual for a low-level
225 problem to bubble its way to the top of the system, causing high-level changes
226 and reducing the benefit of earlier integration work. Minimize the bubbling
227 problem through early careful developer testing and performance analysis of the
228 classes that exercise system interfaces.

229 Another problem with pure top-down integration is that it takes a dump truck full
230 of stubs to integrate from the top down. Many low-level classes haven't been
231 integrated, which implies that a large number of stubs will be needed during
232 intermediate steps in integration. Stubs are problematic in that, as test code, they
233 are more likely to contain errors than the more carefully designed production
234 code. Errors in the new stubs that support a new class defeat the purpose of
235 incremental integration, which is to restrict the source of errors to one new class.

236 **CROSS-REFERENCE** Top-
237 down integration is related to
238 top-down design in name
239 only. For details on top-down
240 design, see "Top-Down and
241 Bottom-Up Design
242 Approaches" in Section 5.4.
243

244 Finally, you can't use top-down integration if the collection of classes doesn't
245 have a top. In many interactive systems, the location of the "top" is subjective. In
246 many systems, the user interface is the top. In other systems, *main()* is the top.

247 A good alternative to pure top-down integration is the vertical-slice approach
248 shown in Figure 29-6. In this approach, the system is implemented top-down in
249 sections, perhaps fleshing out areas of functionality one by one, and then moving
250 to the next area.

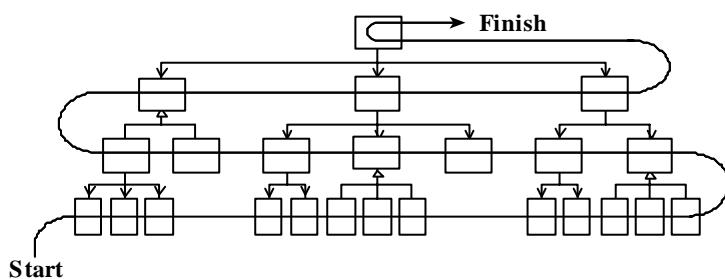
**F29xx06****Figure 29-6**

As an alternative to proceeding strictly top to bottom, you can integrate from the top down in vertical slices.

Even though pure top-down integration isn't workable, thinking about it will help you decide on a general approach. Some of the benefits and hazards that apply to a pure top-down approach apply, less obviously, to looser top-down approaches like vertical-slice integration, so keep them in mind.

Bottom-Up Integration

In bottom-up integration, you write and integrate the classes at the bottom of the hierarchy first. Adding the low-level classes one at a time rather than all at once is what makes bottom-up integration an incremental integration strategy. You write test drivers to exercise the low-level classes initially and add classes to the test-driver scaffolding as they're developed. As you add higher-level classes, you replace driver classes with real ones. Here's the order in which classes are integrated in the bottom-up approach:

**F29xx07****Figure 29-7**

In bottom-up integration, you integrate classes at the bottom first, at the top last.

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

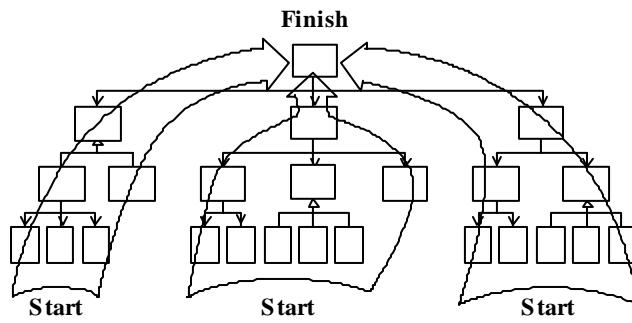
272 Bottom-up integration provides a limited set of incremental-integration
273 advantages. It restricts the possible sources of error to the single class being
274 integrated, so errors are easy to locate. Integration can start early in the project.

275 Bottom-up integration also exercises potentially troublesome system interfaces
276 early. Since system limitations often determine whether you can meet the
277 system's goals, making sure the system has done a full set of calisthenics is
278 worth the trouble.

279 What are the problems with bottom-up integration? The main problem is that it
280 leaves integration of the major, high-level system interfaces until last. If the
281 system has conceptual design problems at the higher levels, construction won't
282 find them until all the detailed work has been done. If the design must be
283 changed significantly, some of the low-level work might have to be discarded.

284 Bottom-up integration requires you to complete the design of the whole system
285 before you start integration. If you don't, assumptions that needn't have
286 controlled the design might end up deeply embedded in low-level code, giving
287 rise to the awkward situation in which you design high-level classes to work
288 around problems in low-level ones. Letting low-level details drive the design of
289 higher-level classes contradicts principles of information hiding and object-
290 oriented design. The problems of integrating higher-level classes are but a
291 teardrop in a rainstorm compared to the problems you'll have if you don't
292 complete the design of high-level classes before you begin low-level coding.

293 As with top-down integration, pure bottom-up integration is rare, and you can
294 use a hybrid approach instead, including integrating in sections.



295

296

F29xx08

297

Figure 29-8

298

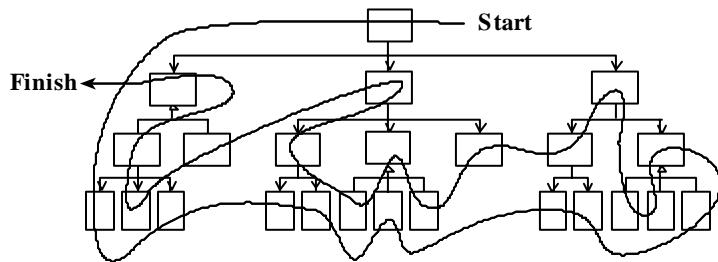
299 *As an alternative to proceeding purely bottom to top, you can integrate from the
300 bottom up in sections. This blurs the line between bottom-up integration and feature-
 oriented integration, which is described later in this chapter.*

301 Sandwich Integration

302 The problems with pure top-down and pure bottom-up integration have led some
303 experts to recommend a sandwich approach (Myers 1976). You first integrate the
304 high-level business-object classes at the top of the hierarchy. Then you integrate
305 the device-interface classes and widely used utility classes at the bottom. These
306 high-level and low-level classes are the bread of the sandwich.

307 You leave the middle-level classes until later. These make up the meat, cheese,
308 and tomatoes of the sandwich. If you're a vegetarian, they might make up the
309 tofu and bean sprouts of the sandwich, but the author of sandwich integration is
310 silent on this point—maybe his mouth was full.

311 Here's an illustration of the sandwich approach:



312 F29xx09

313 **Figure 29-9**

314 *In sandwich integration, you integrate top-level and widely used bottom-level classes
315 first, and save middle-level classes for last.*

316 This approach avoids the rigidity of pure bottom-up or top-down integration. It
317 integrates the often-troublesome classes first and has the potential to minimize
318 the amount of scaffolding you'll need. It's a realistic, practical approach.
319

320 The next approach is similar and more sophisticated.

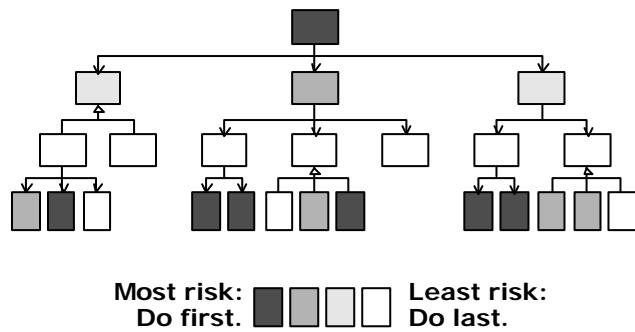
321 Risk-Oriented Integration

322 Risk-oriented integration is also called "hard part first" integration. It's like
323 sandwich integration in that it seeks to avoid the problems inherent in pure top-
324 down or pure bottom-up integration. Coincidentally, it also tends to integrate the
325 classes at the top and the bottom first, saving the middle-level classes for last.
326 The motivation, however, is different.

327 In risk-oriented integration, you identify the level of risk associated with each
328 class. You decide which will be the most challenging parts to implement, and

329 you implement them first. Experience indicates that top-level interfaces are
 330 risky, so they are often at the top of the risk list. System interfaces, usually at the
 331 bottom level of the hierarchy, are also risky, so they're also at the top of the risk
 332 list. In addition, you might know of classes in the middle that will be
 333 challenging. Perhaps a class implements a poorly understood algorithm or has
 334 ambitious performance goals. Such classes can also be identified as high risks
 335 and integrated relatively early.

336 The remainder of the code, the easy stuff, can wait until later. Some of it will
 337 probably turn out to be harder than you thought, but that's unavoidable. Here's
 338 an illustration of risk-oriented integration:



F29xx10

Figure 29-10

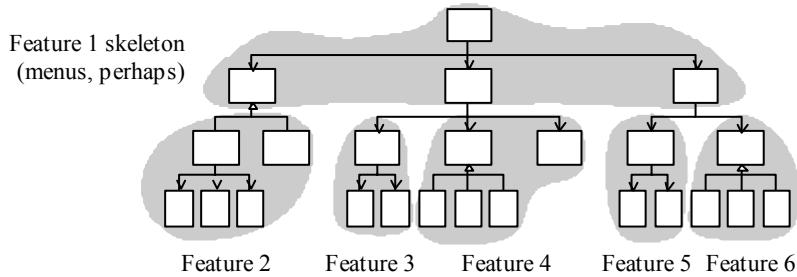
In risk-oriented integration, you integrate classes that you expect to be most troublesome first; you implement easier classes later.

Feature-Oriented Integration

345 Another approach is to integrate one feature at a time. The term "feature" doesn't
 346 refer to anything fancy—just an identifiable function of the system you're
 347 integrating. If you're writing a word processor, a feature might be displaying
 348 underlining on the screen or reformatting the document automatically—
 349 something like that.

350 When the feature to be integrated is bigger than a single class, the "increment" in
 351 incremental integration is bigger than a single class. This diminishes the benefit
 352 of incrementalism a little in that it reduces your certainty about the source of new
 353 errors, but if you have thoroughly tested the classes that implement the new
 354 feature before you integrate them, that's only a small disadvantage. You can use
 355 the incremental integration strategies recursively by integrating small pieces to
 356 form features and then incrementally integrating features to form a system.

357 You'll usually want to start with a skeleton you've chosen for its ability to
358 support the other features. In an interactive system, the first feature might be the
359 interactive menu system. You can hang the rest of the features on the feature that
360 you integrate first. Here's how it looks graphically:



361 F29xx11

362 Figure 29-11

363 *In feature-oriented integration, you integrate classes in groups that make up*
364 *identifiable features—usually, but not always, multiple classes at a time.*

366 Components are added in “feature trees,” hierarchical collections of classes that
367 make up a feature. Integration is easier if each feature is relatively independent,
368 perhaps calling the same low-level library code as the classes for other features,
369 but having no calls to middle-level code in common with other features. (The
370 shared, low-level library classes aren’t shown in the illustration above.)

371 Feature-oriented integration offers three main advantages. First, it eliminates
372 scaffolding for virtually everything except low-level library classes. The skeleton
373 might need a little scaffolding, or some parts of the skeleton might simply not be
374 operational until particular features have been added. When each feature has
375 been hung on the structure, however, no additional scaffolding is needed. Since
376 each feature is self-contained, each feature contains all the support code it needs.

377 The second main advantage is that each newly integrated feature brings about an
378 incremental addition in functionality. This provides evidence that the project is
379 moving steadily forward.

380 A third advantage is that feature-oriented integration works well with object-
381 oriented design. Objects tend to map well to features, which makes feature-
382 oriented integration a natural choice for object-oriented systems.

383 Pure feature-oriented integration is as difficult to pursue as pure top-down or
384 bottom-up integration. Usually some of the low-level code must be integrated
385 before certain significant features can be.

386

387

388

389

390

391

392

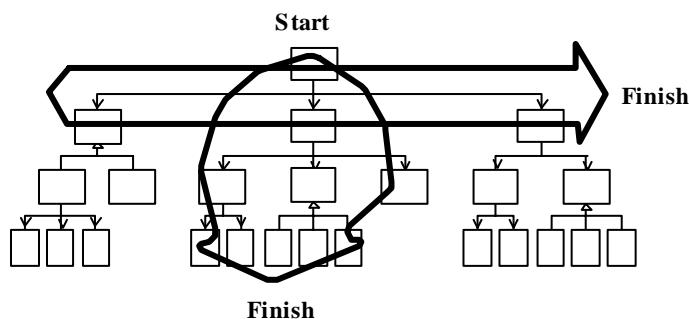
393

394

395

T-Shaped Integration

A final approach that often addresses the problems associated with top-down and bottom-up integration is called “T-Shaped Integration.” In this approach, one specific vertical slice is selected for early development and integration. That slice should exercise the system end-to-end, and should be capable of flushing out any major problems in the system’s design assumptions. Once that vertical slice has been implemented (and any associated problems have been corrected), then the overall breadth of the system can be developed—such as the menu system in a desktop application. This approach is often combined with risk-oriented or feature-oriented integration.



396

397

398

399

400

401

F29xx12

Figure 29-12

In T-Shaped integration, you build and integrate a deep slice of the system to verify architectural assumptions, then you build and integrate the breadth of the system to provide a framework for developing the remaining functionality.

402

Summary of Integration Approaches

403

404

405

Bottom-up, top-down, sandwich, risk-oriented, feature-oriented, T-shape—do you get the feeling that people are making these names up as they go along? They are.

406

407

408

409

410

None of these approaches are robust procedures that you should follow methodically from step 1 to step 47 and then declare yourself to be done. Like software-design approaches, they are heuristics more than algorithms, and rather than following any procedure dogmatically, you come out ahead by making up a unique strategy tailored to your specific project.

411

412 **FURTHER READING** Much
413 of this discussion is adapted
414 from Chapter 18 of *Rapid
415 Development* (McConnell
1996). If you've read that
416 discussion, you might skip
ahead to the "Continuous
417 Integration" section.

418

419

420

421

422

423

424

425

426

427

428

429

430

431

432

433

434

435

436

437

438

439

440

441

442

443

444

445

446

29.4 Daily Build and Smoke Test

Whatever integration strategy you select, a good approach to integrating the software is the "daily build and smoke test." Every file is compiled, linked, and combined into an executable program every day, and the program is then put through a "smoke test," a relatively simple check to see whether the product "smokes" when it runs.

This simple process produces several significant benefits.

It reduces the risk of low quality. Related to the risk of unsuccessful or problematic integration is the risk of low quality. By minimally smoke-testing all the code daily, quality problems are prevented from taking control of the project. You bring the system to a known, good state, and then you keep it there. You simply don't allow it to deteriorate to the point where time-consuming quality problems can occur.

It supports easier defect diagnosis. When the product is built and tested every day, it's easy to pinpoint why the product is broken on any given day. If the product worked on Day 17 and is broken on Day 18, something that happened between the two builds broke the product.

It improves morale. Seeing a product work provides an incredible boost to morale. It almost doesn't matter what the product does. Developers can be excited just to see it display a rectangle! With daily builds, a bit more of the product works every day, and that keeps morale high.

One side effect of frequent integration is that it surfaces work that can otherwise accumulate unseen until it appears unexpectedly at the end of the project. That accumulation of unsurfaced work can turn into an end-of-project tar pit that takes weeks or months to wrestle out of. Teams that haven't used the daily build process previously sometimes feel that daily builds slow their progress to a snail's crawl. What's really happening is that daily builds amortize work more steadily throughout the project, and the project team is just getting a more accurate picture of how fast it's been working all along.

Here are some of the ins and outs of using daily builds.

Build daily

The most fundamental part of the daily build is the "daily" part. As Jim McCarthy says, treat the daily build as the heartbeat of the project (McCarthy 1995). If there's no heartbeat, the project is dead. A little less metaphorically, Michael Cusumano and Richard W. Selby describe the daily build as the sync pulse of a project (Cusumano and Selby 1995). Different developers' code is

447 allowed to get a little out of sync between these pulses, but every time there's a
448 sync pulse, the code has to come back into alignment. When you insist on
449 keeping the pulses close together, you prevent developers from getting out of
450 sync entirely.

451 Some organizations build every week, rather than every day. The problem with
452 this is that if the build is broken one week, you might go for several weeks
453 before the next good build. When that happens, you lose virtually all of the
454 benefit of frequent builds.

Check for broken builds

455 For the daily-build process to work, the software that's built has to work. If the
456 software isn't usable, the build is considered to be broken and fixing it becomes
457 top priority.

459 Each project sets its own standard for what constitutes "breaking the build." The
460 standard needs to set a quality level that's strict enough to keep showstopper
461 defects out but lenient enough to dis-regard trivial defects, an undue attention to
462 which could paralyze progress.

463 At a minimum, a "good" build should

- 464 • compile all files, libraries, and other components successfully
- 465 • link all files, libraries, and other components successfully
- 466 • not contain any showstopper bugs that prevent the program from being
467 launched or that make it hazardous to operate
- 468 • pass the smoke test

Smoke test daily

469 The smoke test should exercise the entire system from end to end. It does not
470 have to be exhaustive, but it should be capable of exposing major problems. The
471 smoke test should be thorough enough that if the build passes, you can assume
472 that it is stable enough to be tested more thoroughly.

474 The daily build has little value without the smoke test. The smoke test is the
475 sentry that guards against deteriorating product quality and creeping integration
476 problems. Without it, the daily build becomes just a time-wasting exercise in
477 ensuring that you have a clean compile every day.

478 The smoke test must evolve as the system evolves. At first, the smoke test will
479 probably test something simple, such as whether the system can say, "Hello,
480 World." As the system develops, the smoke test will become more thorough. The

481 first test might take a matter of seconds to run; as the system grows, the smoke
482 test can grow to 30 minutes, an hour, or more.

483 ***Automate the daily build and smoke test***

484 Care and feeding of the build can become time consuming. Automating the build
485 and smoke test helps ensure that the code gets built and the smoke test gets run.

486 ***Establish a build group***

487 On most projects, tending the daily build and keeping the smoke test up to date
488 becomes a big enough task to be an explicit part of someone's job. On large
489 projects, it can become a full-time job for more than one person. On the first
490 release of Windows NT, for example, there were four full-time people in the
491 build group (Zachary 1994).

492 ***Add revisions to the build only when it makes sense to do so***

493 Individual developers usually don't write code quickly enough to add
494 meaningful increments to the system on a daily basis. They should work on a
495 chunk of code and then integrate it when they have a collection of code in a
496 consistent state-usually once every few days.

497 ***... but don't wait too long to add a set of revisions***

498 Beware of checking in code infrequently. It's possible for a developer to become
499 so embroiled in a set of revisions that every file in the system seems to be
500 involved. That undermines the value of the daily build. The rest of the team will
501 continue to realize the benefit of incremental integration, but that particular
502 developer will not. If a developer goes more than a couple of days without
503 checking in a set of changes, consider that developer's work to be at risk. As
504 Kent Beck points out, frequent integration sometimes forces you to break the
505 construction of a single feature into multiple episodes. That overhead is an
506 acceptable price to pay for the reduced integration risk, improved status
507 visibility, improved testability, and other benefits of frequent integration (Beck
508 2000).

509 ***Require developers to smoke test their code before adding it to the system***

510 Developers need to test their own code before they add it to the build. A
511 developer can do this by creating a private build of the system on a personal
512 machine, which the developer then tests individually. Or the developer can
513 release a private build to a "testing buddy," a tester who focuses on that
514 developer's code. The goal in either case is to be sure that the new code passes
515 the smoke test before it's allowed to influence other parts of the system.

516 ***Create a holding area for code that's to be added to the build***

517 Part of the success of the daily-build process depends on knowing which builds
518 are good and which are not. In testing their own code, developers need to be able
519 to rely on a known good system.

520 Most groups solve this problem by creating a holding area for code that
521 developers think is ready to be added to the build. New code goes into the
522 holding area, the new build is built, and if the build is acceptable, the new code
523 is migrated into the master sources.

524 On small and medium-sized projects, a version-control system can serve this
525 function. Developers check new code into the version-control system.
526 Developers who want to use a known good build simply set a date flag in their
527 version-control options file that tells the system to retrieve files based on the date
528 of the last-known good build.

529 On large projects or projects that use unsophisticated version-control software,
530 the holding-area function has to be handled manually. The author of a set of new
531 code sends email to the build group to tell them where to find the new files to be
532 checked in. Or the group establishes a “check-in” area on a file server where
533 developers put new versions of their source files. The build group then assumes
534 responsibility for checking new code into version control after they have verified
535 that the new code doesn’t break the build.

536 ***Create a penalty for breaking the build***

537 Most groups that use daily builds create a penalty for breaking the build. Make it
538 clear from the beginning that keeping the build healthy is the project’s top
539 priority. A broken build should be the exception, not the rule. Insist that
540 developers who have broken the build stop all other work until they’ve fixed it.
541 If the build is broken too often, it’s hard to take seriously the job of not breaking
542 the build.

543 A light-hearted penalty can help to emphasize this priority. Some groups give
544 out lollipops to each “sucker” who breaks the build. This developer then has to
545 tape the sucker to his office door until he fixes the problem. Other groups have
546 guilty developers wear goat horns or contribute \$5 to a morale fund.

547 Some projects establish a penalty with more bite. Microsoft developers on high-
548 profile projects such as Windows 2000 and Microsoft Office have taken to
549 wearing beepers in the late stages of their projects. If they break the build, they
550 get called in to fix it even if their defect is discovered at 3 a.m.

551 ***Release builds in the morning***

552 Some groups have found that they prefer to build overnight, smoke test in the
553 early morning, and release new builds in the morning rather than the afternoon.
554 There are several advantages to smoke testing and releasing builds in the
555 morning.

556
557
558
559
560
561

First, if you release a build in the morning, testers can test with a fresh build that day. If you generally release builds in the afternoon, testers feel compelled to launch their automated tests before they leave for the day. When the build is delayed, which it often is, the testers have to stay late to launch their tests. Because it's not their fault that they have to stay late, the build process becomes demoralizing.

562
563
564
565

When you complete the build in the morning, you have more reliable access to developers when there are problems with the build. During the day, developers are down the hall. During the evening, developers can be anywhere. Even when developers are given beepers, they're not always easy to locate.

566
567
568

It might be more macho to start smoke testing at the end of the day and call people in the middle of the night when you find problems, but it's harder on the team, it wastes time, and in the end you lose more than you gain.

569
570
571
572
573
574
575
576

Build and smoke test even under pressure

When schedule pressure becomes intense, the work required to maintain the daily build can seem like extravagant overhead. The opposite is true. Under stress, developers lose some of their discipline. They feel pressure to take construction shortcuts that they would not take under less stressful circumstances. They review and test their own code less carefully than usual. The code tends toward a state of entropy more quickly than it does during less stressful times.

577
578
579

Against this backdrop, daily builds enforce discipline and keep pressure-cooker projects on track. The code still tends toward a state of entropy, but the build process brings that tendency to heel every day.

580
581

What Kinds of Projects Can Use the Daily Build Process?

582
583
584
585
586
587
588
589
590

Some developers protest that it is impractical to build every day because their projects are too large. But what was perhaps the most complex software project in recent history used daily builds successfully. By the time it was released, Microsoft Windows 2000 consisted of about 50 million lines of code spread across about tens of thousands of source files. A complete build took as many as 19 hours on several machines, but the NT development team still managed to build every day (Zachary 1994). Far from being a nuisance, the NT team attributed much of its success on that huge project to their daily builds. The larger the project, the more important incremental integration becomes.

591 **HARD DATA**
592

A review of 104 projects in the U.S., India, Japan, and Europe found that only 20-25 percent of projects used daily builds at either the beginning or middle of

593
594

their projects (Cusumano et al 2003), so this represents a significant opportunity for improvement.

595

596 **HARD DATA**

597
598
599
600
601
602
603
604
605
606

Continuous Integration

Some software writers have taken daily builds as a jumping-off point and recommend integrating *continuously*—literally integrating each change with the latest build every couple of hours (Beck 2000). I think integrating continuously is too much of a good thing. In my free time, I operate a discussion group consisting of the top technical executives from companies like Amazon.com, Boeing, Expedia, Microsoft, Nordstrom, and other Seattle-area companies. In a poll of these top technical executives, *none* of them thought that continuous integration was superior to daily integration. On medium and large projects, there is value in letting the code get out of synch for short periods. Daily builds allow the project team to rendezvous frequently enough. As long as the team synchs up every day, they don't need to rendezvous every hour or continuously.

607 CC2E.COM/2992

608

609
610
611
612
613
614
615
616
617
618

619

620
621
622
623
624
625
626
627

CHECKLIST: Integration

Integration Strategy

- Does the strategy identify the optimal order in which subsystems, classes, and routines should be integrated?
- Is the integration order coordinated with the construction order so that classes will be ready for integration at the right time?
- Does the strategy lead to easy diagnosis of defects?
- Does the strategy keep scaffolding to a minimum?
- Is the strategy better than other approaches?
- Have the interfaces between components been specified well? (Specifying interfaces isn't an integration task, but verifying that they have been specified well is.)

Daily Build and Smoke Test

- Is the project building frequently—ideally, daily—to support incremental integration?
- Is a smoke test run with each build so that you know whether the build works?
- Have you automated the build and the smoke test?
- Do developers check in their code frequently—going no more than a day or two between check-ins?
- Is a broken build a rare occurrence?

628

629

CC2E.COM/2999

630

Additional Resources

631

Integration

632

633

634

635

636

637

638

Lakos, John. *Large-Scale C++ Software Design*, Boston, Mass.: Addison Wesley, 1996. Lakos argues that a system's "physical design"—its hierarchy of files, directories, and libraries—significantly affects a development team's ability to build software. If you don't pay attention to the physical design, build times will become long enough to undermine frequent integration. Lakos's discussion focuses on C++, but the insights related to "physical design" apply just as much to projects in other languages.

639

640

Myers, Glenford J. *The Art of Software Testing*. New York: John Wiley, 1979. This classic testing book discusses integration as a testing activity.

641

Incrementalism

642

643

644

645

646

647

McConnell, Steve. *Rapid Development*. Redmond, WA: Microsoft Press, 1996. Chapter 7 on "Lifecycle Planning" goes into much detail about the tradeoffs involved with more flexible and less flexible lifecycle models. Chapters 20, 21, 35, and 36 discuss specific lifecycle models that support various degrees of incrementalism. Chapter 19 describes "designing for change," a key activity needed to support iterative and incremental development models.

648

649

650

651

652

653

654

655

Boehm, Barry W. "A Spiral Model of Software Development and Enhancement." *Computer*, May 1988: 61-72. In this paper, Boehm describes his "spiral model" of software development. He presents the model as an approach to managing risk in a software-development project, so the paper is about development generally rather than about integration specifically. Boehm is one of the world's foremost expert on the big-picture issues of software development, and the clarity of his explanations reflects the quality of his understanding.

656

657

658

659

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley, 1988. Chapters 7 and 15 contain thorough discussions of evolutionary delivery, one of the first incremental development approaches.

660

661

662

Beck, Kent. *Extreme Programming: Embrace Change*, Reading, Mass.: Addison Wesley, 2000. This book contains a more modern, more concise, and more evangelical presentation of many of the ideas in Gilb's book. I personally prefer

663 the depth of analysis presented in Gilb's book, but some readers may find Beck's
664 presentation more accessible or more directly applicable to the kind of project
665 they're working on.

666

Key Points

- 667
- The construction sequence and integration approach affect the order in
668 which classes are designed, coded, and tested.
 - A well-thought-out integration order reduces testing effort and eases
669 debugging.
 - Daily builds can reduce integration problems, improve developer morale,
670 and provide useful project management information.
 - Incremental integration comes in several varieties, and, unless the project is
671 trivial, any one of them is better than phased integration.
 - The best integration approach for any specific project is usually a
672 combination of top-down, bottom-up, risk-oriented, and other integration
673 approaches. T-shaped integration and vertical-slice integration are two
674 approaches that often work well.
- 675
- 676
- 677
- 678

30

Programming Tools

Contents

- 30.1 Design Tools
- 30.2 Source-Code Tools
- 30.3 Executable-Code Tools
- 30.4 Tool-Oriented Environments
- 30.5 Building Your Own Programming Tools
- 30.6 Tool Fantasyland

Related Topics

Version-control tools: in Section 28.2

Debugging tools: Section 23.5

Test-support tools: Section 22.5

MODERN PROGRAMMING TOOLS DECREASE THE amount of time required for construction. Use of a leading-edge tool set—and familiarity with the tools used—can increase productivity by 50 percent or more (Jones 2000; Boehm, et al 2000). Programming tools can also reduce the amount of tedious detail work that programming requires.

A dog might be man's best friend, but a few good tools are a programmer's best friends. As Barry Boehm discovered long ago, 20 percent of the tools tend to account for 80 percent of the tool usage (1987b). If you're missing one of the more helpful tools, you're missing something that you could use a lot.

This chapter is focused in two ways. First, it covers only construction tools. Requirements-specification, management, and end-to-end-development tools are outside the scope of the book. Refer to the “Additional Resources” section at the end of the chapter for more information on tools for those aspects of software development. Second, this chapter covers kinds of tools rather than specific brands. A few tools are so common that they're discussed by name, but specific versions, products, and companies change so quickly that information about most of them would be out of date before the ink on these pages was dry.

HARD DATA

31 A programmer can work for many years without discovering some of the most
32 valuable tools available. The mission of this chapter is to survey available tools
33 and help you determine whether you've overlooked any tools that might be
34 useful. If you're a tool expert, you won't find much new information in this
35 chapter. You might skim the earlier parts of the chapter, read Section 30.6 on
36 Tool Fantasyland, and then move on to the next chapter.

37 30.1 Design Tools

38 **CROSS-REFERENCE** For
39 details on design, see
40 Chapters 5 through 9.

Current design tools consist mainly of graphical tools that create design diagrams. Design tools are sometimes embedded in a CASE tool with broader functions; some vendors advertise standalone design tools as CASE tools.

41 Graphical design tools generally allow you to express a design in common
42 graphical notations—UML, architecture block diagrams, hierarchy charts, entity
43 relationship diagrams, or class diagrams. Some graphical design tools support
44 only one notation. Others support a variety.

45 In one sense, these design tools are just fancy drawing packages. Using a simple
46 graphics package or pencil and paper, you can draw everything that the tool can
47 draw. But the tools offer valuable capabilities that a simple graphics package
48 can't. If you've drawn a bubble chart and you delete a bubble, a graphical design
49 tool will automatically rearrange the other bubbles, including connecting arrows
50 and lower-level bubbles connected to the bubble. The tool takes care of the
51 housekeeping when you add a bubble too. A design tool can enable you to move
52 between higher and lower levels of abstraction. A design tool will check the
53 consistency of your design, and some tools can create code directly from your
54 design.

55 30.2 Source-Code Tools

56 The tools available for working with source code are richer and more mature
57 than the tools available for working with designs.

58 **Editing**

59 This group of tools relates to editing source code.

60 **Integrated Development Environments (IDEs)**

61 **HARD DATA**
62 Some programmers estimate that they spend as much as 40 percent of their time
63 editing source code (Ratliff 1987, Parikh 1986). If that's the case, spending a few extra dollars for the best possible IDE is a good investment.

64 In addition to basic word-processing functions, good IDEs offer these features:

- 65 • Compilation and error detection from within the editor
66 • Compressed or outline views of programs (class names only or logical
67 structures without the contents)
68 • Jump to definitions of classes, routines, and variables
69 • Jump to all places where a class, routine, or variable is used
70 • Language-specific formatting
71 • Interactive help for the language being edited
72 • Brace (*begin-end*) matching
73 • Templates for common language constructs (the editor completing the
74 structure of a *for* loop after the programmer types *for*, for example)
75 • Smart indenting (including easily changing the indentation of a block of
76 statements when logic changes)
77 • Macros programmable in a familiar programming language
78 • Memory of search strings so that commonly used strings don't need to be
79 retyped
80 • Regular expressions in search-and-replace
81 • Search-and-replace across a group of files
82 • Editing multiple files simultaneously
83 • Multi-level undo

84 Considering some of the primitive editors still in use, you might be surprised to
85 learn that several editors include all of these capabilities.

86 **Multiple-File String Searching and Replacing**

87 If your editor doesn't support search and replace across multiple files, you can
88 still find supplementary tools to do that job. These tools are useful for search for
89 all occurrences of a class name or routine name. When you find an error in your
90 code, you can use such tools to check for similar errors in other files.

91 You can search for exact strings, similar strings (ignoring differences in
92 capitalization), or regular expressions. Regular expressions are particularly
93 powerful because they let you search for complex string patterns. If you wanted
94 to find all the array references containing magic numbers (digits "0" through
95 "9"), you could search for "[" followed by zero or more spaces, followed by one
96 or more digits, followed by zero or more spaces, followed by "]". One widely

97 available search tool is called “grep.” A grep query for magic numbers would
98 look like this:

99 grep "\[*[0-9]* *\]" *.c

100 You can make the search criteria more sophisticated to fine-tune the search.

101 It’s often helpful to be able to change strings across multiple files. For example,
102 if you want to give a routine, constant, or global variable a better name, you
103 might have to change the name in several files. Utilities that allow string changes
104 across multiple files make that easy to do, which is good because you should
105 have as few obstructions as possible to creating excellent class names, routine
106 names, and constant names. Common tools for handling multiple-file string
107 changes include Perl, AWK, and sed.

108 **Diff Tools**

109 Programmers often need to compare two files. If you make several attempts to
110 correct an error and need to remove the unsuccessful attempts, a file comparator
111 will make a comparison of the original and modified files and list the lines
112 you’ve changed. If you’re working on a program with other people and want to
113 see the changes they have made since the last time you worked on the code, a
114 comparator tool such as Diff will make a comparison of the current version with
115 the last version of the code you worked on and show the differences. If you
116 discover a new defect that you don’t remember encountering in an older version
117 of a program, rather than seeing a neurologist about amnesia, you can use a
118 comparator to compare current and old versions of the source code, determine
119 exactly what changed, and find the source of the problem. This functionality is
120 often built into revision control tools.

121 **Merge Tools**

122 One style of revision control locks source files so that only one person can
123 modify a file at a time. Another style allows multiple people to work on files
124 simultaneously and handles merging changes at check-in time. In this working
125 mode, tools that merge changes are critical. These tools typically perform simple
126 merges automatically and query the user for merges that conflict with other
127 merges or that are more involved.

128 **Source-Code Beautifiers**

129 **CROSS-REFERENCE** For
130 details on program layout,
131 see Chapter 31, “Layout and
132 Style.”
133
134 Source-code beautifiers spruce up your source code so that it looks consistent.
They highlight class and routine names, standardize your indentation style,
format comments consistently, and perform other similar functions. Some
beautifiers can put each routine onto a separate web page or printed page or
perform even more dramatic formatting. Many beautifiers let you customize the
way in which the code is beautified.

135 There are at least two classes of source code beautifiers. One class takes the
136 source code as input and produces much better looking output without changing
137 the original source code.

138 Another kind of tool changes the source code itself—standardizing indentation,
139 parameter list formatting, and so on. This capability is useful when working with
140 large quantities of legacy code. The tool can do much of the tedious formatting
141 work needed to make the legacy code conform to your coding style conventions.

142 **Interface Documentation Tools**

143 Some tools extract detailed programmer-interface documentation from source
144 code files. The code inside the source file uses clues such as `@tag` fields to
145 identify text that should be extracted. The interface documentation tool then
146 extracts that tagged text and presents it with nice formatting. JavaDoc is the most
147 prominent example of this kind of tool.

148 **Templates**

149 Templates help you exploit the simple idea of streamlining keyboarding tasks
150 that you do often and want to do consistently. Suppose you want a standard
151 comment prolog at the beginning of your routines. You could build a skeleton
152 prolog with the correct syntax and places for all the items you want in the
153 standard prolog. This skeleton would be a “template” you’d store in a file or a
154 keyboard macro. When you created a new routine, you could easily insert the
155 template into your source file. You can use the template technique for setting up
156 larger entities, such as classes and files, or smaller entities, such as loops.

157 If you’re working on a group project, templates are an easy way to encourage
158 consistent coding and documentation styles. Make templates available to the
159 whole team at the beginning of the project, and the team will use them because
160 they make its job easier—you get the consistency as a side benefit.

161 **Cross-Reference Tools**

162 A cross-reference tool lists variables and routines and all the places in which
163 they’re used—typically on web pages.

164 **Class Hierarchy Generators**

165 A class-hierarchy generator produces information about inheritance trees. This is
166 sometimes useful in debugging but is more often used for analyzing a program’s
167 structure or packaging a program into packages or subsystems. This functionality
168 is also available in some IDEs.

169 **Analyzing Code Quality**

170 Tools in this category examine the static source code to assess its quality.

171
172
173
174
175
176
177

Picky Syntax and Semantics Checkers

Syntax and semantics checkers supplement your compiler by checking code more thoroughly than the compiler normally does. Your compiler might check for only rudimentary syntax errors. A picky syntax checker might use nuances of the language to check for more subtle errors—things that aren’t wrong from a compiler’s point of view but that you probably didn’t intend to write. For example, in C++, the statement

178
179

```
while ( i = 0 ) ...
```

is a perfectly legal statement, but it’s usually meant to be

180
181
182
183
184
185
186
187
188

```
while ( i == 0 ) ...
```

The first line is syntactically correct, but switching = and == is a common mistake and the line is probably wrong. Lint is a picky syntax and semantics checker you can find in many C/C++ environments. Lint warns you about uninitialized variables, completely unused variables, variables that are assigned values and never used, parameters of a routine that are passed out of the routine without being assigned a value, suspicious pointer operations, suspicious logical comparisons (like the one in the example above), inaccessible code, and many other common problems. Other languages offer similar tools.

189

CROSS-REFERENCE For more information on metrics, see Section 28.4, “Measurement.”

190
191
192
193
194
195
196
197
198
199

Metrics Reporters

Some tools analyze your code and report on its quality. For example, you can obtain tools that report on the complexity of each routine so that you can target the most complicated routines for extra review, testing, or redesign. Some tools count lines of code, data declarations, comments, and blank lines in either entire programs or individual routines. They track defects and associate them with the programmers who made them, the changes that correct them, and the programmers who make the corrections. They count modifications to the software and note the routines that are modified the most often. Complexity analysis tools have been found to have about a 20% positive impact on maintenance productivity (Jones 2000).

200
201

Refactoring Source Code

A few tools aid in converting source code from one format to another.

202

CROSS-REFERENCE For more on refactoring, see Chapter 24, “Refactoring.”

203
204
205
206
207
208

Refactorers

A refactoring program supports common code refactorings either on a standalone basis or integrated into an IDE. Refactoring browsers allow you to change the name of a class across an entire code base easily. They allow you to extract a routine simply by highlighting the code you’d like to turn into a new routine, entering the new routine’s name, and order parameters in a parameter list. Refactorers make code changes quicker and less error prone. They’re available

209
210
211
for Java and Smalltalk and are becoming available for other languages. For more
about refactoring tools, see Chapter 14, “Refactoring Tools” in *Refactoring*
(Fowler 1999).

212 **Restructurers**

213
214
215
216
217
218
219
220
221
A restructurer will convert a plate of spaghetti code with *gos* to a more
nutritious entrée of better structured code without *gos*. Capers Jones reports
that in maintenance environments code restructuring tools can have a 25-30
percent positive impact on maintenance productivity (Jones 2000). A restructurer
has to make a lot of assumptions when it converts code, and if the logic is
terrible in the original, it will still be terrible in the converted version. If you’re
doing a conversion manually, however, you can use a restructurer for the general
case and hand-tune the hard cases. Alternatively, you can run the code through
the restructurer and use it for inspiration for the hand conversion.

222 **Code Translators**

223
224
225
226
Some tools translate code from one language to another. A translator is useful
when you have a large code base that you’re moving to another environment.
The hazard in using a language translator is that if you start with bad code the
translator simply translates the bad code into an unfamiliar language.

227 **Version Control**

228 **CROSS-REFERENCE** These tools and their benefits are
229 described in “Software Code Changes” in Section 28.2.
230
You can deal with proliferating software versions by using version-control tools
for

- 231 • Source-code control
- 232 • Make-style dependency control
- 233 • Project documentation versioning

234
Version control tools have been found to have as much as 20% positive impact
on

235 **Data Dictionaries**

236
237
238
239
240
241
242
243
A data dictionary is a database that describes all the significant data in a project.
In many cases, the data dictionary focuses primarily on database schemas. On
large projects, a data dictionary is also useful for keeping track of the hundreds
or thousands of class definitions. On large team projects, it’s useful for avoiding
naming clashes. A clash might be a direct, syntactic clash, in which the same
name is used twice, or it might be a more subtle clash (or gap) in which different
names are used to mean the same thing or the same name is used to mean subtly
different things. For each data item (database table or class), the data dictionary

244
245

contains the item's name and description. The dictionary might also contain notes about how the item is used.

246

30.3 Executable-Code Tools

247
248

Tools for working with executable code are as rich as the tools for working with source code.

249

Code Creation

250

The tools described in this section help with code creation.

251

Compilers and Linkers

252
253

Compilers convert source code to executable code. Most programs are written to be compiled, although some are still interpreted.

254
255
256
257
258

A standard linker links one or more object files, which the compiler has generated from your source files, with the standard code needed to make an executable program. Linkers typically can link files from multiple languages, allowing you to choose the language that's most appropriate for each part of your program without your having to handle the integration details yourself.

259
260
261
262

An overlay linker helps you put 10 pounds in a 5-pound sack by developing programs that execute in less memory than the total amount of space they consume. An overlay linker creates an executable file that loads only part of itself into memory at any one time, leaving the rest on a disk until it's needed.

263

Make

264
265
266
267

Make is a utility that's associated with UNIX and the C/C++ languages. The purpose of make is to minimize the time needed to create current versions of all your object files. For each object file in your project, you specify the files that the object file depends on and how to make it.

268
269
270
271
272

Suppose you have an object file named *userface.obj*. In the make file, you indicate that to make *userface.obj*, you have to compile the file *userface.cpp*. You also indicate that *userface.cpp* depends on *userface.h*, *stdlib.h*, and *project.h*. The concept of "depends on" simply means that if *userface.h*, *stdlib.h*, or *project.h* changes, *userface.cpp* needs to be recompiled.

273
274
275
276

When you build your program, make checks all the dependencies you've described and determines the files that need to be recompiled. If 5 of your 25 source files depend on data definitions in *userface.h* and it changes, make automatically recompiles the 5 files that depend on it. It doesn't recompile the 20

277 files that don't depend on *userface.h*. Using make beats the alternatives of
278 recompiling all 25 files or recompiling each file manually, forgetting one, and
279 getting weird out-of-synch errors. Overall, make substantially improves the time
280 and reliability of the average compile-link-run cycle.

281 Some groups have found interesting alternatives to make. For example, the
282 Microsoft Word group found that simply rebuilding all source files was faster
283 than performing extensive dependency checking with make as long as the source
284 files themselves were optimized (header file contents and so on). With this
285 approach, the average developer's machine on the Word project could rebuild
286 the entire Word executable—several million lines of code—in about 13 minutes.

287 **Code Libraries**

288 A good way to write high-quality code in a short amount of time is not to write it
289 all—but to buy it instead. You can find high-quality libraries in at least these
290 areas:

- 291 • Container classes
- 292 • Credit card transaction services (e-commerce services)
- 293 • Cross-platform development tools. You might write code that executes in
294 Microsoft Windows, Apple Macintosh, and the X Window System just by
295 recompiling for each environment.
- 296 • Data compression tools
- 297 • Data types and algorithms
- 298 • Database operations and data-file manipulation tools
- 299 • Diagramming, graphing, and charting tools
- 300 • Imaging tools
- 301 • License managers
- 302 • Mathematical operations
- 303 • Networking and internet communications tools
- 304 • Report generators and report query builders
- 305 • Security and encryption tools
- 306 • Spreadsheet and grid tools
- 307 • Text and spelling tools
- 308 • Voice, phone, and fax tools

309 **Code Generation Wizards**

310 If you can't find the code you want, how about getting someone else to write it
311 instead? You don't have to put on your yellow plaid jacket and slip into a car
312 salesman's patter to con someone else into writing your code. You can find tools
313 that write code for you, and such tools are often integrated into IDEs.

314 Code-generating tools tend to focus on database applications, but that includes a
315 lot of applications. Commonly available code generators write code for
316 databases, user interfaces, and compilers. The code they generate is rarely as
317 good as code generated by a human programmer, but many applications don't
318 require handcrafted code. It's worth more to some users to have 10 working
319 applications than to have one that works exceptionally well.

320 Code generators are also useful for making prototypes of production code. Using
321 a code generator, you might be able to hack out a prototype in a few hours that
322 demonstrates key aspects of a user interface or you might be able to experiment
323 with various design approaches. It might take you several weeks to hand-code as
324 much functionality. If you're just experimenting, why not do it in the cheapest
325 possible way?

326 **Setup and Installation**

327 Numerous vendors provide tools that support creation of setup programs. These
328 tools typically support creation of disks, CDs, DVDs, or installing over the web.
329 They check whether common library files already exist on the target installation
330 machine, perform version checking, and so on.

331 **Macro Preprocessors**

332 **CROSS-REFERENCE** For
333 guidelines on using simple
334 macro substitutions, see
335 Section 12.7, "Named
336 Constants." For guidelines on
337 using macro routines, see
338 Section 7.7, "Macro Routines."
339
340
341
342

If you've programmed in C++ using C++'s macro preprocessor, you probably find it hard to conceive of programming in a language without a preprocessor. Macros allow you to create simple named constants with no run-time penalty. For example, if you use *MAX_EMPS* instead of the literal *5000*, the preprocessor will substitute the literal value *5000* before the code is compiled.

A macro preprocessor will also allow you to create more complicated functional replacements for substitution at compile time—and again, without any run-time penalty. This gives you the twin advantages of readability and modifiability. Your code is more readable because you've used a macro that you have presumably given a good name. It's more modifiable because you've put all the code in one place, where you can easily change it.

343 **CROSS-REFERENCE** For
344 details on moving debugging
345 aids in and out of the code,
346 see “Plan to Remove
347 Debugging Aids” in Section
348 8.6.

349
350

351
352
353

354 CC2E.COM/3091
355
356

357

358 **CROSS-REFERENCE** These tools and their benefits are
359 described in Section 23.5,
“Debugging Tools—Obvious
360 and Not-So-Obvious.”

361
362
363
364
365

366

367 **CROSS-REFERENCE** These tools and their benefits are
368 described in Section 22.5,
“Test-Support Tools.”

369
370
371
372
373
374

Preprocessor functions are good for debugging because they’re easy to shift into development code and out of production code. During development, if you want to check memory fragmentation at the beginning of each routine, you can use a macro at the beginning of each routine. You might not want to leave the checks in production code, so for the production code you can redefine the macro so that it doesn’t generate any code at all. For similar reasons, preprocessor macros are good for writing code that’s targeted to be compiled in multiple environments—for example, in both Microsoft Windows and Linux.

If you use a language with primitive control constructs, such as assembler, you can write a control-flow preprocessor to emulate the structured constructs of *if-then-else* and *while* loops in your language.

If you’re not fortunate enough to program in a language that has a preprocessor, you can use a standalone preprocessor as part of your build process. One readily available preprocessor is M4, available from www.gnu.org/software/m4/.

Debugging

These tools help in debugging:

- Compiler warning messages
- Test scaffolding
- File comparators (for comparing different versions of source-code files)
- Execution profilers
- Trace monitors
- Interactive debuggers—both software and hardware.

Testing tools, discussed next, are related to debugging tools.

Testing

These features and tools can help you do effective testing:

- Automatic test frameworks like JUnit, NUnit, CppUnit and so on
- Automated test generators
- Test-case record and playback utilities
- Coverage monitors (logic analyzers and execution profilers)
- Symbolic debuggers
- System perturbers (memory fillers, memory shakers, selective memory failers, memory-access checkers)

- 375 • Diff tools (for comparing data files, captured output, and screen images)
376 • Scaffolding
377 • Defect tracking software

378 **Code Tuning**

379 These tools can help you fine-tune your code.

380 **Execution Profilers**

381 An execution profiler watches your code while it runs and tells you how many
382 times each statement is executed or how much time the program spends on each
383 statement. Profiling your code while it's running is like having a doctor press a
384 stethoscope to your chest and tell you to cough. It gives you insight into how
385 your program works, where the hot spots are, and where you should focus your
386 code-tuning efforts.

387 **Assembler Listings and Disassemblers**

388 Some day you might want to look at the assembler code generated by your high-
389 level language. Some high-level-language compilers generate assembler listings.
390 Others don't, and you have to use a disassembler to recreate the assembler from
391 the machine code that the compiler generates. Looking at the assembler code
392 generated by your compiler shows you how efficiently your compiler translates
393 high-level-language code into machine code. It can tell you why high-level code
394 that looks fast runs slowly. In Chapter 26 on code-tuning techniques, several of
395 the benchmark results are counterintuitive. While benchmarking that code, I
396 frequently referred to the assembler listings to better understand the results that
397 didn't make sense in the high-level language.

398 If you're not comfortable with assembly language and you want an introduction,
399 you won't find a better one than comparing each high-level-language statement
400 you write to the assembler instructions generated by the compiler. A first
401 exposure to assembler is often a loss of innocence. When you see how much
402 code the compiler creates—how much more than it needs to—you'll never look
403 at your compiler in quite the same way again.

404 Conversely, in some environments the compiler must generate extremely
405 complex code. Studying the compiler output can foster an appreciation for just
406 how much work would be required to program in a lower level language.

407

408

409

410

411

412

413

414

415

416

417

418

419

420

421

422

423

424

425

426

427

428

429

430

431

432

433

434

435

30.4 Tool-Oriented Environments

Some environments have proven to be better suited to tool-oriented programming than others. This section looks at three examples.

UNIX

UNIX and the philosophy of programming with small, sharp tools are inseparable. The UNIX environment is famous for its collection of small tools with funny names that work well together: grep, diff, sort, make, crypt, tar, lint, ctags, sed, awk, vi, and others. The C and C++ languages, closely coupled with UNIX, embody the same philosophy; the standard C++ library is composed of small functions that can easily be composed into larger functions because they work so well together.

Some programmers work so productively in UNIX that they take it with them. They use UNIX work-alike tools to support their UNIX habits in Microsoft Windows and other environments. One tribute to the success of the UNIX paradigm is the availability of tools that put a UNIX costume on a Windows machine.

30.5 Building Your Own Programming Tools

Suppose you're given five hours to do the job and you have a choice:

1. Do the job comfortably in five hours, or
2. Spend four hours and 45 minutes feverishly building a tool to do the job, and then have the tool do the job in 15 minutes.

Most good programmers would choose the first option one time out of a million and the second option in every other case. Building tools is part of the warp and woof of programming. Nearly all large organizations (organizations with more than 1000 programmers) have internal tool and support groups. Many have proprietary requirements and design tools that are superior to those on the market (Jones 2000).

You can write many of the tools described in this chapter. It might not be cost effective to do it, but there aren't any mountainous technical barriers to doing it.

436

437

438

439

440

441

442

443

444

445

446

447

448

449

450

451

452

453

454

455

456

457

458

459

460

461

462

463

464

465

466

467

468

469

470

471

472

473

Project-Specific Tools

Most medium and large projects need special tools unique to the project. For example, you might need tools to generate special kinds of test data, to verify the quality of data files, or to emulate hardware that isn't yet available. Here are some examples of project-specific tool support:

- An aerospace team was responsible for developing in-flight software to control an infrared sensor and analyze its data. To verify the performance of the software, an in-flight data recorder documented the actions of the in-flight software. Engineers wrote custom data-analysis tools to analyze the performance of the in-flight systems. After each flight, they used the custom tools to check the primary systems.
- Microsoft planned to include a new font technology in a release of its Windows graphical environment. Since both the font data files and the software to display the fonts were new, errors could have arisen from either the data or the software. Microsoft developers wrote several custom tools to check for errors in the data files, which improved their ability to discriminate between font data errors and software errors.
- An insurance company developed an ambitious system to calculate its rate increases. Because the system was complicated and accuracy was essential, hundreds of computed rates needed to be checked carefully, even though hand calculating a single rate took several minutes. The company wrote a separate software tool to compute rates one at a time. With the tool, the company could compute a single rate in a few seconds and check rates from the main program in a small fraction of the time it would have taken to check the main program's rates by hand.

Part of planning for a project should be thinking about the tools that might be needed and allocating time for building them.

Scripts

A script is a tool that automates a repetitive chore. In some systems, scripts are called batch files or macros. Scripts can be simple or complex, and some of the most useful are the easiest to write. For example, I keep a journal, and to protect my privacy, I encrypt it except when I'm writing in it. To make sure that I always encrypt and decrypt it properly, I have a script that decrypts my journal, executes the word processor, and then encrypts the journal. The script looks like this:

```
crypto c:\word\journal.* %1 /d /Es /s
word c:\word\journal.doc
crypto c:\word\journal.* %1 /Es /s
```

474 The %1 is the field for my password which, for obvious reasons, isn't included
475 in the script. The script saves me the work of typing all the parameters (and
476 mistyping them) and ensures that I always perform all the operations and
477 perform them in the right order.

478 If you find yourself typing something longer than about five characters more
479 than a few times a day, it's a good candidate for a script or batch file. Examples
480 include compile/link sequences, backup commands, and any command with a lot
481 of parameters.

482 30.6 Tool Fantasyland

483 **CROSS-REFERENCE** Tool
484 availability depends partly on
485 the maturity of the technical
486 environment. For more on
487 this, see Section 4.3, "Your
488 Location on the Technology
Wave."

489 Fortran did succeed in making it possible for scientists and engineers to write
490 programs, but from our vantage point today, Fortran appears to be a
491 comparatively low level programming language. It hardly eliminated the need
492 for programmers, and what the industry experienced with Fortran is indicative of
493 progress in the software industry as a whole.

494 The software industry constantly develops new tools that reduce or eliminate
495 some of the most tedious aspects of programming—details of laying out source
496 statements; steps needed to edit, compile, link, and run a program; work needed
497 to find mismatched braces; the number of steps needed to create standard
498 message boxes; and so on. As each of these new tools begins to demonstrate
499 incremental gains in productivity, pundits extrapolate those gains out to infinity,
500 assuming that the gains will eventually "eliminate the need for programming."
501 But what's happening in reality is that each new programming innovation arrives
502 with a few blemishes. As time goes by, the blemishes are removed, and that
503 innovation's full potential is realized. However, once the fundamental tool
504 concept is realized, further gains are achieved by stripping away the accidental
505 difficulties that were created as side effects of creating the new tool. Elimination
506 of these accidental difficulties does not increase productivity per se; it simply
507 eliminates the "one step back" from the typical "two steps forward, one step
508 back" equation.

509 Over the past several decades programmers have seen numerous tools that were
510 supposed to eliminate programming. First it was third generation languages.

511 Then it was fourth generation languages. Then it was automatic programming.
512 Then it was CASE tools. Then it was visual programming. Each of these
513 advances spun off valuable, incremental improvements to computer
514 programming—and collectively they have made programming unrecognizable to
515 anyone who learned programming before these advances. But none of these
516 innovations succeeded in eliminating programming.

517 The reason for this dynamic is that, at its essence, programming is fundamentally
518 *hard*—even with good tool support. (Reasons for this are described in
519 “Accidental and Essential Difficulties” in Section 5.2.) No matter what tools are
520 available, programmers will have to wrestle with the messy real world; we will
521 have to think rigorously about sequences, dependencies, and exceptions; and we
522 will have to deal with end users who can’t make up their minds. We will always
523 have to wrestle with ill-defined interfaces to other software and hardware, and
524 we will have to account for regulations, business rules, and other sources of
525 complexity that arise from outside the world of computer programming.

526 We will always need people who can bridge the gap between the real world
527 problem to be solved and the computer that is supposed to be solving the
528 problem. These people will be called programmers regardless of whether we’re
529 manipulating machine registers in assembler or dialog boxes in Visual Basic. As
530 long as we have computers, we’ll need people who tell the computers what to do,
531 and that activity will be called programming.

532 When you hear a tool vendor claim, “This new tool will eliminate computer
533 programming”—run! Or at least smile to yourself at the vendor’s naive
534 optimism.

CC2E.COM/3098

535 Additional Resources

536 CC2E.COM/3005 www.sdmagazine.com/jolts. *Software Development Magazine*’s annual Jolt
537 Productivity award website is a good source of information about the best
538 current tools.

539 Hunt, Andrew and David Thomas. *The Pragmatic Programmer*, Boston, Mass.:
540 Addison Wesley, 2000. Section 3 of this book provides an in-depth discussion of
541 programming tools including editors, code generators, debuggers, source code
542 control and related tools.

543 CC2E.COM/3012 Vaughn-Nichols, Steven. “Building Better Software with Better Tools,” *IEEE*
544 *Computer*, September 2003, pp. 12-14. This article surveys tool initiatives led by
545 IBM, Microsoft Research, and Sun Research.

546 Glass, Robert L. *Software Conflict: Essays on the Art and Science of Software*
547 *Engineering*. Englewood Cliffs, N.J.: Yourdon Press, 1991. The chapter titled
548 “Recommended: A Minimum Standard Software Toolset” provides a thoughtful
549 counterpoint to the more-tools-is-better view. Glass argues for the identification
550 of a minimum set of tools that should be available to all developers and proposes
551 a starting kit.

552 Jones, Capers. *Estimating Software Costs*, New York: McGraw-Hill, 1998.

553 Boehm, Barry, et al. *Software Cost Estimation with Cocomo II*, Reading, Mass.:
554 Addison Wesley, 2000. Both the Jones and the Boehm books devote sections to
555 the impact of tool use on productivity.

556 Kernighan, Brian W., and P. J. Plauger. *Software Tools*. Reading, Mass.:
557 Addison-Wesley, 1976.

558 Kernighan, Brian W., and P. J. Plauger. *Software Tools in Pascal*. Reading,
559 Mass.: Addison-Wesley, 1981. The two Kernighan and Plauger books cover the
560 same ground—the first in Rational Fortran, the second in Pascal. The books have
561 two agendas and meet both nicely. The first is to give you the source code for a
562 useful set of programming tools. The tools include a multiple-file finder, a
563 multiple-file changer, a macro preprocessor, a diff tool, an editor, and a print
564 utility. The second agenda is to expose you to good programming practices by
565 showing you how each of the tools is developed. Both authors are expert
566 programmers, and the books are full of design-decision rationales and analyses
567 of trade-offs, adding up to rare and valuable insight into how experienced
568 designers and programmers approach their work.

CC2E.COM/3019

569

Checklist: Programming Tools

- 570 Do you have an effective IDE?
571 Does your IDE support outline view of your program; jumping to definitions
572 of classes, routines, and variables; source code formatting; brace matching
573 or begin-end matching; multiple file string search and replace; convenient
574 compilation; and integrated debugging?
575 Do you have tools that automate common refactorings?
576 Are you using version control to manage source code, content, requirements,
577 designs, project plans, and other project artifacts?
578 If you’re working on a very large project, are you using a data dictionary or
579 some other central repository that contains authoritative descriptions of each
580 class used in the system?
581 Have you considered code libraries as alternatives to writing custom code,
582 where available?

- 583 Are you making use of an interactive debugger?
584 Do you use make or other dependency-control software to build programs
585 efficiently and reliably?
586 Does your test environment include an automated test framework, automated
587 test generators, coverage monitors, system perturbers, diff tools, and defect
588 tracking software?
589 Have you created any custom tools that would help support your specific
590 project's needs, especially tools that automate repetitive tasks?
591 Overall, does your environment benefit from adequate tool support?
-

592

593 **Key Points**

- 594
 - 595 • Programmers sometimes overlook some of the most powerful tools for years
596 before discovering them.
 - 597 • Good tools can make your life a lot easier.
 - 598 • Tools are readily available for editing, analyzing code quality, refactoring,
599 version control, debugging, testing, and code tuning.
 - 600 • You can make many of the special-purpose tools you need.
 - 601 • Good tools can reduce the more tedious aspects of software development,
602 but they can't eliminate the need for programming, though they will
 continue to reshape what we mean by "programming."

31

Layout and Style

Contents

- 31.1 Layout Fundamentals
- 31.2 Layout Techniques
- 31.3 Layout Styles
- 31.4 Laying Out Control Structures
- 31.5 Laying Out Individual Statements
- 31.6 Laying Out Comments
- 31.7 Laying Out Routines
- 31.8 Laying Out Classes

Related Topics

Self-documenting code: Chapter 32

THIS CHAPTER TURNS TO AN AESTHETIC ASPECT of computer programming—the layout of program source code. The visual and intellectual enjoyment of well-formatted code is a pleasure that few nonprogrammers can appreciate. But programmers who take pride in their work derive great artistic satisfaction from polishing the visual structure of their code.

The techniques in this chapter don't affect execution speed, memory use, or other aspects of a program that are visible from outside the program. They affect how easy it is to understand the code, review it, and revise it months after you write it. They also affect how easy it is for others to read, understand, and modify once you're out of the picture.

This chapter is full of the picky details that people refer to when they talk about “attention to detail.” Over the life of a project, attention to such details makes a difference in the initial quality and the ultimate maintainability of the code you write. Such details are too integral to the coding process to be changed effectively later. If they're to be done at all, they must be done during initial construction. If you're working on a team project, have your team read this chapter and agree on a team style before you begin coding.

31 You might not agree with everything you read here. But the point is less to win
32 your agreement than to convince you to consider the issues involved in format-
33 ting style. If you have high blood pressure, move on to the next chapter. It's less
34 controversial.

35 31.1 Layout Fundamentals

36 This section explains the theory of good layout. The rest of the chapter explains
37 the practice.

38 Layout Extremes

39 Consider the routine shown in Listing 31-1:

40 **Listing 31-1. Java layout example #1.**

41 **CODING HORROR**

```
42 /* Use the insertion sort technique to sort the "data" array in ascending order.  
43 This routine assumes that data[ firstElement ] is not the first element in data and  
44 that data[ firstElement-1 ] can be accessed. */ public void InsertionSort( int[]  
45 data, int firstElement, int lastElement ) { /* Replace element at lower boundary  
46 with an element guaranteed to be first in a sorted list. */ int lowerBoundary =  
47 data[ firstElement-1 ]; data[ firstElement-1 ] = SORT_MIN; /* The elements in  
48 positions firstElement through sortBoundary-1 are always sorted. In each pass  
49 through the loop, sortBoundary is increased, and the element at the position of the  
50 new sortBoundary probably isn't in its sorted place in the array, so it's inserted  
51 into the proper place somewhere between firstElement and sortBoundary. */ for ( int  
52 sortBoundary = firstElement+1; sortBoundary <= lastElement; sortBoundary++ ) { int  
53 insertVal = data[ sortBoundary ]; int insertPos = sortBoundary; while ( insertVal <  
54 data[ insertPos-1 ] ) { data[ insertPos ] = data[ insertPos-1 ]; insertPos =  
55 insertPos-1; } data[ insertPos ] = insertVal; } /* Replace original lower-boundary  
56 element */ data[ firstElement-1 ] = lowerBoundary; }
```

57 The routine is syntactically correct. It's thoroughly commented and has good
58 variable names and clear logic. If you don't believe that, read it and find a mis-
59 take! What the routine doesn't have is good layout. This is an extreme example,
60 headed toward "negative infinity" on the number line of bad-to-good layout.

61 Listing 31-2 is a less extreme example:

62 **Listing 31-2. Java layout example #2.**

63 **CODING HORROR**

```
64 /* Use the insertion sort technique to sort the "data" array in ascending  
65 order. This routine assumes that data[ firstElement ] is not the  
66 first element in data and that data[ firstElement-1 ] can be accessed. */  
67 public void InsertionSort( int[] data, int firstElement, int lastElement ) {  
68 /* Replace element at lower boundary with an element guaranteed to be first in a  
69 sorted list. */  
70 int lowerBoundary = data[ firstElement-1 ];
```

```
69     data[ firstElement-1 ] = SORT_MIN;
70     /* The elements in positions firstElement through sortBoundary-1 are
71      always sorted. In each pass through the loop, sortBoundary
72      is increased, and the element at the position of the
73      new sortBoundary probably isn't in its sorted place in the
74      array, so it's inserted into the proper place somewhere
75      between firstElement and sortBoundary. */
76     for (
77         int sortBoundary = firstElement+1;
78         sortBoundary <= lastElement;
79         sortBoundary++
80     ) {
81         int insertVal = data[ sortBoundary ];
82         int insertPos = sortBoundary;
83         while ( insertVal < data[ insertPos-1 ] ) {
84             data[ insertPos ] = data[ insertPos-1 ];
85             insertPos = insertPos-1;
86         }
87         data[ insertPos ] = insertVal;
88     }
89     /* Replace original lower-boundary element */
90     data[ firstElement-1 ] = lowerBoundary;
91 }
```

This code is the same as Listing 31-1's. Although most people would agree that the code's layout is much better than the first example's, the code is still not very readable. The layout is still crowded and offers no clue to the routine's logical organization. It's at about 0 on the number line of bad-to-good layout. The first example was contrived, but the second one isn't at all uncommon. I've seen programs several thousand lines long with layout at least as bad as this; with no documentation and bad variable names, overall readability was worse than in this example. This code is formatted for the computer. There's no evidence that the author expected the code to be read by humans. Listing 31-3 is an improvement.

101 **Listing 31-3. Java layout example #3.**

```
102    /* Use the insertion sort technique to sort the "data" array in ascending
103       order. This routine assumes that data[ firstElement ] is not the
104       first element in data and that data[ firstElement-1 ] can be accessed.
105    */
106
107    public void InsertionSort( int[] data, int firstElement, int lastElement ) {
108        // Replace element at lower boundary with an element guaranteed to be
109        // first in a sorted list.
110        int lowerBoundary = data[ firstElement-1 ];
111        data[ firstElement-1 ] = SORT_MIN;
112
113        /* The elements in positions firstElement through sortBoundary-1 are
```

```
114     always sorted. In each pass through the loop, sortBoundary  
115     is increased, and the element at the position of the  
116     new sortBoundary probably isn't in its sorted place in the  
117     array, so it's inserted into the proper place somewhere  
118     between firstElement and sortBoundary.  
119     */  
120     for ( int sortBoundary = firstElement + 1; sortBoundary <= lastElement;  
121         sortBoundary++ ) {  
122         int insertVal = data[ sortBoundary ];  
123         int insertPos = sortBoundary;  
124         while ( insertVal < data[ insertPos - 1 ] ) {  
125             data[ insertPos ] = data[ insertPos - 1 ];  
126             insertPos = insertPos - 1;  
127         }  
128         data[ insertPos ] = insertVal;  
129     }  
130  
131     // Replace original lower-boundary element  
132     data[ firstElement - 1 ] = lowerBoundary;  
133 }
```

140 **FURTHER READING** For
141 details on the typographic
142 approach to formatting
143 source code, see *Human Fac-
tors and Typography for
More Readable Programs*
144 (Baecker and Marcus 1990).

145
146
147
148
149
150
151

152
153
154

This layout of the routine is a strong positive on the number line of bad-to-good layout. The routine is now laid out according to principles that are explained throughout this chapter. The routine has become much more readable, and the effort that has been put into documentation and good variable names is now evident. The variable names were just as good in the earlier examples, but the layout was so poor that they weren't helpful.

The only difference between this example and the first two is the use of white space—the code and comments are exactly the same. White space is of use only to human readers—your computer could interpret any of the three fragments with equal ease. Don't feel bad if you can't do as well as your computer!

Still another formatting example is shown in Figure 31-1. It's based on a source-code format developed by Ronald M. Baecker and Aaron Marcus (1990). In addition to using white space as the previous example did, it uses shading, different typefaces, and other typographic techniques. Baecker and Marcus have developed a tool that automatically prints normal source code in a way similar to that shown in Figure 31-1. Although the tool isn't commercially available, this sample is a glimpse of the source-code layout support that tools will offer within the next few years.

The Fundamental Theorem of Formatting

The Fundamental Theorem of Formatting is that good visual layout shows the logical structure of a program.

155 KEY POINT

156
157
158
159
160
161
162

163 *Any fool can write code
that a computer can un-
derstand. Good pro-
grammers write code that
humans can understand.*
164 —Martin Fowler
165
166
167
168

169
170
171

172
173
174
175
176
177
178
179
180
181
182
183

184
185
186
187
188
189
190
191

Making the code look pretty is worth something, but it's worth less than showing the code's structure. If one technique shows the structure better and another looks better, use the one that shows the structure better. This chapter presents numerous examples of formatting styles that look good but misrepresent the code's logical organization. In practice, prioritizing logical representation usually doesn't create ugly code—unless the logic of the code is ugly. Techniques that make good code look good and bad code look bad are more useful than techniques that make all code look good.

Human and Computer Interpretations of a Program

Layout is a useful clue to the structure of a program. Whereas the computer might care exclusively about braces or *begin* and *end*, a human reader is apt to draw clues from the visual presentation of the code. Consider the code fragment in Listing 31-4, in which the indentation scheme makes it look to a human as if three statements are executed each time the loop is executed.

F31xx01

Figure 31-1.

Source-code formatting that exploits typographic features.

Listing 31-4. Java example of layout that tells different stories to humans and computers.

```
// swap left and right elements for whole array
for ( i = 0; i < MAX_ELEMENTS; i++ )
    leftElement = left[ i ];
    left[ i ]   = right[ i ];
    right[ i ]  = leftElement;
```

If the code has no enclosing braces, the compiler will execute the first statement *MAX_ELEMENTS* times and the second and third statements one time each. The indentation makes it clear to you and me that the author of the code wanted all three statements to be executed together and intended to put braces around them. That won't be clear to the compiler.

Listing 31-5 is another example:

Listing 31-5. Another Java example of layout that tells different stories to humans and computers.

```
x = 3+4 * 2+7;
```

A human reader of this code would be inclined to interpret the statement to mean that *x* is assigned the value $(3+4) * (2+7)$, or 63. The computer will ignore the white space and obey the rules of precedence, interpreting the expression as $3 + (4*2) + 7$, or 18. The point is that a good layout scheme would make the visual

192 structure of a program match the logical structure, or tell the same story to the
193 human that it tells to the computer.

194 How Much Is Good Layout Worth?

195 *Our studies support the claim that knowledge of pro-*
196 *gramming plans and rules of programming discourse can have*
197 *a significant impact on program comprehension. In their book*
198 *called [The] Elements of [Programming] Style, Kernighan and*
199 *Plauger also identify what we would call discourse rules. Our*
200 *empirical results put teeth into these rules: It is not merely a*
201 *matter of aesthetics that programs should be written in a par-*
202 *ticular style. Rather there is a psychological basis for writing*
203 *programs in a conventional manner: programmers have*
204 *strong expectations that other programmers will follow these*
205 *discourse rules. If the rules are violated, then the utility af-*
206 *firmed by the expectations that programmers have built up*
207 *over time is effectively nullified. The results from the experi-*
208 *ments with novice and advanced student programmers and*
209 *with professional programmers described in this paper pro-*
210 *vide clear support for these claims.*

211

Elliot Soloway and Kate Ehrlich

212 **CROSS-REFERENCE** Goo
213 d layout is one key to read-
214 ability. For details on the
215 value of readability, see Sec-
216 tion 34.3, “Write Programs
217 for People First, Computers
218 Second.”

219

220

221

222

223

224

225

226

227

228

229

In layout, perhaps more than in any other aspect of programming, the difference between communicating with the computer and communicating with human readers comes into play. The smaller part of the job of programming is writing a program so that the computer can read it; the larger part is writing it so that other humans can read it.

In their classic paper “Perception in Chess,” Chase and Simon reported on a study that compared the abilities of experts and novices to remember the positions of pieces in chess (1973). When pieces were arranged on the board as they might be during a game, the experts’ memories were far superior to the novices’. When the pieces were arranged randomly, there was little difference between the memories of the experts and the novices. The traditional interpretation of this result is that an expert’s memory is not inherently better than a novice’s but that the expert has a knowledge structure that helps him or her remember particular kinds of information. When new information corresponds to the knowledge structure—in this case, the sensible placement of chess pieces—the expert can remember it easily. When new information doesn’t correspond to a knowledge structure—the chess pieces are randomly positioned—the expert can’t remember it any better than the novice.

230 A few years later, Ben Shneiderman duplicated Chase and Simon's results in the
231 computer-programming arena and reported his results in a paper called "Explora-
232 tory Experiments in Programmer Behavior" (1976). Shneiderman found that
233 when program statements were arranged in a sensible order, experts were able to
234 remember them better than novices. When statements were shuffled, the experts'
235 superiority was reduced. Shneiderman's results have been confirmed in other
236 studies (McKeithen et al. 1981, Soloway and Ehrlich 1984). The basic concept
237 has also been confirmed in the games Go and bridge and in electronics, music,
238 and physics (McKeithen et al. 1981).

239 After I published the first edition of this book, Hank, one of the programmers
240 who reviewed the manuscript commented that, "I was surprised that you didn't
241 argue more strongly in favor of a brace style that looks like this:

242 for (...)
243 {
244 }
245

"I was surprised that you even included the brace style that looked like this:

246 for (...) {
247 }
248

"I thought that, with both Tony and me arguing for the first style, you'd prefer
249 that."

250 I responded, "You mean you were arguing for the first style, and Tony was argu-
251 ing for the second style, don't you? Tony argued for the second style, not the
252 first."

253 Hank responded, "That's funny. The last project Tony and I worked on together,
254 I preferred style #2, and Tony preferred style #1. We spent the whole project
255 arguing about which style was best. I guess we talked one another into preferring
256 each other's styles!"

257 **KEY POINT**
258 This experience as well as the studies cited above suggest that structure helps
259 experts to perceive, comprehend, and remember important features of programs.
260 Given the variety of styles of layout and the tenacity with which programmers
261 cling to their own styles, even when they're vastly different from other styles,
262 it's easy to believe that the details of a specific method of structuring a program
are much less important than the fact that the program is structured at all.

263 **Layout as Religion**

264 The importance to comprehension and memory of structuring one's environment
265 in a familiar way has led some researchers to hypothesize that layout might harm
266 an expert's ability to read a program if the layout is different from the scheme

267 the expert uses (Sheil 1981, Soloway and Ehrlich 1984). That possibility, com-
268 pounded by the fact that layout is an aesthetic as well as a logical exercise,
269 means that debates about program formatting often sound more like religious
270 wars than philosophical discussions.

271 At a coarse level, it's clear that some forms of layout are better than others. The
272 successively better layouts of the same code at the beginning of the chapter made
273 that evident. This book won't steer clear of the finer points of layout just because
274 they're controversial. Good programmers should be open-minded about their
275 layout practices and accept practices proven to be better than the ones they're
276 used to, even if adjusting to a new method results in some initial discomfort.



277
278
279
280
281
282

F31xx01

Figure 31-1

283

284 *The results point out the
285 fragility of programming
286 expertise: advanced pro-
287 grammers have strong
288 expectations about what
289 programs should look
290 like, and when those ex-
291 pectations are violated—
292 in seemingly innocuous
ways—their performance
293 drops drastically.
294 —Elliot Soloway and
Kate Ehrlich*

Source code formatting can be a religious topic to some developers. If you're mixing software and religion, you might read Section 34.9, "Thou Shalt Rend Software and Religion Asunder" before reading the rest of this chapter.

Objectives of Good Layout

Many decisions about layout details are a matter of subjective aesthetics—often, you can accomplish the same goal in many ways. You can make debates about subjective issues less subjective if you explicitly specify the criteria for your preferences. Explicitly, then, a good layout scheme should:

Accurately represent the logical structure of the code

That's the Fundamental Theorem of Formatting again—the primary purpose of good layout is to show the logical structure of the code. Typically, programmers use indentation and other white space to show the logical structure.

Consistently represent the logical structure of the code

Some styles of layout have rules with so many exceptions that it's hard to follow the rules consistently. A good style applies to most cases.

295
296
297
298
299

300
301
302

303
304

305
306 **KEY POINT**
307
308

309
310
311
312
313

314

315
316

317
318
319
320

321 **CROSS-REFERENCE** Some researchers have explored the similarity between the structure of a book and the structure of a program. For information, see “The Book Paradigm for Program Documentation” in Section 32.5.

Improve readability

An indentation strategy that's logical but that makes the code harder to read is useless. A layout scheme that calls for spaces only where they are required by the compiler is logical but not readable. A good layout scheme makes code easier to read.

Withstand modifications

The best layout schemes hold up well under code modification. Modifying one line of code shouldn't require modifying several others.

In addition to these criteria, minimizing the number of lines of code needed to implement a simple statement or block is also sometimes considered.

How to Put the Layout Objectives to Use

You can use the criteria for a good layout scheme to ground a discussion of layout so that the subjective reasons for preferring one style over another are brought into the open.

Weighting the criteria in different ways might lead to different conclusions. For example, if you feel strongly that minimizing the number of lines used on the screen is important—perhaps because you have a small computer screen—you might criticize one style because it uses two more lines for a routine parameter list than another.

31.2 Layout Techniques

You can achieve good layout by using a few layout tools in several different ways. This section describes each of them.

White Space

Use white space to enhance readability. White space, including spaces, tabs, line breaks, and blank lines, is the main tool available to you for showing a program's structure.

You wouldn't think of writing a book with no spaces between words, no paragraph breaks, and no divisions into chapters. Such a book might be readable cover to cover, but it would be virtually impossible to skim it for a line of thought or to find an important passage. Perhaps more important, the book's layout wouldn't show the reader how the author intended to organize the information. The author's organization is an important clue to the topic's logical organization.

328 Breaking a book into chapters, paragraphs, and sentences shows a reader how to
329 mentally organize a topic. If the organization isn't evident, the reader has to pro-
330 vide the organization, which puts a much greater burden on the reader and adds
331 the possibility that the reader may never figure out how the topic is organized.

332 The information contained in a program is denser than the information contained
333 in most books. Whereas you might read and understand a page of a book in a
334 minute or two, most programmers can't read and understand a naked program
335 listing at anything close to that rate. A program should give more organizational
336 clues than a book, not fewer.

Grouping

337 From the other side of the looking glass, white space is grouping, making sure
338 that related statements are grouped together.

340 In writing, thoughts are grouped into paragraphs. A well-written paragraph con-
341 tains only sentences that relate to a particular thought. It shouldn't contain extra-
342 neous sentences. Similarly, a paragraph of code should contain statements that
343 accomplish a single task and that are related to each other.

Blank lines

344 Just as it's important to group related statements, it's important to separate unre-
345 lated statements from each other. The start of a new paragraph in English is iden-
346 tified with indentation or a blank line. The start of a new paragraph of code
347 should be identified with a blank line.

349 Using blank lines is a way to indicate how a program is organized. You can use
350 them to divide groups of related statements into paragraphs, to separate routines
351 from one another, and to highlight comments.

HARD DATA

352 Although this particular statistic may be hard to put to work, a study by Gorla,
353 Benander, and Benander found that the optimal number of blank lines in a pro-
354 gram is about 8 to 16 percent. Above 16 percent, debug time increases dramati-
355 cally (1990).

Indentation

356 Use indentation to show the logical structure of a program. As a rule, you should
357 indent statements under the statement to which they are logically subordinate.

HARD DATA

359 Indentation has been shown to be correlated with increased programmer com-
360 prehension. The article "Program Indentation and Comprehensibility" reported
361 that several studies found correlations between indentation and improved com-
362 prehension (Miarra et al. 1983). Subjects scored 20 to 30 percent higher on a test
363 of comprehension when programs had a two-to-four-spaces indentation scheme
364 than they did when programs had no indentation at all.

365 | HARD DATA366
367
368
369
370
371
372
373
374

The same study found that it was important to neither under-emphasize nor over-emphasize a program's logical structure. The lowest comprehension scores were achieved on programs that were not indented at all. The second lowest were achieved on programs that used six-space indentation. The study concluded that two-to-four-space indentation was optimal. Interestingly, many subjects in the experiment felt that the six-space indentation was easier to use than the smaller indentations, even though their scores were lower. That's probably because six-space indentation looks pleasing. But regardless of how pretty it looks, six-space indentation turns out to be less readable. This is an example of a collision between aesthetic appeal and readability.

375

Parentheses

Use more parentheses than you think you need. Use parentheses to clarify expressions that involve more than two terms. They may not be needed, but they add clarity and they don't cost you anything. For example, how are the following expressions evaluated?

380

C++ Version: 12 + 4 % 3 * 7 / 8

381

Visual Basic Version: 12 + 4 mod 3 * 7 \ 8

382
383
384
385
386

The key question is, did you have to think about how the expressions are evaluated? Can you be confident in your answer without checking some references? Even experienced programmers don't answer confidently, and that's why you should use parentheses whenever there is any doubt about how an expression is evaluated.

387

31.3 Layout Styles

388
389
390
391
392
393
394

Most layout issues have to do with laying out blocks, the groups of statements below control statements. A block is enclosed between braces or keywords: *{* and *}* in C++ and Java; *if-then-endif* in Visual Basic; and other similar structures in other languages. For simplicity, much of this discussion uses *begin* and *end* generically, assuming that you can figure out how the discussion applies to braces in C++ and Java or other blocking mechanisms in other languages. The following sections describe four general styles of layout:

395
396
397
398

- Pure blocks
- Emulating pure blocks
- using *begin-end* pairs (braces) to designate block boundaries
- Endline layout

399

Pure Blocks

400 Much of the layout controversy stems from the inherent awkwardness of the
401 more popular programming languages. A well-designed language has clear block
402 structures that lend themselves to a natural indentation style. In Visual Basic, for
403 example, each control construct has its own terminator, and you can't use a con-
404 trol construct without using the terminator. Code is blocked naturally. Some ex-
405 amples in Visual Basic are shown in Listing 31-6, Listing 31-7, and Listing 31-8:

406 **Listing 31-6. Visual Basic example of a pure *if* block.**

```
407 If pixelColor = Color_Red Then  
408     statement1  
409     statement2  
410     ...  
411 End If
```

412 **Listing 31-7. Visual Basic example of a pure *while* block.**

```
413 While pixelColor = Color_Red  
414     statement1  
415     statement2  
416     ...  
417 Wend
```

418 **Listing 31-8. Visual Basic example of a pure *case* block.**

```
419 Select Case pixelColor  
420     Case Color_Red  
421         statement1  
422         statement2  
423         ...  
424     Case Color_Green  
425         statement1  
426         statement2  
427         ...  
428     Case Else  
429         statement1  
430         statement2  
431         ...  
432 End Select
```

433 A control construct in Visual Basic always has a beginning statement—*If-Then*,
434 *While*, and *Select-Case* in the examples—and it always has a corresponding *End*
435 statement. Indenting the inside of the structure isn't a controversial practice, and
436 the options for aligning the other keywords are somewhat limited. Listing 31-9 is
437 an abstract representation of how this kind of formatting works:

438 **Listing 31-9. Abstract example of the pure-block layout style.**

```
A [REDACTED]  
B [REDACTED]
```

441
442
443
444
445

C [REDACTED]
D [REDACTED]

In this example, statement A begins the control construct and statement D ends the control construct. The alignment between the two provides solid visual closure.

446
447
448
449
450
451
452
453
454
455

The controversy about formatting control structures arises in part from the fact that some languages don't *require* block structures. You can have an *if-then* followed by a single statement and not have a formal block. You have to add a *begin-end* pair or opening and closing braces to create a block rather than getting one automatically with each control construct. Uncoupling *begin* and *end* from the control structure—as languages like C++ and Java do with { and }—leads to questions about where to put the *begin* and *end*. Consequently, many indentation problems are problems only because you have to compensate for poorly designed language structures. Various ways to compensate are described in the following sections.

456

Emulating Pure Blocks

457
458
459
460
461

A good approach in languages that don't have pure blocks is to view the *begin* and *end* keywords (or { and } tokens) as extensions of the control construct they're used with. Then it's sensible to try to emulate the Visual Basic formatting in your language. Listing 31-10 is an abstract view of the visual structure you're trying to emulate:

462

Listing 31-10. Abstract example of the pure-block layout style.

463
464
465
466

A [REDACTED]
B [REDACTED]
C [REDACTED]
D [REDACTED]

467
468
469
470

In this style, the control structure opens the block in statement A and finishes the block in statement D. This implies that the *begin* should be at the end of statement A and the *end* should be statement D. In the abstract, to emulate pure blocks, you'd have to do something like Listing 31-11:

471

Listing 31-11. Abstract example of emulating the pure-block style.

472
473
474
475

A [REDACTED] {
B [REDACTED]
C [REDACTED]
D [REDACTED]}

476
477

Some examples of how the style looks in C++ are shown in Listing 31-12, Listing 31-13, and Listing 31-14:

478

Listing 31-12. C++ example of emulating a pure *if* block.

479

```
if ( pixelColor == Color_Red ) {
```

```
480     statement1;  
481     statement2;  
482     ...  
483 }
```

Listing 31-13. C++ example of emulating a pure *while* block.

```
484 while ( pixelColor == Color_Red ) {  
485     statement1;  
486     statement2;  
487     ...  
488 }  
489
```

Listing 31-14. C++ example of emulating a pure *switch/case* block.

```
490 switch ( pixelColor ) {  
491     case Color_Red:  
492         statement1;  
493         statement2;  
494         ...  
495         break;  
496     case Color_Green:  
497         statement1;  
498         statement2;  
499         ...  
500         break;  
501     default:  
502         statement1;  
503         statement2;  
504         ...  
505         break;  
506 }  
507
```

This style of alignment works pretty well. It looks good, you can apply it consistently, and it's maintainable. It supports the Fundamental Theorem of Formatting in that it helps to show the logical structure of the code. It's a reasonable style choice. This style is standard in Java and common in C++.

Using *begin-end* pairs (braces) to Designate Block Boundaries

A substitute for a pure block structure is to view *begin-end* pairs as block boundaries. If you take that approach, you view the *begin* and the *end* as statements that follow the control construct rather than as fragments that are part of it.

Graphically, this is the ideal, just as it was with the pure-block emulation shown again in Listing 31-15:

Listing 31-15. Abstract example of the pure-block layout style.

520
521
522
523
524
525
526
527
528

A [REDACTED]
B [REDACTED]
C [REDACTED]
D [REDACTED]

But in this style, to treat the *begin* and the *end* as parts of the block structure rather than the control statement, you have to put the *begin* at the beginning of the block (rather than at the end of the control statement) and the *end* at the end of the block (rather than terminating the control statement). In the abstract, you'll have to do something like Listing 31-16.

529

530

531

532

533

534

535

536

537

Listing 31-16. Abstract example of using *begin* and *end* as block boundaries.

A [REDACTED]
{ [REDACTED]
B [REDACTED]
C [REDACTED]
}

Some examples of how using *begin* and *end* as block boundaries looks in C++ are shown in Listing 31-17, Listing 31-18, and Listing 31-19:

538

539

540

541

542

543

544

545

Listing 31-17. C++ example of using *begin* and *end* as block boundaries in an *if* block.

```
if ( pixelColor == Color_Red )
{
    statement1;
    statement2;
    ...
}
```

546

547

Listing 31-18. C++ example of using *begin* and *end* as block boundaries in a *while* block.

```
while ( pixelColor == Color_Red )
{
    statement1;
    statement2;
    ...
}
```

554

555

556

557

558

559

560

561

562

Listing 31-19. C++ example of using *begin* and *end* as block boundaries in a *switch/case* block.

```
switch ( pixelColor )
{
    case Color_Red:
        statement1;
        statement2;
        ...
        break;
```

```
563     case Color_Green:  
564         statement1;  
565         statement2;  
566         ...  
567         break;  
568     default:  
569         statement1;  
570         statement2;  
571         ...  
572         break;  
573 }
```

This alignment style works well. It supports the Fundamental Theorem of Formatting by exposing the code's underlying logical structure. Its only limitation is that it can't be applied literally in *switch/case* statements in C++ and Java, as shown by Listing 31-19. (The *break* keyword is a substitute for the closing brace, but there is no equivalent to the opening brace.)

Endline Layout

Another layout strategy is “endline layout,” which refers to a large group of layout strategies in which the code is indented to the middle or end of the line. The endline indentation is used to align a block with the keyword that began it, to make a routine’s subsequent parameters line up under its first parameter, to line up cases in a *case* statement, and for other similar purposes. Listing 31-20 is an abstract example:

Listing 31-20. Abstract example of the endline layout style.

```
586 A [REDACTED]  
587 B [REDACTED]  
588 C [REDACTED]  
589 D [REDACTED]
```

In this example, statement A begins the control construct and statement D ends it. Statements B, C, and D are aligned under the keyword that began the block in statement A. The uniform indentation of B, C, and D shows that they’re grouped together. Listing 31-21 is a less abstract example of code formatted using this strategy:

Listing 31-21. Visual Basic example of endline layout of a while block.

```
596 While ( pixelColor = Color_Red )  
597     statement1;  
598     statement2;  
599     ...  
600 Wend
```

602
603
604

In the example, the *begin* is placed at the end of the line rather than under the corresponding keyword. Some people prefer to put *begin* under the keyword, but choosing between those two fine points is the least of this style's problems.

605
606

The endline layout style works acceptably in a few cases. Listing 31-22 is an example in which it works:

607
608
609
610
611

Listing 31-22. A rare Visual Basic example in which endline layout seems appealing.

612 *The else keyword is aligned with the then keyword above it.*
613 *with the then keyword above it.*
614
615
616

```
If ( soldCount > 1000 ) Then
    markdown = 0.10
    profit = 0.05
Else
    markdown = 0.05
End If
```

In this case, the *Then*, *Else*, and *End If* keywords are aligned, and the code following them is also aligned. The visual effect is a clear logical structure.

617
618
619
620
621

If you look critically at the earlier *case*-statement example, you can probably predict the unraveling of this style. As the conditional expression becomes more complicated, the style will give useless or misleading clues about the logical structure. Listing 31-23 is an example of how the style breaks down when it's used with a more complicated conditional:

622
623
624 **CODING HORROR**

Listing 31-23. A more typical Visual Basic example, in which endline layout breaks down.

```
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
```

```
If ( soldCount > 10 And prevMonthSales > 10 ) Then
    If ( soldCount > 100 And prevMonthSales > 10 ) Then
        If ( soldCount > 1000 ) Then
            markdown = 0.1
            profit = 0.05
        Else
            markdown = 0.05
        End If
    Else
        markdown = 0.025
    End If
Else
    markdown = 0.0
End If
```

What's the reason for the bizarre formatting of the *Else* clauses at the end of the example? They're consistently indented under the corresponding keywords, but it's hard to argue that their indentations clarify the logical structure. And if the code were modified so that the length of the first line changed, the endline style would require that the indentation of corresponding statements be changed. This

643 poses a maintenance problem that pure block, pure-block emulation, and using
644 begin-end to designate block boundaries do not.

645 You might think that these examples are contrived just to make a point, but this
646 style has been persistent despite its drawbacks. Numerous textbooks and pro-
647 gramming references have recommended this style. The earliest book I saw that
648 recommended this style was published in the mid-1970s and the most recent was
649 published in 2003.

650 Overall, endline layout is inaccurate, hard to apply consistently, and hard to
651 maintain. You'll see other problems with endline layout throughout the chapter.

652 **Which Style Is Best?**

653 If you're working in Visual Basic, use pure-block indentation. (The Visual Basic
654 IDE makes it hard not to use this style anyway.)

655 In Java, standard practice is to use pure-block indentation.

656 In C++, you might simply choose the style you like or the one that is preferred
657 by the majority of people on your team. Either pure-block emulation or *begin-*
658 *end* block boundaries work equally well. The only study that has compared the
659 two styles found no statistically significant difference between the two as far as
660 understandability is concerned (Hansen and Yim 1987).

661 Neither of the styles is foolproof, and each requires an occasional "reasonable
662 and obvious" compromise. You might prefer one or the other for aesthetic rea-
663 sons. This book uses pure block style in its code examples, so you can see many
664 more illustrations of how that style works just by skimming through the exam-
665 ples. Once you've chosen a style, you reap the most benefit from good layout
666 when you apply it consistently.

667 **31.4 Laying Out Control Structures**

668 **CROSS-REFERENCE** For
669 details on documenting con-
670 trol structures, see "Com-
671 menting Control Structures"
in Section 32.5. For a dis-
672 cussion of other aspects of con-
673 trol structures, see Chapters
14 through 19.

The layout of some program elements is primarily a matter of aesthetics. Layout
of control structures, however, affects readability and comprehensibility and is
therefore a practical priority.

674 **Fine Points of Formatting Control-Structure Blocks**

Working with control-structure blocks requires attention to some fine details.
Here are some guidelines:

674 **Avoid unindented begin-end pairs**

675 In the style shown in Listing 31-24, the *begin-end* pair is aligned with the control
676 structure, and the statements that *begin* and *end* enclose are indented under *be-*
677 *gin*.

678 **Listing 31-24. Java example of unindented begin-end pairs.**

```
679       for ( int i = 0; i < MAX_LINES; i++ )  
680       {  
681           ReadLine( i );  
682           ProcessLine( i );  
683       }
```

684 Although this approach looks fine, it violates the Fundamental Theorem of For-
685 matting; it doesn't show the logical structure of the code. Used this way, the
686 *begin* and *end* aren't part of the control construct, but they aren't part of the
687 statement(s) after it either.

688 Listing 31-25 is an abstract view of this approach:

689 **Listing 31-25. Abstract example of misleading indentation.**

```
690       A [REDACTED]  
691       B [REDACTED]  
692       C [REDACTED]  
693       D [REDACTED]  
694       E [REDACTED]
```

695 In this example, is statement B subordinate to statement A? It doesn't look like
696 part of statement A, and it doesn't look as if it's subordinate to it either. If you
697 have used this approach, change to one of the two layout styles described earlier,
698 and your formatting will be more consistent.

699 **Avoid double indentation with begin and end**

700 A corollary to the rule against nonindented *begin-end* pairs is the rule against
701 doubly indented *begin-end* pairs. In this style, shown in Listing 31-26, *begin* and
702 *end* are indented and the statements they enclose are indented again:

703 **Listing 31-26. Java example of inappropriate double indentation of**
704 ***begin-end* block.**

```
705       CODING HORROR  
706       for ( int i = 0; i < MAX_LINES; i++ )  
707       {  
708           ReadLine( i );  
709           ProcessLine( i );  
710       }
```

711 This is another example of a style that looks fine but violates the Fundamental
712 Theorem of Formatting. One study showed no difference in comprehension be-
713 tween programs that are singly indented and programs that are doubly indented
(Miaria et al. 1983), but this style doesn't accurately show the logical structure

714 of the program; *ReadLine()* and *ProcessLine()* are shown as if they are logically
715 subordinate to the *begin-end* pair, and they aren't.

716 The approach also exaggerates the complexity of a program's logical structure.
717 Which of the structures shown in Listing 31-27 and Listing 31-28 looks more
718 complicated?

719 **Listing 31-27. Abstract Structure 1.**

720 [REDACTED]

721 [REDACTED]

722 [REDACTED]

723 [REDACTED]

724 [REDACTED]

725 **Listing 31-28. Abstract Structure 2.**

726 [REDACTED]

727 [REDACTED]

728 [REDACTED]

729 [REDACTED]

730 [REDACTED]

731 Both are abstract representations of the structure of the *for* loop. Abstract Struc-
732 ture 1 looks more complicated even though it represents the same code as Ab-
733 stract Structure 2. If you were to nest statements to two or three levels, double
734 indentation would give you four or six levels of indentation. The layout that re-
735 sulted would look more complicated than the actual code would be. Avoid the
736 problem by using pure-block emulation or by using *begin* and *end* as block
737 boundaries and aligning *begin* and *end* with the statements they enclose.

738

Other Considerations

739 Although indentation of blocks is the major issue in formatting control struc-
740 tures, you'll run into a few other kinds of issues. Here are some more guidelines:

741 **Use blank lines between paragraphs**

742 Some blocks of code aren't demarcated with *begin-end* pairs. A logical block—a
743 group of statements that belong together—should be treated the way paragraphs
744 in English are. Separate them from each other with blank lines. Listing 31-29
745 shows an example of paragraphs that should be separated.

746 **Listing 31-29. C++ example of code that should be grouped and sepa-
747 rated.**

748 cursor.start = startingScanLine;
749 cursor.end = endingScanLine;
750 window.title = editWindow.title;
751 window.dimensions = editWindow.dimensions;
752 window.foregroundColor = userPreferences.foregroundColor;

753
754
755
756
757 **CROSS-REFERENCE** If you use the Pseudocode Programming Process, your blocks of code will be separated automatically. For details, see Chapter 9, "The Pseudocode Programming Process."
764
765

766
767
768 These lines set up a text window.
769
770
771
772
773 These lines set up a cursor and should be separated from the preceding lines.
774
775
776
777
778
779
780

781
782
783
784
785
786
787
788

789
790
791 Style 1
792
793

```
cursor.blinkRate      = editMode.blinkRate;
window.backgroundColor = userPreferences.backgroundColor;
SaveCursor( cursor );
SetCursor( cursor );
```

This code looks all right, but blank lines would improve it in two ways. First, when you have a group of statements that don't have to be executed in any particular order, it's tempting to lump them all together this way. You don't need to further refine the statement order for the computer, but human readers appreciate more clues about which statements need to be performed in a specific order and which statements are just along for the ride. The discipline of putting blank lines throughout a program makes you think harder about which statements really belong together. The revised fragment in Listing 31-30 shows how this collection should really be organized.

Listing 31-30. C++ example of code that is appropriately grouped and separated.

```
window.dimensions = editWindow.dimensions;
window.title = editWindow.title;
window.backgroundColor = userPreferences.backgroundColor;
window.foregroundColor = userPreferences.foregroundColor;

cursor.start = startingScanLine;
cursor.end = endingScanLine;
cursor.blinkRate = editMode.blinkRate;
SaveCursor( cursor );
SetCursor( cursor );
```

The reorganized code shows that two things are happening. In the first example, the lack of statement organization and blank lines, and the old aligned-equals-signs trick, make the statements look more related than they are.

The second way in which using blank lines tends to improve code is that it opens up natural spaces for comments. In the code above, a comment above each block would nicely supplement the improved layout.

Format single-statement blocks consistently

A single-statement block is a single statement following a control structure, such as one statement following an *if* test. In such a case, *begin* and *end* aren't needed for correct compilation and you have the three style options shown in Listing 31-31.

Listing 31-31. Java example of style options for single-statement blocks.

```
if ( expression )
    one-statement;
```

```
794     Style 2a | if ( expression ) {  
795         one-statement;  
796     }  
797  
798     Style 2b | if ( expression )  
799     {  
800         one-statement;  
801     }  
802  
803     Style 3 | if ( expression ) one-statement;
```

There are arguments in favor of each of these approaches. Style 1 follows the indentation scheme used with blocks, so it's consistent with other approaches. Style 2 (either 2a or 2b) is also consistent, and the *begin-end* pair reduces the chance that you'll add statements after the *if* test and forget to add *begin* and *end*. This would be a particularly subtle error because the indentation would tell you that everything is OK, but the indentation wouldn't be interpreted the same way by the compiler. Style 3's main advantage over Style 2 is that it's easier to type. Its advantage over Style 1 is that if it's copied to another place in the program, it's more likely to be copied correctly. Its disadvantage is that in a line-oriented debugger, the debugger treats the line as one line and the debugger doesn't show you whether it executes the statement after the *if* test.

I've used Style 1 and have been the victim of incorrect modification many times. I don't like the exception to the indentation strategy caused by Style 3, so I avoid it altogether. On a group project, I favor either variation of Style 2 for its consistency and safe modifiability. Regardless of the style you choose, use it consistently and use the same style for *if* tests and all loops.

For complicated expressions, put separate conditions on separate lines
Put each part of a complicated expression on its own line. Listing 31-32 shows an expression that's formatted without any attention to readability:

Listing 31-32. Java example of an essentially unformatted (and unreadable) complicated expression.

```
825     if (((('0' <= inChar) && (inChar <= '9')) || ((('a' <= inChar) &&  
826         (inChar <= 'z')) || ((('A' <= inChar) && (inChar <= 'Z'))))  
827     ...
```

This is an example of formatting for the computer instead of for human readers. By breaking the expression into several lines, as in Listing 31-33, you can improve readability.

Listing 31-33. Java example of a readable complicated expression.

832 **CROSS-REFERENCE** An
 833 other technique for making
 834 complicated expressions
 835 readable is to put them into
 836 boolean functions. For details
 837 on putting complicated ex-
 838 pressions into boolean func-
 839 tions and other readability
 840 techniques, see Section 19.1,
 841 “Boolean Expressions.”

842 **CROSS-REFERENCE** For
 843 details on the use of *gos*,
 844 see in Section 17.3, “*g*oto.”
 845
 846
 847
 848
 849

850 **Goto labels should be
 851 left-aligned in all caps
 852 and should include the
 853 programmer’s name,
 854 home phone number, and
 credit card number.**

855 —Abdul Nizar

856
 857

858

859 **CROSS-REFERENCE** For
 860 other methods of addressing
 861 this problem, see “Error
 Processing and *gos*” in
 862 Section 17.3.
 863

864
 865
 866
 867
 868
 869
 870
 871

```
if ( ( ( '0' <= inChar ) && ( inChar <= '9' ) ) ||
    ( ( 'a' <= inChar ) && ( inChar <= 'z' ) ) ||
    ( ( 'A' <= inChar ) && ( inChar <= 'Z' ) ) )
    ...

```

The second fragment uses several formatting techniques—indentation, spacing, number-line ordering, and making each incomplete line obvious—and the result is a readable expression. Moreover, the intent of the test is clear. If the expression contained a minor error, such as using a *z* instead of a *Z*, it would be obvious in code formatted this way, whereas the error wouldn’t be clear with less careful formatting.

Avoid *gos*

The original reason to avoid *gos* was that they made it difficult to prove that a program was correct. That’s a nice argument for all the people who want to prove their programs correct, which is practically no one. The more pressing problem for most programmers is that *gos* make code hard to format. Do you indent all the code between the *g*oto and the label it goes to? What if you have several *gos* to the same label? Do you indent each new one under the previous one? Here’s some advice for formatting *gos*:

- Avoid *gos*. This sidesteps the formatting problem altogether.
- Use a name in all caps for the label the code goes to. This makes the label obvious.
- Put the statement containing the *g*oto on a line by itself. This makes the *g*oto obvious.
- Put the label the *g*oto goes to on a line by itself. Surround it with blank lines. This makes the label obvious. Outdent the line containing the label to the left margin to make the label as obvious as possible.

Listing 31-34 shows these *g*oto layout conventions at work.

Listing 31-34. C++ example of making the best of a bad situation (using *g*oto).

```
void PurgeFiles( ErrorCode & errorCode ) {
    fileList fileList;
    int numFilesToPurge = 0;
    MakePurgeFileList( fileList, numFilesToPurge );

    errorCode = FileError_Success;
    int fileIndex = 0;
    while ( fileIndex < numFilesToPurge ) {
        DataFile fileToPurge;
        if ( !FindFile( fileList[ fileIndex ], fileToPurge ) ) {
            errorCode = FileError_NotFound;
        }
    }
}
```

```
872     Here's a goto.      goto END_PROC;
873
874
875         if ( !OpenFile( fileToPurge ) ) {
876             errorCode = FileError_NotOpen;
877             goto END_PROC;
878         }
879
880         if ( !OverwriteFile( fileToPurge ) ) {
881             errorCode = FileError_CantOverwrite;
882             goto END_PROC;
883         }
884
885         if ( !Erase( fileToPurge ) ) {
886             errorCode = FileError_CantErase;
887             goto END_PROC;
888         }
889         fileIndex++;
890     }
891
892     Here's the goto label. The
893 intent of the capitalization and
894 layout is to make the label
895 hard to miss.
```

CROSS-REFERENCE For details on using *case* statements, see Section 15.2, “*case* Statements.”

The C++ example in Listing 31-34 is relatively long so that you can see a case in which an expert programmer might conscientiously decide that a *goto* is the best design choice. In such a case, the formatting shown is about the best you can do.

No endlne exception for case statements

One of the hazards of endlne layout comes up in the formatting of *case* statements. A popular style of formatting *cases* is to indent them to the right of the description of each case, as shown in Listing 31-35. The big problem with this style is that it's a maintenance headache.

Listing 31-35. C++ example of hard-to-maintain endlne layout of a case statement.

```
904
905
906 switch ( ballColor ) {
907     case BallColor_Blue:           Rollout();
908                             break;
909     case BallColor_Orange:        SpinOnFinger();
910                             break;
911     case BallColor_FluorescentGreen: Spike();
912                             break;
913     case BallColor_White:         KnockCoverOff();
914                             break;
915     case BallColor_WhiteAndBlue:   if ( mainColor == BallColor_White ) {
```

```
916                                     KnockCoverOff();
917                                 }
918             else if ( mainColor == BallColor_Blue ) {
919                 RollOut();
920             }
921             break;
922         default:
923             FatalError( "Unrecognized kind of ball." );
924             break;
925     }
```

If you add a case with a longer name than any of the existing names, you have to shift out all the cases and the code that goes with them. The large initial indentation makes it awkward to accommodate any more logic, as shown in the *WhiteAndBlue* case. The solution is to switch to your standard indentation increment. If you indent statements in a loop three spaces, indent cases in a *case* statement the same number of spaces, as in Listing 31-36:

Listing 31-36. C++ example of good standard indentation of a *case* statement.

```
931 switch ( ballColor ) {
932     case BallColor_Blue:
933         Rollout();
934         break;
935     case BallColor_Orange:
936         SpinOnFinger();
937         break;
938     case BallColor_FluorescentGreen:
939         Spike();
940         break;
941     case BallColor_White:
942         KnockCoverOff();
943         break;
944     case BallColor_WhiteAndBlue:
945         if ( mainColor = BallColor_White ) {
946             KnockCoverOff();
947         }
948         else if ( mainColor = BallColor_Blue ) {
949             RollOut();
950         }
951         break;
952     default:
953         FatalError( "Unrecognized kind of ball." );
954         break;
955     }
956 }
```

This is an instance in which many people might prefer the looks of the first example. For the ability to accommodate longer lines, consistency, and maintainability, however, the second approach wins hands down.

961
962
963
964
965

If you have a *case* statement in which all the cases are exactly parallel and all the actions are short, you could consider putting the case and action on the same line. In most instances, however, you'll live to regret it. The formatting is a pain initially and breaks under modification, and it's hard to keep the structure of all the cases parallel as some of the short actions become longer ones.

966

967 **CROSS-REFERENCE** For details on documenting individual statements, see
968 "Commenting Individual
969 Lines" in Section 32.5.
970

971

972

973

974

975

976

977

978

979

980

981

982

983

984

985

986

987

988

989

990

31.5 Laying Out Individual Statements

This section explains many ways to improve individual statements in a program.

Statement Length

A common rule is to limit statement line length to 80 characters. Here are the reasons:

- Lines longer than 80 characters are hard to read.
- The 80-character limitation discourages deep nesting.
- Lines longer than 80 characters often won't fit on 8.5" x 11" paper.
- Paper larger than 8.5" x 11" is hard to file.

With larger screens, narrow typefaces, laser printers, and landscape mode, the arguments for the 80-character limit aren't as compelling as they used to be. A single 90-character-long line is usually more readable than one that has been broken in two just to avoid spilling over the 80th column. With modern technology, it's probably all right to exceed 80 columns occasionally.

Using Spaces for Clarity

Add white space within a statement for the sake of readability:

Use spaces to make logical expressions readable

The expression

```
while(pathName[startPath+position]<> ';' ) and  
      ((startPath+position)<length(pathName)) do  
is about as readable as Idareyoutoreadthis.
```

As a rule, you should separate identifiers from other identifiers with spaces. If you use this rule, the *while* expression looks like this:

```
while ( pathName[ startPath+position ] <> ';' ) and  
      ( ( startPath + position ) < length( pathName ) ) do
```

991 Some software artists might recommend enhancing this particular expression
992 with additional spaces to emphasize its logical structure, this way:

993 while (pathName[startPath + position] < ';') and
994 ((startPath + position) < length(pathName)) do

995 This is fine, although the first use of spaces was sufficient to ensure readability.
996 Extra spaces hardly ever hurt, however, so be generous with them.

997 ***Use spaces to make array references readable***

998 The expression

999 grossRate[census[groupId].gender, census[groupId].ageGroup]
1000 is no more readable than the earlier dense *while* expression. Use spaces around
1001 each index in the array to make the indexes readable. If you use this rule, the
1002 expression looks like this:

1003 grossRate[census[groupId].gender, census[groupId].ageGroup]

1004 ***Use spaces to make routine arguments readable***

1005 What is the fourth argument to the following routine?

1006 ReadEmployeeData(maxEmps, empData, inputFile, empCount, inputError);
1007 Now, what is the fourth argument to the following routine?

1008 GetCensus(inputFile, empCount, empData, maxEmps, inputError);

1009 Which one was easier to find? This is a realistic, worthwhile question because
1010 argument positions are significant in all major procedural languages. It's com-
1011 mon to have a routine specification on one half of your screen and the call to the
1012 routine on the other half, and to compare each formal parameter with each actual
1013 parameter.

1014 **Formatting Continuation Lines**

1015 One of the most vexing problems of program layout is deciding what to do with
1016 the part of a statement that spills over to the next line. Do you indent it by the
1017 normal indentation amount? Do you align it under the keyword? What about
1018 assignments?

1019 Here's a sensible, consistent approach that's particularly useful in Java, C, C++,
1020 Visual Basic, and other languages that encourage long variable names.

1021 ***Make the incompleteness of a statement obvious***

1022 Sometimes a statement must be broken across lines, either because it's longer
1023 than programming standards allow or because it's too absurdly long to put on
1024 one line. Make it obvious that the part of the statement on the first line is only

1026
1027
1028

part of a statement. The easiest way to do that is to break up the statement so that the part on the first line is blatantly incorrect syntactically if it stands alone.

Some examples are shown in Listing 31-37:

1029
1030 *The && signals that the
1031 statement isn't complete.*

1032
1033
1034 *The plus sign (+) signals that
1035 the statement isn't complete.*

1036
1037
1038 *The comma (,) signals that
1039 the statement isn't complete.*

1040
1041
1042
1043
1044

Listing 31-37. Java examples of obviously incomplete statements.

```
while ( pathName[ startPath + position ] != ';' ) &&  
    ( ( startPath + position ) <= pathName.length() )  
    ...  
  
totalBill = totalBill + customerPurchases[ customerID ] +  
    SalesTax( customerPurchases[ customerID ] );  
    ...  
  
DrawLine( window.north, window.south, window.east, window.west,  
    currentWidth, currentAttribute );  
    ...
```

In addition to telling the reader that the statement isn't complete on the first line, the break helps prevent incorrect modifications. If the continuation of the statement were deleted, the first line wouldn't look as if you had merely forgotten a parenthesis or semicolon—it would clearly need something more.

1045
1046
1047
1048

Keep closely related elements together

When you break a line, keep things together that belong together—array references, arguments to a routine, and so on. The example shown in Listing 31-38 is poor form:

1049
1050 **CODING HORROR**
1051
1052
1053
1054
1055
1056

Listing 31-38. Java example of breaking a line poorly.

```
customerBill = PreviousBalance( paymentHistory[ customerID ] ) + LateCharge(  
    paymentHistory[ customerID ] );
```

Admittedly, this line break follows the guideline of making the incompleteness of the statement obvious, but it does so in a way that makes the statement unnecessarily hard to read. You might find a case in which the break is necessary, but in this case it isn't. It's better to keep the array references all on one line. Listing 31-39 shows better formatting:

1057
1058
1059

Listing 31-39. Java example of breaking a line well.

```
customerBill = PreviousBalance( paymentHistory[ customerID ] ) +  
    LateCharge( paymentHistory[ customerID ] );
```

1060
1061
1062
1063

Indent routine-call continuation lines the standard amount

If you normally indent three spaces for statements in a loop or a conditional, indent the continuation lines for a routine by three spaces. Some examples are shown in Listing 31-40:

1064
1065
1066
1067
1068
1069
1070
1071
1072

Listing 31-40. Java examples of indenting routine-call continuation lines using the standard indentation increment.

```
DrawLine( window.north, window.south, window.east, window.west,  
         currentWidth, currentAttribute );  
SetFontAttributes( faceName[ fontId ], size[ fontId ], bold[ fontId ],  
                   italic[ fontId ], syntheticAttribute[ fontId ].underline,  
                   syntheticAttribute[ fontId ].strikeout );
```

One alternative to this approach is to line up the continuation lines under the first argument to the routine, as shown in Listing 31-41:

1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083

Listing 31-41. Java examples of indenting a routine-call continuation line to emphasize routine names.

```
DrawLine( window.north, window.south, window.east, window.west,  
         currentWidth, currentAttribute );  
SetFontAttributes( faceName[ fontId ], size[ fontId ], bold[ fontId ],  
                   italic[ fontId ], syntheticAttribute[ fontId ].underline,  
                   syntheticAttribute[ fontId ].strikeout );
```

From an aesthetic point of view, this looks a little ragged compared to the first approach. It is also difficult to maintain as routine names changes, argument names change, and so on. Most programmers tend to gravitate toward the first style over time.

1084
1085
1086
1087
1088

Make it easy to find the end of a continuation line

One problem with the approach shown above is that you can't easily find the end of each line. Another alternative is to put each argument on a line of its own and indicate the end of the group with a closing parenthesis. Listing 31-42 shows how it looks.

1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106

Listing 31-42. Java examples of formatting routine-call continuation lines one argument to a line.

```
DrawLine(  
         window.north,  
         window.south,  
         window.east,  
         window.west,  
         currentWidth,  
         currentAttribute  
);  
  
SetFontAttributes(  
                 faceName[ fontId ],  
                 size[ fontId ],  
                 bold[ fontId ],  
                 italic[ fontId ],  
                 syntheticAttribute[ fontId ].underline,  
                 syntheticAttribute[ fontId ].strikeout
```

1107
1108
1109
1110
1111
1112

) ;
This approach takes up a lot of real estate. If the arguments to a routine are long object-field references or pointer names, however, as the last two are, using one argument per line improves readability substantially. The); at the end of the block makes the end of the call clear. You also don't have to reformat when you add a parameter; you just add a new line.

1113
1114
1115

In practice, usually only a few routines need to be broken into multiple lines. You can handle others on one line. Any of the three options for formatting multiple-line routine calls works all right if you use it consistently.

1116
1117
1118
1119

Indent control-statement continuation lines the standard amount
If you run out of room for a *for* loop, a *while* loop, or an *if* statement, indent the continuation line by the same amount of space that you indent statements in a loop or after an *if* statement. Two examples are shown in Listing 31-43:

1120
1121

Listing 31-43. Java examples of indenting control-statement continuation lines.

1122
1123 *This continuation line is*
1124 *indented the standard number*
1125 *of spaces...*
1126
1127
1128 *...as is this one.*
1129
1130
1131

1132 **CROSS-REFERENCE** Sometimes the best solution to a
1133 complicated test is to put it
1134 into a boolean function. For
1135 examples, see "Making
1136 Complicated Expressions
1137 Simple" in Section 19.1.
1138

```
while ( ( pathName[ startPath + position ] != ';' ) &&
       ( ( startPath + position ) <= pathName.length() ) ) {
    ...
}

for ( int employeeNum = employee.first + employee.offset;
      employeeNum < employee.first + employee.offset + employee.total;
      employeeNum++ ) {
    ...
}
```

This meets the criteria set earlier in the chapter. The continuation part of the statement is done logically—it's always indented underneath the statement it continues. The indentation can be done consistently—it uses only a few more spaces than the original line. It's as readable as anything else, and it's as maintainable as anything else. In some cases you might be able to improve readability by fine-tuning the indentation or spacing, but be sure to keep the maintainability trade-off in mind when you consider fine-tuning.

1139
1140
1141

Do not align right sides of assignment statements

In the first edition of this book I recommended aligning the right sides of statements containing assignments as shown in Listing 31-44:

1142
1143
1144
1145
1146

Listing 31-44. Java example of endline layout used for assignment-statement continuation—bad practice.

```
customerPurchases = customerPurchases + CustomerSales( CustomerID );
customerBill     = customerBill + customerPurchases;
totalCustomerBill = customerBill + PreviousBalance( customerID ) +
```

```
1147                         LateCharge( customerID );
1148     customerRating      = Rating( customerID, totalCustomerBill );
1149
1150     With the benefit of 10 years' hindsight, I have found that while this indentation
1151     style might look attractive it becomes a headache to maintain the alignment of
1152     the equals signs as variable names change, code is run through tools that substi-
1153     tute tabs for spaces and spaces for tabs. It is also hard to maintain as lines are
1154     moved among different parts of the program that have different levels of indenta-
1155     tion.
```

```
1155
1156     For consistency with the other indentation guidelines as well as maintainability,
1157     treat groups of statements containing assignment operations just as you would
1158     treat other statements, as Listing 31-45 shows:
```

Listing 31-45. Java example of standard indentation for assignment-statement continuation—good practice.

```
1160     customerPurchases = customerPurchases + CustomerSales( CustomerID );
1161     customerBill = customerBill + customerPurchases;
1162     totalCustomerBill = customerBill + PreviousBalance( customerID ) +
1163         LateCharge( customerID );
1164     customerRating = Rating( customerID, totalCustomerBill );
```

Indent assignment-statement continuation lines the standard amount

In Listing 31-45, the continuation line for the third assignment statement is indented the standard amount. This is done for the same reasons that assignment statements in general are not formatted in any special way—general readability and maintainability.

Using Only One Statement per Line

Modern languages such as C++ and Java allow multiple statements per line. The power of free formatting is a mixed blessing, however, when it comes to putting multiple statements on a line:

```
1174     i = 0; j = 0; k = 0; DestroyBadLoopNames( i, j, k );
```

This line contains several statements that could logically be separated onto lines of their own.

One argument in favor of putting several statements on one line is that it requires fewer lines of screen space or printer paper, which allows more of the code to be viewed at once. It's also a way to group related statements, and some programmers believe that it provides optimization clues to the compiler.

These are good reasons, but the reasons to limit yourself to one statement per line are more compelling:

1183
1184
1185
1186

1187 **CROSS-REFERENCE** Code-level performance optimizations are discussed in Chapter 25, "Code-Tuning Strategies," and Chapter 26, "Code-Tuning Techniques."

1192
1193
1194

1195
1196
1197
1198

1199
1200
1201
1202

1203
1204
1205

1206
1207
1208
1209
1210

1211
1212

1213
1214
1215
1216
1217
1218
1219
1220

- Putting each statement on a line of its own provides an accurate view of a program's complexity. It doesn't hide complexity by making complex statements look trivial. Statements that are complex look complex. Statements that are easy look easy.
- Putting several statements on one line doesn't provide optimization clues to modern compilers. Today's optimizing compilers don't depend on formatting clues to do their optimizations. This is illustrated later in this section.
- With statements on their own lines, the code reads from top to bottom, instead of top to bottom and left to right. When you're looking for a specific line of code, your eye should be able to follow the left margin of the code. It shouldn't have to dip into each and every line just because a single line might contain two statements.
- With statements on their own lines, it's easy to find syntax errors when your compiler provides only the line numbers of the errors. If you have multiple statements on a line, the line number doesn't tell you which statement is in error.
- With one statement to a line, it's easy to step through the code with line-oriented debuggers. If you have several statements on a line, the debugger executes them all at once, and you have to switch to assembler to step through individual statements.
- With one to a line, it's easy to edit individual statements—to delete a line or temporarily convert a line to a comment. If you have multiple statements on a line, you have to do your editing between other statements.

In C++, avoid using multiple operations per line (side effects)

Side effects are consequences of a statement other than its main consequence. In C++, the `++` operator on a line that contains other operations is a side effect. Likewise, assigning a value to a variable and using the left side of the assignment in a conditional is a side effect.

Side effects tend to make code difficult to read. For example, if `n` equals 4, what is the printout of the statement shown in Listing 31-46?

Listing 31-46. C++ example of an unpredictable side effect.

```
PrintMessage( ++n, n + 2 );
```

Is it 4 and 6? Is it 5 and 7? Is it 5 and 6? The answer is None of the above. The first argument, `++n`, is 5. But the C++ language does not define the order in which terms in an expression or arguments to a routine are evaluated. So the compiler can evaluate the second argument, `n + 2`, either before or after the first argument; the result might be either 6 or 7, depending on the compiler. Listing 31-47 shows how you should rewrite the statement so that the intent is clear:

1221
1222
1223
1224
1225

Listing 31-47. C++ example of avoiding an unpredictable side effect.

```
++n;  
PrintMessage( n, n + 2 );
```

If you're still not convinced that you should put side effects on lines by themselves, try to figure out what the routine shown in Listing 31-48 does:

1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237

Listing 31-48. C example of too many operations on a line.

```
strcpy( char * t, char * s ) {  
    while ( *++t = *++s )  
        ;  
}
```

Some experienced C programmers don't see the complexity in that example because it's a familiar function; they look at it and say, "That's *strcpy()*." In this case, however, it's not quite *strcpy()*. It contains an error. If you said, "That's *strcpy()*" when you saw the code, you were recognizing the code, not reading it. This is exactly the situation you're in when you debug a program: The code that you overlook because you "recognize" it rather than read it can contain the error that's harder to find than it needs to be.

1238
1239

The fragment shown in Listing 31-49 is functionally identical to the first and is more readable:

1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251

Listing 31-49. C example of a readable number of operations on each line.

```
strcpy( char * t, char * s ) {  
    do {  
        ++t;  
        ++s;  
        *t = *s;  
    }  
    while ( *t != '\0' );  
}
```

In the reformatted code, the error is apparent. Clearly, *t* and *s* are incremented before **s* is copied to **t*. The first character is missed.

1252
1253
1254

The second example looks more elaborate than the first, even though the operations performed in the second example are identical. The reason it looks more elaborate is that it doesn't hide the complexity of the operations.

1255 **CROSS-REFERENCE** For
1256 details on code tuning, see
1257 Chapter 25, "Code-Tuning
1258 Strategies," and Chapter 26,
1259 "Code-Tuning Techniques."

Improved performance doesn't justify putting multiple operations on the same line either. Because the two *strcpy()* routines are logically equivalent, you would expect the compiler to generate identical code for them. When both versions of the routine were profiled, however, the first version took 4.81 seconds to copy 5,000,000 strings and the second took 4.35 seconds.

1260
1261
1262
1263

In this case, the “clever” version carries an 11 percent speed penalty, which makes it look a lot less clever. The results vary from compiler to compiler, but in general they suggest that until you’ve measured performance gains, you’re better off striving for clarity and correctness first, performance second.

1264
1265
1266
1267
1268

Even if you read statements with side effects easily, take pity on other people who will read your code. Most good programmers need to think twice to understand expressions with side effects. Let them use their brain cells to understand the larger questions of how your code works rather than the syntactic details of a specific expression.

1269

1270 **CROSS-REFERENCE** For
1271 details on documenting data
1272 declarations, see “Comment-
1273 ing Data Declarations” in
1274 Section 32.5. For aspects of
1275 data use, see Chapters 10
through 13.
1276

1277
1278

1279
1280
1281 **CODING HORROR**
1282
1283
1284
1285

1286
1287
1288 **CODING HORROR**
1289
1290
1291
1292
1293
1294
1295
1296

Laying Out Data Declarations

Use only one data declaration per line

As shown in the examples above, you should give each data declaration its own line. It’s easier to put a comment next to each declaration if each one is on its own line. It’s easier to modify declarations because each declaration is self-contained. It’s easier to find specific variables because you can scan a single column rather than reading each line. It’s easier to find and fix syntax errors because the line number the compiler gives you has only one declaration on it.

Quickly—in the data declaration in Listing 31-50, what type of variable is *currentBottom*?

Listing 31-50. C++ example of crowding more than one variable declaration onto a line.

```
int rowIndex, columnIdx; Color previousColor, currentColor, nextColor; Point
previousTop, previousBottom, currentTop, currentBottom, nextTop, nextBottom; Font
previousTypeface, currentTypeface, nextTypeface; Color choices[ NUM_COLORS ];
```

This is an extreme example. But it is not too far removed from a much more common style shown in Listing 31-51:

Listing 31-51. C++ example of crowding more than one variable declaration onto a line.

```
int rowIndex, columnIdx;
Color previousColor, currentColor, nextColor;
Point previousTop, previousBottom, currentTop, currentBottom, nextTop, nextBottom;
Font previousTypeface, currentTypeface, nextTypeface;
Color choices[ NUM_COLORS ];
```

This is not an uncommon style of declaring variables, and the variable is still hard to find because all the declarations are jammed together. The variable’s type is hard to find too.

Now, what is *nextColor*’s type in Listing 31-52?

1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316

Listing 31-52. C++ example of readability achieved by putting only one variable declaration on each line.

```
int rowIndex;
int columnIdx;
Color previousColor;
Color currentColor;
Color nextColor;
Point previousTop;
Point previousBottom;
Point currentTop;
Point currentBottom;
Point nextTop;
Point nextBottom;
Font previousTypeface;
Font currentTypeface;
Font nextTypeface;
Color choices[ NUM_COLORS ];
```

The variable *nextColor* was probably easier to find than *nextTypeface* was in Listing 31-51. This style is characterized by one declaration per line and a complete declaration including the variable type on each line.

1317
1318
1319
1320
1321
1322
1323

Admittedly, this style chews up a lot of screen space—20 lines instead of the 3 in the first example, although those 3 lines were pretty ugly. I can't point to any studies that show that this style leads to fewer bugs or greater comprehension. If Sally Programmer, Jr. asked me to review her code, however, and her data declarations looked like the first example, I'd say, "No way—too hard to read." If they looked like the second example, I'd say, "Uh...maybe I'll get back to you." If they looked like the final example, I would say, "Certainly—it's a pleasure."

1324
1325
1326
1327
1328
1329

Declare variables close to where they're first used

A style that's preferable to declaring all variables in a big block is to declare each variable close to where it's first used. This reduces "span" and "live time" and facilitates refactoring code into smaller routines when necessary. For more details, see "Keep Variables Live for As Short a Time As Possible" in Section 10.4.

1330
1331
1332
1333
1334
1335
1336
1337

Order declarations sensibly

In the example above, the declarations are grouped by types. Grouping by types is usually sensible since variables of the same type tend to be used in related operations. In other cases, you might choose to order them alphabetically by variable name. Although alphabetical ordering has many advocates, my feeling is that it's too much work for what it's worth. If your list of variables is so long that alphabetical ordering helps, your routine is probably too big. Break it up so that you have smaller routines with fewer variables.

1338 ***In C++, put the asterisk next to the variable name in pointer declarations***
1339 ***or declare pointer types***

1340 It's common to see pointer declarations that put the asterisk next to the type, as
1341 in Listing 31-53:

1342 **Listing 31-53. C++ example of asterisks in pointer declarations.**

```
1343 EmployeeList* employees;  
1344 File* inputFile;
```

1345 The problem with putting the asterisk next to the type name rather than the vari-
1346 able name is that, when you put more than one declaration on a line, the asterisk
1347 will apply only to the first variable even though the visual formatting suggests it
1348 applies to all variables on the line.

1349 You can avoid this problem by putting the asterisk next to the variable name
1350 rather than the type name, as in Listing 31-54:

1351 **Listing 31-54. C++ example of using asterisks in pointer declarations.**

```
1352 EmployeeList *employees;  
1353 File *inputFile;
```

1354 This approach has the weakness of suggesting that the asterisk is part of the vari-
1355 able name, which it isn't. The variable can be used either with or without the
1356 asterisk.

1357 The best approach is to declare a type for the pointer and use that instead. An
1358 example is shown in Listing 31-55:

1359 **Listing 31-55. C++ example of good uses of a pointer type in declara-**
1360 **tions.**

```
1361 EmployeeListPointer employees;  
1362 FilePointer inputFile;
```

1363 The particular problem addressed by this approach can be solved either by re-
1364 quiring all pointers to be declared using pointer types, as shown in Listing 31-55,
1365 or by requiring no more than one variable declaration per line. Be sure to choose
1366 at least one of these solutions!

1367

31.6 Laying Out Comments

1368 **CROSS-REFERENCE** For
1369 details on other aspects of
1370 comments, see Chapter 32,
“Self-Documenting Code.”

Comments done well can greatly enhance a program's readability. Comments done poorly can actually hurt it. The layout of comments plays a large role in whether they help or hinder readability.

1371
1372
1373
1374

1375
1376 **CODING HORROR**

1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411

Listing 31-56. Visual Basic example of poorly indented comments.

```
For transactionId = 1 To totalTransactions
    ' get transaction data
    GetTransactionType( transactionType )
    GetTransactionAmount( transactionAmount )

    ' process transaction based on transaction type
    If transactionType = Transaction_Sale Then
        AcceptCustomerSale( transactionAmount )

    Else
        If transactionType = Transaction_CustomerReturn Then

            ' either process return automatically or get manager approval, if required
            If transactionAmount >= MANAGER_APPROVAL_LEVEL Then

                ' try to get manager approval and then accept or reject the return
                ' based on whether approval is granted
                GetMgrApproval( isTransactionApproved )
                If ( isTransactionApproved ) Then
                    AcceptCustomerReturn( transactionAmount )
                Else
                    RejectCustomerReturn( transactionAmount )
                End If
            Else
                ' manager approval not required, so accept return
                AcceptCustomerReturn( transactionAmount )
            End If
        End If
    End If
Next
```

1412
1413

In this example you don't get much of a clue to the logical structure because the comments completely obscure the visual indentation of the code. You might find it hard to believe that anyone ever makes a conscious decision to use such an indentation style, but I've seen it in professional programs and know of at least one textbook that recommends it.

The code shown in Listing 31-57 is exactly the same as in Listing 31-56, except for the indentation of the comments.

1414
1415 For transactionId = 1 To totalTransactions
1416 ' get transaction data
1417 GetTransactionType(transactionType)
1418 GetTransactionAmount(transactionAmount)
1419
1420 ' process transaction based on transaction type
1421 If transactionType = Transaction_Sale Then
1422 AcceptCustomerSale(transactionAmount)
1423
1424 Else
1425 If transactionType = Transaction_CustomerReturn Then
1426
1427 ' either process return automatically or get manager approval, if required
1428 If transactionAmount >= MANAGER_APPROVAL_LEVEL Then
1429
1430 ' try to get manager approval and then accept or reject the return
1431 ' based on whether approval is granted
1432 GetMgrApproval(isTransactionApproved)
1433 If (isTransactionApproved) Then
1434 AcceptCustomerReturn(transactionAmount)
1435 Else
1436 RejectCustomerReturn(transactionAmount)
1437 End If
1438 Else
1439 ' manager approval not required, so accept return
1440 AcceptCustomerReturn(transactionAmount)
1441 End If
1442 End If
1443 End If
1444 Next

1445 In Listing 31-57, the logical structure is more apparent. One study of the effectiveness of commenting found that the benefit of having comments was not conclusive, and the author speculated that it was because they “disrupt visual scanning of the program” (Shneiderman 1980). From these examples, it’s obvious that the *style* of commenting strongly influences whether comments are disruptive.

1451 ***Set off each comment with at least one blank line***

1452 If someone is trying to get an overview of your program, the most effective way
1453 to do it is to read the comments without reading the code. Setting comments off
1454 with blank lines helps a reader scan the code. An example is shown in Listing
1455 31-58:

1456 ***Listing 31-58. Java example of setting off a comment with a blank line.***

1457 // comment zero

```
1458     CodeStatementZero;  
1459     CodeStatementOne;  
1460  
1461     // comment one  
1462     CodeStatementTwo;  
1463     CodeStatementThree;
```

Some people use a blank line both before and after the comment. Two blanks use more display space, but some people think the code looks better than with just one. An example is shown in Listing 31-59:

Listing 31-59. Java example of setting off a comment with two blank lines.

```
// comment zero  
  
CodeStatementZero;  
CodeStatementOne;  
  
// comment one  
  
CodeStatementTwo;  
CodeStatementThree;
```

Unless your display space is at a premium, this is a purely aesthetic judgment and you can make it accordingly. In this, as in many other areas, the fact that a convention exists is more important than the convention's specific details.

31.7 Laying Out Routines

CROSS-REFERENCE For details on documenting routines, see “Commenting Routines” in Section 32.5. For details on the process of writing a routine, see Section 9.3, “Constructing Routines Using the PPP.” For a discussion of the differences between good and bad routines, see Chapter 7, “High-Quality Routines.”

Routines are composed of individual statements, data, control structures, comments—all the things discussed in the other parts of the chapter. This section provides layout guidelines unique to routines.

Use blank lines to separate parts of a routine

Use blank lines between the routine header, its data and named-constant declarations (if any), and its body.

Use standard indentation for routine arguments

The options with routine-header layout are about the same as they are in a lot of other areas of layout: no conscious layout, endline layout, or standard indentation. As in most other cases, standard indentation does better in terms of accuracy, consistency, readability, and modifiability.

Listing 31-60 shows two examples of routine headers with no conscious layout:

1495
1496
Listing 31-60. C++ examples of routine headers with no conscious lay-
1497 **out.**

```
1498     bool ReadEmployeeData(int maxEmployees,EmployeeList *employees,  
1499         EmployeeFile *inputFile,int *employeeCount,bool *isInputError)  
1500     ...  
1501     void InsertionSort(SortArray data,int firstElement,int lastElement)
```

These routine headers are purely utilitarian. The computer can read them as well as it can read headers in any other format, but they cause trouble for humans. Without a conscious effort to make the headers hard to read, how could they be any worse?

1506
1507 The second approach in routine-header layout is the endline layout, which usually works all right. Listing 31-61 shows the same routine headers reformatted:

1508
1509
Listing 31-61. C++ example of routine headers with mediocre endline
layout.

```
1510     bool ReadEmployeeData( int             maxEmployees,  
1511                           EmployeeList   *employees,  
1512                           EmployeeFile  *inputFile,  
1513                           int            *employeeCount,  
1514                           bool           *isInputError )  
1515     ...  
1516     void InsertionSort( SortArray    data,  
1517                           int          firstElement,  
1518                           int          lastElement )
```

1519 **CROSS-REFERENCE** For
1520 more details on using routine
1521 parameters, see Section 7.5,
1522 “How to Use Routine Pa-
1523 rameters.”

The endline approach is neat and aesthetically appealing. The main problem is that it takes a lot of work to maintain, and styles that are hard to maintain aren’t maintained. Suppose that the function name changes from *ReadEmployeeData()* to *ReadNewEmployeeData()*. That would throw the alignment of the first line off from the alignment of the other four lines. You’d have to reformat the other four lines of the parameter list to align with the new position of *maxEmployees* caused by the longer function name. And you’d probably run out of space on the right side since the elements are so far to the right already.

1527
1528 The examples shown in Listing 31-62, reformatted using standard indentation, are just as appealing aesthetically but take less work to maintain.

1529
1530
Listing 31-62. C++ example of routine headers with readable, maintain-
able standard indentation.

```
1531     public bool ReadEmployeeData  
1532         int maxEmployees,  
1533         EmployeeList *employees,  
1534         EmployeeFile *inputFile,  
1535         int *employeeCount,
```

```
1536     bool *isInputError  
1537 )  
1538 ...  
1539  
1540 public void InsertionSort(  
1541     SortArray data,  
1542     int firstElement,  
1543     int lastElement  
1544 )  
1545  
1546  
1547  
1548  
1549  
1550
```

This style holds up better under modification. If the routine name changes, the change has no effect on any of the parameters. If parameters are added or deleted, only one line has to be modified—plus or minus a comma. The visual cues are similar to those in the indentation scheme for a loop or an *if* statement. Your eye doesn't have to scan different parts of the page for every individual routine to find meaningful information; it knows where the information is every time.

```
1551  
1552
```

This style translates to Visual Basic in a straightforward way, though it requires the use of line-continuation characters, as shown in Listing 31-63:

Listing 31-63. Visual Basic example of routine headers with readable, maintainable standard indentation.

```
Public Sub ReadEmployeeData ( _  
    ByVal maxEmployees As Integer, _  
    ByRef employees As EmployeeList, _  
    ByRef inputFile As EmployeeFile, _  
    ByRef employeeCount As Integer, _  
    ByRef isInputError As Boolean _  
)
```

31.8 Laying Out Classes

Here are several guidelines for laying out code within a class. The next section contains guidelines for laying out code within a file.

Laying Out Class Interfaces

In laying out class interfaces, the convention is to present the class members in the following order:

1. Header comment that describes the class and provides any notes about the overall usage of the class
2. Constructors and destructors

1563 CROSS-REFERENCE For details on documenting classes, see “Commenting Classes, Files, and Programs” in Section 32.5. For details on the process of creating classes, see Section 9.1, “Summary of Steps in Building Classes and Routines.” For a discussion of the differences between good and bad classes, see Chapter 6, “Working Classes.”

- 1571 3. Public routines
- 1572 4. Protected routines
- 1573 5. Private routines and member data

1574 **Laying Out Class Implementations**

1575 Class implementations are generally laid out in this order:

- 1576 1. Header comment that describes the contents of the file the class is in
- 1577 2. Class data
- 1578 3. Public routines
- 1579 4. Protected routines
- 1580 5. Private routines

1581 *If you have more than one class in a file, identify each class clearly*

1582 Routines that are related should be grouped together into classes. A reader scanning
1583 your code should be able to tell easily which class is which. Identify each
1584 class clearly by using several blank lines between it and the classes next to it. A
1585 class is like a chapter in a book. In a book, you start each chapter on a new page
1586 and use big print for the chapter title. Emphasize the start of each class similarly.
1587 An example of separating classes is shown in Listing 31-64.

1588 **Listing 31-64. C++ example of formatting the separation between 1589 classes.**

1590 *This is the last routine in a
1591 class.*

```
// create a string identical to sourceString except that the
// blanks are replaced with underscores.
void EditString::ConvertBlanks(
    char *sourceString,
    char *targetString
) {
    Assert( strlen( sourceString ) <= MAX_STRING_LENGTH );
    Assert( sourceString != NULL );
    Assert( targetString != NULL );
    int charIndex = 0;
    do {
        if ( sourceString[ charIndex ] == " " ) {
            targetString[ charIndex ] = '_';
        }
        else {
            targetString[ charIndex ] = sourceString[ charIndex ];
        }
        charIndex++;
    } while ( sourceString[ charIndex ] != '\0' );
}
```

```
1606
1607
1608
1609
1610
1611      The beginning of the new
1612      class is marked with several
1613      blank lines and the name of
1614      the class.
1615
1616
1617      This is the first routine in a
1618      new class.
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628      This routine is separated from
1629      the previous routine by blank
1630      lines only.
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
```

```
        }
        charIndex++;
    } while sourceString[ charIndex ] != '\0';
}

//-----
// MATHEMATICAL FUNCTIONS
//
// This class contains the program's mathematical functions.
//-----

// find the arithmetic maximum of arg1 and arg2
int Math::Max( int arg1, int arg2 ) {
    if ( arg1 > arg2 ) {
        return arg1;
    }
    else {
        return arg2;
    }
}

// find the arithmetic minimum of arg1 and arg2
int Math::Min( int arg1, int arg2 ) {
    if ( arg1 < arg2 ) {
        return arg1;
    }
    else {
        return arg2;
    }
}
```

Avoid overemphasizing comments within classes. If you mark every routine and comment with a row of asterisks instead of blank lines, you'll have a hard time coming up with a device that effectively emphasizes the start of a new class. An example is shown in Listing 31-65.

Listing 31-65. C++ example of overformatting a class.

```
*****
*****
// MATHEMATICAL FUNCTIONS
//
// This class contains the program//s mathematical functions.
*****
*****
```

```
1651 // find the arithmetic maximum of arg1 and arg2
1652 //*****
1653 int Math::Max( int arg1, int arg2 ) {
1654     //*****
1655     if ( arg1 > arg2 ) {
1656         return arg1;
1657     }
1658     else {
1659         return arg2;
1660     }
1661 }
1662
1663 //*****
1664 // find the arithmetic maximum of arg1 and arg2
1665 //*****
1666 int Math::Min( int arg1, int arg2 ) {
1667     //*****
1668     if ( arg1 < arg2 ) {
1669         return arg1;
1670     }
1671     else {
1672         return arg2;
1673     }
1674 }
```

In this example, so many things are highlighted with asterisks that nothing is really emphasized. The program becomes a dense forest of asterisks. Although it's more an aesthetic than a technical judgment, in formatting, less is more.

If you must separate parts of a program with long lines of special characters, develop a hierarchy of characters (from densest to lightest) instead of relying exclusively on asterisks. For example, use asterisks for class divisions, dashes for routine divisions, and blank lines for important comments. Refrain from putting two rows of asterisks or dashes together. An example is shown in Listing 31-66.

Listing 31-66. C++ example of good formatting with restraint.

```
1683 //*****
1684 // MATHEMATICAL FUNCTIONS
1685 //
1686 // This class contains the program's mathematical functions.
1687 //*****
1688
1689
1690 //-----
```

The lightness of this line compared to the line of asterisks visually reinforces the fact that the routine is subordinate to the class.

1695

1696

1697

1698

1699

1700

1701

1702

1703

1704

1705

1706

1707

1708

1709

1710

1711

1712

1713

1714

1715

1716

1717

1718

1719

1720

1721 **CROSS-REFERENCE** For documentation details, see “Commenting Classes, Files, and Programs” in Section

32.5.

1724

1725

1726

1727

```
// find the arithmetic maximum of arg1 and arg2
//-----
int Math::Max( int arg1, int arg2 ) {
    if ( arg1 > arg2 ) {
        return arg1;
    }
    else {
        return arg2;
    }
}

//-----
// find the arithmetic minimum of arg1 and arg2
//-----
int Math::Min( int arg1, int arg2 ) {
    if ( arg1 < arg2 ) {
        return arg1;
    }
    else {
        return arg2;
    }
}
```

This advice about how to identify multiple classes within a single file applies only when your language restricts the number of files you can use in a program. If you’re using C++, Java, Visual Basic or other languages that support multiple source files, put only one class in each file unless you have a compelling reason to do otherwise (such as including a few small classes that make up a single pattern). Within a single class, however, you might still have subgroups of routines, and you can group them using techniques such as the ones shown here.

Laying Out Files and Programs

Beyond the formatting techniques for routines is a larger formatting issue. How do you organize routines within a file, and how do you decide which routines to put in a file in the first place?

Put one class in one file

A file isn’t just a bucket that holds some code. If your language allows it, a file should hold a collection of routines that supports one and only one purpose. A file reinforces the idea that a collection of routines are in the same class.

1728 **CROSS-REFERENCE** For
1729 details on the differences
1730 between classes and routines
1731 and how to make a collection
1732 of routines into a class, see
Chapter 6, "Working
1733 Classes."

1734

1735

1736

1737

1738

1739

1740

1741

1742

1743

1744

1745

1746

1747

1748

1749

1750

1751

1752

1753

1754

1755 At least two blank lines sepa-
1756 rate the two routines.

1757

1758

1759

1760

1761

1762

1763

1764

1765

1766

1767

1768

All the routines within a file make up the class. The class might be one that the program really recognizes as such, or it might be just a logical entity that you've created as part of your design.

Classes are a semantic language concept. Files are a physical operating-system concept. The correspondence between classes and files is coincidental and continues to weaken over time as more environments support putting code into databases or otherwise obscuring the relationship between routines, classes, and files.

Give the file a name related to the class name

Most projects have a one-to-one correspondence between class names and file names. A class named *CustomerAccount* would have files named *CustomerAccount.cpp* and *CustomerAccount.h*, for example.

Separate routines within a file clearly

Separate each routine from other routines with at least two blank lines. The blank lines are as effective as big rows of asterisks or dashes, and they're a lot easier to type and maintain. Use two or three to produce a visual difference between blank lines that are part of a routine and blank lines that separate routines. An example is shown in Listing 31-67:

Listing 31-67. Visual Basic example of using blank lines between routines.

```
'find the arithmetic maximum of arg1 and arg2
Function Max( arg1 As Integer, arg2 As Integer ) As Integer
    If ( arg1 > arg2 ) Then
        Max = arg1
    Else
        Max = arg2
    End If
End Function
```

```
'find the arithmetic minimum of arg1 and arg2
Function Min( arg1 As Integer, arg2 As Integer ) As Integer
    If ( arg1 < arg2 ) Then
        Min = arg1
    Else
        Min = arg2
    End If
End Function
```

Blank lines are easier to type than any other kind of separator and look at least as good. Three blank lines are used here so that the separation between routines is more noticeable than the blank lines within each routine.

1769 ***Sequence routines alphabetically***
1770 An alternative to grouping related routines in a file is to put them in alphabetical
1771 order. If you can't break a program up into classes or if your editor doesn't allow
1772 you to find functions easily, the alphabetical approach can save search time.

1773 ***In C++, order the source file carefully***
1774 Here's the standard order of source-file contents in C++:

1775 File-description comment
1776 *#include* files
1777 Constant definitions
1778 Enums
1779 Macro function definitions
1780 Type definitions
1781 Global variables and functions imported
1782 Global variables and functions exported
1783 Variables and functions that are private to the file
1784 Classes

CC2E.COM/3194

CHECKLIST: Layout

- 1785
- 1786 **General**
1787 Is formatting done primarily to illuminate the logical structure of the code?
1788 Can the formatting scheme be used consistently?
1789 Does the formatting scheme result in code that's easy to maintain?
1790 Does the formatting scheme improve code readability?
- 1791 **Control Structures**
1792 Does the code avoid doubly indented *begin-end* or {} pairs?
1793 Are sequential blocks separated from each other with blank lines?
1794 Are complicated expressions formatted for readability?
1795 Are single-statement blocks formatted consistently?
1796 Are *case* statements formatted in a way that's consistent with the formatting
1797 of other control structures?

- 1798 Have *gos* been formatted in a way that makes their use obvious?

1799 **Individual Statements**

- 1800 Is white space used to make logical expressions, array references, and rou-
1801 tine arguments readable?
- 1802 Do incomplete statements end the line in a way that's obviously incorrect?
- 1803 Are continuation lines indented the standard indentation amount?
- 1804 Does each line contain at most one statement?
- 1805 Is each statement written without side effects?
- 1806 Is there at most one data declaration per line?

1807 **Comments**

- 1808 Are the comments indented the same number of spaces as the code they
1809 comment?
- 1810 Is the commenting style easy to maintain?

1811 **Routines**

- 1812 Are the arguments to each routine formatted so that each argument is easy to
1813 read, modify, and comment?
- 1814 Are blank lines used to separate parts of a routine?

1815 **Classes, Files and Programs**

- 1816 Is there a one-to-one relationship between classes and files for most classes
1817 and files?
- 1818 If a file does contain multiple classes, are all the routines in each class
1819 grouped together and is the class clearly identified?
- 1820 Are routines within a file clearly separated with blank lines?
- 1821 In lieu of a stronger organizing principle, are all routines in alphabetical se-
1822 quence?

CC2E.COM/3101

1824 **Additional Resources**

1825 Most programming textbooks say a few words about layout and style, but thor-
1826 ough discussions of programming style are rare; discussions of layout are rarer
1827 still. The following books talk about layout and programming style.

1828 Kernighan, Brian W. and Rob Pike. *The Practice of Programming*, Reading,
1829 Mass.: Addison Wesley, 1999. Chapter 1 of this book discusses programming
1830 style focusing on C and C++.

- 1831 Vermeulen, Allan, et al. *The Elements of Java Style*, Cambridge University
1832 Press, 2000.
- 1833 Bumgardner, Greg, Andrew Gray, and Trevor Misfeldt, 2004. *The Elements of
1834 C++ Style*, Cambridge University Press, 2004.
- 1835 Kernighan, Brian W., and P. J. Plauger. *The Elements of Programming Style*, 2d
1836 ed. New York: McGraw-Hill, 1978. This is the classic book on programming
1837 style—the first in the genre of programming-style books.
- 1838 For a substantially different approach to readability, see the discussion of Donald
1839 Knuth’s “literate programming” listed below.
- 1840 Knuth, Donald E. *Literate Programming*. Cambridge University Press, 2001.
1841 This is a collection of papers describing the “literate programming” approach of
1842 combining a programming language and a documentation language. Knuth has
1843 been writing about the virtues of literate programming for about 20 years, and in
1844 spite of his strong claim to the title Best Programmer on the Planet, literate pro-
1845 gramming isn’t catching on. Read some of his code to form your own conclu-
1846 sions about the reason.

1847 Key Points

- 1848 • The first priority of visual layout is to illuminate the logical organization of
1849 the code. Criteria used to assess whether the priority is achieved include ac-
1850 curacy, consistency, readability, and maintainability.
- 1851 • Looking good is secondary to the other criteria—a distant second. If the
1852 other criteria are met and the underlying code is good, however, the layout
1853 will look fine.
- 1854 • Visual Basic has pure blocks and the conventional practice in Java is to use
1855 pure block style, so you can use a pure-block layout if you program in those
1856 languages. In C++, either pure-block emulation or *begin-end* block bounda-
1857 ries work well.
- 1858 • Structuring code is important for its own sake. The specific convention you
1859 follow may be less important than the fact that you follow some convention
1860 consistently. A layout convention that’s followed inconsistently might actu-
1861 ally hurt readability.
- 1862 • Many aspects of layout are religious issues. Try to separate objective prefer-
1863 ences from subjective ones. Use explicit criteria to help ground your discus-
1864 sions about style preferences.

32

Self-Documenting Code

3 CC2E.COM/3245

4

5

6

7

8

9

10

11

12

13

14

15 *Code as if whoever
16 maintains your program
17 is a violent psychopath
18 who knows where you
19 live.*
20 —Anonymous

Contents

- 32.1 External Documentation
- 32.2 Programming Style as Documentation
- 32.3 To Comment or Not to Comment
- 32.4 Keys to Effective Comments
- 32.5 Commenting Techniques

Related Topics

- Layout: Chapter 31
- The Pseudocode Programming Process: Chapter 9
- High quality classes: Chapter 6
- High-quality routines: Chapter 7
- Programming as communication: Sections 33.5 and 34.3

MOST PROGRAMMERS ENJOY WRITING DOCUMENTATION if the documentation standards aren't unreasonable. Like layout, good documentation is a sign of the professional pride a programmer puts into a program. Software documentation can take many forms, and, after describing the sweep of the documentation landscape, this chapter cultivates the specific patch of documentation known as "comments."

32.1 External Documentation

Documentation on a software project consists of information both inside the source-code listings and outside them—usually in the form of separate documents or unit development folders. On large, formal projects, most of the documentation is outside the source code (Jones 1998). External construction documentation tends to be at a high level compared to the code, at a low level compared to the documentation from problem definition, requirements, and architecture.

22 **HARD DATA**

23

24

25

26

27

28

29 **FURTHER READING** For a
30 detailed description, see “The
31 Unit Development Folder
32 (UDF): An Effective
33 Management Tool for
34 Software Development”
35 (Ingrassia 1976) or “The Unit
36 Development Folder (UDF):
37 A Ten-Year Perspective”
38 (Ingrassia 1987).

39

40
41
42
43
44
45
46
47
48

Unit development folders

A Unit Development Folder (UDF), or software-development folder (SDF), is an informal document that contains notes used by a developer during construction. A “unit” is loosely defined, usually to mean a class. The main purpose of a UDF is to provide a trail of design decisions that aren’t documented elsewhere. Many projects have standards that specify the minimum content of a UDF, such as copies of the relevant requirements, the parts of the top-level design the unit implements, a copy of the development standards, a current code listing, and design notes from the unit’s developer. Sometimes the customer requires a software developer to deliver the project’s UDFs; often they are for internal use only.

Detailed-design document

The detailed-design document is the low-level design document. It describes the class-level or routine-level design decisions, the alternatives that were considered, and the reasons for selecting the approaches that were selected. Sometimes this information is contained in a formal document. In such cases, detailed design is usually considered to be separate from construction. Sometimes it consists mainly of developer’s notes collected into a “Unit Development Folder” (UDF). Sometimes—often—it exists only in the code itself.

49

32.2 Programming Style as Documentation

50
51
52
53
54

In contrast to external documentation, internal documentation is found within the program listing itself. It’s the most detailed kind of documentation, at the source-statement level. Because it’s most closely associated with the code, internal documentation is also the kind of documentation most likely to remain correct as the code is modified.

55
56
57
58
59

The main contributor to code-level documentation isn’t comments, but good programming style. Style includes good program structure, use of straightforward and easily understandable approaches, good variable names, good routine names, use of named constants instead of literals, clear layout, and minimization of control-flow and data-structure complexity.

60

Here’s a code fragment with poor style:

CODING HORROR

61
62
63
64
65

Java Example of Poor Documentation Resulting from Bad Programming Style

```
for ( i = 1; i <= num; i++ ) {  
    meetsCriteria[ i ] = True;  
}
```

```

66     for ( i = 2; i <= num / 2; i++ ) {
67         j = i + i;
68         while ( j <= num ) {
69             meetsCriteria[ j ] = False;
70             j = j + i;
71         }
72     }
73     for ( i = 1; i <= num; i++ ) {
74         if ( meetsCriteria[ i ] ) {
75             System.out.println ( i + " meets criteria." );
76         }
77     }

```

What do you think this routine does? It's unnecessarily cryptic. It's poorly documented not because it lacks comments, but because it lacks good programming style. The variable names are uninformative, and the layout is crude. Here's the same code improved—just improving the programming style makes its meaning much clearer:

Java Example of Documentation Without Comments (with Good Style)

```

for ( primeCandidate = 1; primeCandidate <= num; primeCandidate++ ) {
    isPrime[ primeCandidate ] = True;
}

for ( int factor = 2; factor < ( num / 2 ); factor++ ) {
    int factorableNumber = factor + factor;
    while ( factorableNumber <= num ) {
        isPrime[ factorableNumber ] = False;
        factorableNumber = factorableNumber + factor;
    }
}

for ( primeCandidate = 1; primeCandidate <= num; primeCandidate++ ) {
    if ( isPrime[ primeCandidate ] ) {
        System.out.println( primeCandidate + " is prime." );
    }
}

```

Unlike the first piece of code, this one lets you know at first glance that it has something to do with prime numbers. A second glance reveals that it finds the prime numbers between *1* and *Num*. With the first code fragment, it takes more than two glances just to figure out where the loops end.

The difference between the two code fragments has nothing to do with comments. Neither fragment has any. The second one is much more readable, however, and approaches the Holy Grail of legibility: self-documenting code. Such code relies

CROSS-REFERENCE In
this code, the variable
FactorableNumber is added
solely for the sake of
clarifying the operation. For
details on adding variables to
clarify operations, see
"Making Complicated
Expressions Simple" in
Section 19.1.

92
93
94
95

96
97
98
99
100
101
102
103
104

105
106
107

108
109

on good programming style to carry the greater part of the documentation burden. In well-written code, comments are the icing on the readability cake.

110 CC2E.COM/3252

CHECKLIST: Self-Documenting Code

111

112
113
114
115
116

Classes

- Does the class's interface present a consistent abstraction?
- Is the class well named, and does its name describe its central purpose?
- Does the class's interface make obvious how you should use the class?
- Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?

117

118
119
120
121
122

Routines

- Does each routine's name describe exactly what the routine does?
- Does each routine perform one well-defined task?
- Have all parts of each routine that would benefit from being put into their own routines been put into their own routines?
- Is each routine's interface obvious and clear?

123

124
125
126
127
128
129
130
131
132

Data Names

- Are type names descriptive enough to help document data declarations?
- Are variables named well?
- Are variables used only for the purpose for which they're named?
- Are loop counters given more informative names than *i*, *j*, and *k*?
- Are well-named enumerated types used instead of makeshift flags or boolean variables?
- Are named constants used instead of magic numbers or magic strings?
- Do naming conventions distinguish among type names, enumerated types, named constants, local variables, class variables, and global variables?

133

134
135
136
137
138

Data Organization

- Are extra variables used for clarity when needed?
- Are references to variables close together?
- Are data types simple so that they minimize complexity?
- Is complicated data accessed through abstract access routines (abstract data types)?

139

140

Control

- Is the nominal path through the code clear?

- 141 Are related statements grouped together?
142 Have relatively independent groups of statements been packaged into their
143 own routines?
144 Does the normal case follow the *if* rather than the *else*?
145 Are control structures simple so that they minimize complexity?
146 Does each loop perform one and only one function, as a well-defined routine
147 would?
148 Is nesting minimized?
149 Have boolean expressions been simplified by using additional boolean
150 variables, boolean functions, and decision tables?

151 **Layout**

- 152 Does the program's layout show its logical structure?

153 **Design**

- 154 Is the code straightforward, and does it avoid cleverness?
155 Are implementation details hidden as much as possible?
156 Is the program written in terms of the problem domain as much as possible
157 rather than in terms of computer-science or programming-language
158 structures?

160 **32.3 To Comment or Not to Comment**

161 Comments are easier to write poorly than well, and commenting can be more
162 damaging than helpful. The heated discussions over the virtues of commenting
163 often sound like philosophical debates over moral virtues, which makes me think
164 that if Socrates had been a computer programmer, he and his students might have
165 had the following discussion.

166 **❖ THE COMMENTO ❖**

167 CHARACTERS:

168 THRASYMACHUS A green, theoretical purist who believes everything he
169 reads

170 CALLICLES A battle-hardened veteran from the old school—a “real”
171 programmer

172 GLAUCON A young, confident, hot-shot computer jock

173 ISMENE A senior programmer tired of big promises, just looking for a
174 few practices that work

175 SOCRATES The wise old programmer

176 SETTING: The Weekly Team Meeting

177 “I want to suggest a commenting standard for our projects,” Thrasymachus said.
178 “Some of our programmers barely comment their code, and everyone knows that
179 code without comments is unreadable.”

180 “You must be fresher out of college than I thought,” Callicles responded.
181 “Comments are an academic panacea, but everyone who’s done any real
182 programming knows that comments make the code harder to read, not easier.
183 English is less precise than Java or Visual Basic and makes for a lot of excess
184 verbiage. Programming-language statements are short and to the point. If you
185 can’t make the code clear, how can you make the comments clear? Plus,
186 comments get out of date as the code changes. If you believe an out-of-date
187 comment, you’re sunk.”

188 “I agree with that,” Glaucon joined in. “Heavily commented code is harder to
189 read because it means more to read. I already have to read the code; why should I
190 have to read a lot of comments too?”

191 “Wait a minute,” Ismene said, putting down her coffee mug to put in her two
192 drachmas’ worth. “I know that commenting can be abused, but good comments
193 are worth their weight in gold. I’ve had to maintain code that had comments and
194 code that didn’t, and I’d rather maintain code with comments. I don’t think we
195 should have a standard that says use one comment for every x lines of code, but
196 we should encourage everyone to comment.”

197 “If comments are a waste of time, why does anyone use them, Callicles?”
198 Socrates asked.

199 “Either because they’re required to or because they read somewhere that they’re
200 useful. No one who’s thought about it could ever decide they’re useful.”

201 “Ismene thinks they’re useful. She’s been here three years, maintaining your
202 code without comments and other code with comments, and she prefers the code
203 with comments. What do you make of that?”

204 “Comments are useless because they just repeat the code in a more verbose—”

KEY POINT

205 “Wait right there,” Thrasymachus interrupted. “Good comments don’t repeat the
206 code or explain it. They clarify its intent. Comments should explain, at a higher
207 level of abstraction than the code, what you’re trying to do.”

208 “Right,” Ismene said. “I scan the comments to find the section that does what I
209 need to change or fix. You’re right that comments that repeat the code don’t help
210 at all because the code says everything already. When I read comments, I want it
211 to be like reading headings in a book, or a table of contents. Comments help me
212 find the right section, and then I start reading the code. It’s a lot faster to read
213 one sentence in English than it is to parse 20 lines of code in a programming
214 language.” Ismene poured herself another cup of coffee.

215 “I think that people who refuse to write comments (1) think their code is clearer
216 than it could possibly be; (2) think that other programmers are far more
217 interested in their code than they really are; (3) think other programmers are
218 smarter than they really are; (4) are lazy; or (5) are afraid someone else might
219 figure out how their code works.

220 “Code reviews would be a big help here, Socrates,” Ismene continued. “If
221 someone claims they don’t need to write comments and are bombarded by
222 questions in a review—several peers start saying, ‘What the heck are you trying
223 to do in this piece of code?’—then they’ll start putting in comments. If they
224 don’t do it on their own, at least their manager will have the ammo to make them
225 do it.

226 “I’m not accusing you of being lazy or afraid that people will figure out your
227 code, Callicles. I’ve worked on your code and you’re one of the best
228 programmers in the company. But have a heart, huh? Your code would be easier
229 for me to work on if you used comments.”

230 “But they’re a waste of resources,” Callicles countered. “A good programmer’s
231 code should be self-documenting; everything you need to know should be in the
232 code.”

233 “No way!” Thrasymachus was out of his chair. “Everything the compiler needs
234 to know is in the code! You might as well argue that everything you need to
235 know is in the binary executable file! If you were smart enough to read it! What
236 is *meant* to happen is not in the code.”

237 Thrasymachus realized he was standing up and sat down. “Socrates, this is
238 ridiculous. Why do we have to argue about whether or not comments are
239 valuable? Everything I’ve ever read says they’re valuable and should be used
240 liberally. We’re wasting our time.”

241 *Clearly, at some level
comments have to be
useful. To believe
otherwise would be to
believe that the
comprehensibility of a
program is independent
of how much information
the reader might already
have about it.*

242 —B. A. Sheil

250

251
252

253
254
255

256
257
258
259
260
261

262
263
264
265
266
267
268
269
270

271
272
273
274
275

“Cool down, Thrasymachus. Ask Callicles how long he’s been programming.”

“How long, Callicles?”

“Well, I started on the Acropolis IV about 15 years ago. I guess I’ve seen about a dozen major systems from the time they were born to the time we gave them a cup of hemlock. And I’ve worked on major parts of a dozen more. Two of those systems had over half a million lines of code, so I know what I’m talking about. Comments are pretty useless.”

Socrates looked at the younger programmer. “As Callicles says, comments have a lot of legitimate problems, and you won’t realize that without more experience. If they’re not done right, they’re worse than useless.”

“Even when they’re done right, they’re useless,” Callicles said. “Comments are less precise than a programming language. I’d rather not have them at all.”

“Wait a minute,” Socrates said. “Ismene agrees that comments are less precise. Her point is that comments give you a higher level of abstraction, and we all know that levels of abstraction are one of a programmer’s most powerful tools.”

“I don’t agree with that. Instead of focusing on commenting, you should focus on making code more readable. Refactoring eliminates most of my comments. Once I’ve refactored, my code might have 20 or 30 routine calls without needing any comments. A good programmer can read the intent from the code itself, and what good does it do to read about somebody’s intent when you know the code has an error?” Glaucon was pleased with his contribution. Callicles nodded.

“It sounds like you guys have never had to modify someone else’s code,” Ismene said. Callicles suddenly seemed very interested in the pencil marks on the ceiling tiles. “Why don’t you try reading your own code six months or a year after you write it? You can improve your code-reading ability and your commenting. You don’t have to choose one or the other. If you’re reading a novel, you might not want section headings. But if you’re reading a technical book, you’d like to be able to find what you’re looking for quickly. I shouldn’t have to switch into ultra-concentration mode and read hundreds of lines of code just to find the two lines I want to change.”

“All right, I can see that it would be handy to be able to scan code,” Glaucon said. He’d seen some of Ismene’s programs and had been impressed. “But what about Callicles’ other point, that comments get out of date as the code changes? I’ve only been programming for a couple of years, but even I know that nobody updates their comments.”

276 “Well, yes and no,” Ismene said. “If you take the comment as sacred and the
277 code as suspicious, you are in deep trouble. Actually, finding a disagreement
278 between the comment and the code tends to mean that both are wrong. The fact
279 that some comments are bad doesn’t mean that commenting is bad. I’m going to
280 the lunchroom to get another pot of coffee.” Ismene left the room.

281 “My main objection to comments,” Callicles said, “is that they’re a waste of
282 resources.”

283 “Can anyone think of ways to minimize the time it takes to write the
284 comments?” Socrates asked.

285 “Design routines in pseudocode, and then convert the pseudocode to comments
286 and fill in the code between them,” Glaucon said.

287 “OK, that would work as long as the comments don’t repeat the code,” Callicles
288 said.

289 “Writing a comment makes you think harder about what your code is doing,”
290 Ismene said, returning from the lunchroom. “If it’s hard to comment, either it’s
291 bad code or you don’t understand it well enough. Either way, you need to spend
292 more time on the code, so the time you spend commenting isn’t wasted.”

293 “All right,” Socrates said. “I can’t think of any more questions, and I think
294 Ismene got the best of you guys today. We’ll encourage commenting, but we
295 won’t be naive about it. We’ll have code reviews so that everyone will get a
296 good sense of the kind of comments that actually help. If you have trouble
297 understanding someone else’s code, let them know how they can improve it.”

298

299 *As long as there are ill-defined goals, bizarre
300 bugs, and unrealistic
301 schedules, there will be
302 Real Programmers
303 willing to jump in and
304 Solve The Problem,
305 saving the documentation
306 for later. Long live
307 Fortran!
308 —Ed Post, from “Real
309 Programmers Don’t Use
310 Pascal”
311*

312
313
314
315

316

317
318
319
320
321
322
323
324
325
326
327

328
329
330

331

332
333

32.4 Keys to Effective Comments

What does the following routine do?

Java Mystery Routine Number One

```
// write out the sums 1..n for all n from 1 to num
current = 1;
previous = 0;
sum = 1;
for ( int i = 0; i < num; i++ ) {
    System.out.println( "Sum = " + sum );
    sum = current + previous;
    previous = current;
    current = sum;
}
```

Your best guess?

This routine computes the first *num* Fibonacci numbers. Its coding style is a little better than the style of the routine at the beginning of the chapter, but the comment is wrong, and if you blindly trust the comment, you head down the primrose path in the wrong direction.

What about this one?

Java Mystery Routine Number Two

```
// set product to "base"
product = base;

// loop from 2 to "num"
for ( int i = 2; i <= num; i++ ) {
    // multiply "base" by "product"
    product = product * base;
}
System.out.println( "Product = " + product );
```

Your best guess?

This routine raises an integer *base* to the integer power *num*. The comments in this routine are accurate, but they add nothing to the code. They are merely a more verbose version of the code itself.

Here’s one last routine:

Java Mystery Routine Number Three

```
// compute the square root of Num using the Newton-Raphson approximation
```

```
334     r = num / 2;
335     while ( abs( r - (num/r) ) > TOLERANCE ) {
336         r = 0.5 * ( r + (num/r) );
337     }
338     System.out.println( "r = " + r );
339     Your best guess?
```

340 This routine computes the square root of *num*. The code isn't great, but the
341 comment is accurate.

342 Which routine was easiest for you to figure out correctly? None of the routines is
343 particularly well written—the variable names are especially poor. In a nutshell,
344 however, these routines illustrate the strengths and weaknesses of internal
345 comments. Routine One has an incorrect comment. Routine Two's commenting
346 merely repeats the code and is therefore useless. Only Routine Three's
347 commenting earns its rent. Poor comments are worse than no comments.
348 Routines One and Two would be better with no comments than with the poor
349 comments they have.

350 The following subsections describe keys to writing effective comments.

351 **Kinds of Comments**

352 Comments can be classified into five categories:

353 **Repeat of the Code**

354 A repetitious comment restates what the code does in different words. It merely
355 gives the reader of the code more to read without providing additional
356 information.

357 **Explanation of the Code**

358 Explanatory comments are typically used to explain complicated, tricky, or
359 sensitive pieces of code. In such situations they are useful, but usually that's only
360 because the code is confusing. If the code is so complicated that it needs to be
361 explained, it's nearly always better to improve the code than it is to add
362 comments. Make the code itself clearer, and then use summary or intent
363 comments.

364 **Marker in the Code**

365 A marker comment is one that isn't intended to be left in the code. It's a note to
366 the developer that the work isn't done yet. Some developers type in a marker
367 that's syntactically incorrect (*****
368 for example) so that the compiler flags it and reminds them that they have more work to do. Other developers put a

369 specified set of characters in comments so that they can search for them but they
370 don't interfere with compilation.

371 Few feelings are worse than having a customer report a problem in the code,
372 debugging the problem, and tracing it to a section of code where you find
373 something like this:

374 return NULL; // ***** NOT DONE! FIX BEFORE RELEASE!!!

375 Releasing defective code to customers is bad enough; releasing code that you
376 knew was defective is even worse.

377 I have found that standardizing the style of marker comments is helpful. If you
378 don't standardize, some programmers will use *****, some will use !!!!!,
379 some will use *TBD*, and some will use various other conventions. Using a variety
380 of notations makes mechanical searching for incomplete code error prone or
381 impossible. Standardizing on one specific technique—such as using *TBD*—allows
382 you to do a mechanical search for incomplete sections of code as one of the steps
383 in a release checklist, which avoids the *FIX BEFORE RELEASE!!!* problem.

384 **Summary of the Code**

385 A comment that summarizes code does just that: It distills a few lines of code
386 into one or two sentences. Such comments are more valuable than comments that
387 merely repeat the code because a reader can scan them more quickly than the
388 code. Summary comments are particularly useful when someone other than the
389 code's original author tries to modify the code.

390 **Description of the Code's Intent**

391 A comment at the level of intent explains the purpose of a section of code. Intent
392 comments operate more at the level of the problem than at the level of the
393 solution. For example,

394 -- get current employee information
395 is an intent comment, whereas

396 -- update employeeRecord object
397 **HARD DATA**
398 is a summary comment in terms of the solution. A six-month study conducted by
399 IBM found that maintenance programmers “most often said that understanding
400 the original programmer’s intent was the most difficult problem” (Fjelstad and
401 Hamlen 1979). The distinction between intent and summary comments isn’t
402 always clear, and it’s usually not important. Examples of intent comments are
given throughout the chapter.

403 The only two kinds of comments that are acceptable for completed code are
404 intent and summary comments.

405 Commenting Efficiently

406 Effective commenting isn't that time-consuming. Too many comments are as
407 bad as too few, and you can achieve a middle ground economically.

408 Comments can take a lot of time to write for two common reasons. First, the
409 commenting style might be time-consuming or tedious—a pain in the neck. If it
410 is, find a new style. A commenting style that requires a lot of busy work is a
411 maintenance headache: If the comments are hard to change, they won't be
412 changed; they'll become inaccurate and misleading, which is worse than having
413 no comments at all.

414 Second, commenting might be difficult because the words to describe what the
415 program is doing don't come easily. That's usually a sign that you don't really
416 understand what the program does. The time you spend "commenting" is really
417 time spent understanding the program better, which is time that needs to be spent
418 regardless of whether you comment.

419 *Use styles that don't break down or discourage modification*

420 Any style that's too fancy is annoying to maintain. For example, pick out the part
421 of the comment below that won't be maintained:

422 Java Example of a Commenting Style That's Hard to Maintain

```
423 // Variable      Meaning
424 // -----      -----
425 // xPos ..... XCoordinate Position (in meters)
426 // yPos ..... YCoordinate Position (in meters)
427 // ndsCmptng.... Needs Computing (= 0 if no computation is needed,
428 //                   = 1 if computation is needed)
429 // ptGrdTtl..... Point Grand Total
430 // ptValMax..... Point Value Maximum
431 // psblScrMax.... Possible Score Maximum
```

432 If you said that the leader dots (....) will be hard to maintain, you're right! They
433 look nice, but the list is fine without them. They add busy work to the job of
434 modifying comments, and you'd rather have accurate comments than nice-
435 looking ones, if that's the choice—and it usually is.

436 Here's another example of a common style that's hard to maintain:

437 C++ Example of a Commenting Style That's Hard to Maintain

```
438 ****
439 * class: GigaTron (GIGATRON.CPP) *
440 * *
441 * author: Dwight K. Coder *
```

```

442 * date: July 4, 2014 *
443 *
444 * Routines to control the twenty-first century's code evaluation *
445 * tool. The entry point to these routines is the EvaluateCode() *
446 * routine at the bottom of this file.
447 *****/

```

This is a nice-looking block comment. It's clear that the whole block belongs together, and the beginning and ending of the block are obvious. What isn't clear about this block is how easy it is to change. If you have to add the name of a file to the bottom of the comment, chances are pretty good that you'll have to fuss with the pretty column of asterisks at the right. If you need to change the paragraph comments, you'll have to fuss with asterisks on both the left and the right. In practice, this means that the block won't be maintained because it will be too much work. If you can press a key and get neat columns of asterisks, that's great. Use it. The problem isn't the asterisks but that they're hard to maintain. The following comment looks almost as good and is a cinch to maintain:

C++ Example of a Commenting Style That's Easy to Maintain

```

/*****
class: GigaTron (GIGATRON.CPP)

author: Dwight K. Coder
date: July 4, 2014

Routines to control the twenty-first century's code evaluation
tool. The entry point to these routines is the EvaluateCode()
routine at the bottom of this file.
*****/

```

Here's a particularly hard style to maintain:

Visual Basic Example of a Commenting Style That's Hard to Maintain

```

' set up Color enumerated type
' +-----+
...
' set up Vegetable enumerated type
' +-----+
...

```

It's hard to know what value the plus sign at the beginning and end of each dashed line adds to the comment, but easy to guess that every time a comment changes, the underline has to be adjusted so that the ending plus sign is in precisely the right place. And what do you do when a comment spills over into two lines? How do you align the plus signs? Take words out of the comment so

CODING HORROR

472
473
474
475
476
477
478
479
480
481
482
483

484
485 that it takes up only one line? Make both lines the same length? The problems
with this approach multiply when you try to apply it consistently.

486
487 A common guideline for Java and C++ that arises from a similar motivation is to
488 use // syntax for single-line comments and /* ... */ syntax for longer comments,
as shown here:

489
490 **Java Example of Using Different Comment Syntaxes for Different
Purposes**

491
492
493 // This is a short comment
494
495 ...
496 /* This is a much longer comment. Four score and seven years ago our fathers
497 brought forth on this continent a new nation, conceived in liberty and dedicated to
498 the proposition that all men are created equal. Now we are engaged in a great civil
499 war, testing whether that nation or any nation so conceived and so dedicated can
500 long endure. We are met on a great battlefield of that war. We have come to
501 dedicate a portion of that field as a final resting-place for those who here gave
502 their lives that that nation might live. It is altogether fitting and proper that
503 we should do this.
504 */
505

The first comment is easy to maintain as long as it is kept short. For longer
comments, the task of creating long columns of double slashes, manually
breaking lines of text between rows, and similar activities is not very rewarding,
and so the /* ... */ syntax is more appropriate for multi-line comments.

506 **KEY POINT**

507
508
509
510
511
512
513
514 The point is that you should pay attention to how you spend your time. If you
spend a lot of time entering and deleting dashes to make plus signs line up,
you're not programming; you're wasting time. Find a more efficient style. In the
case of the underlines with plus signs, you could choose to have just the
comments without any underlining. If you need to use underlines for emphasis,
find some way other than underlines with plus signs to emphasize those
comments. One way would be to have a standard underline that's always the
same length regardless of the length of the comment. Such a line requires no
maintenance, and you can use a text-editor macro to enter it in the first place.

515 **CROSS-REFERENCE** For
516 details on the Pseudocode
517 Programming Process, see
518 Chapter 9, "The Pseudocode
519 Programming Process."

520
521 **Use the Pseudocode Programming Process to reduce commenting time**
522 If you outline the code in comments before you write it, you win in several ways.
When you finish the code, the comments are done. You don't have to dedicate
time to comments. You also gain all the design benefits of writing in high-level
pseudocode before filling in the low-level programming-language code.

523 becomes a task in its own right, which makes it seem like more work than when
524 it's done a little bit at a time. Commenting done later takes more time because
525 you have to remember or figure out what the code is doing instead of just writing
526 down what you're already thinking about. It's also less accurate because you
527 tend to forget assumptions or subtleties in the design.

528 The common argument against commenting as you go along is "When you're
529 concentrating on the code you shouldn't break your concentration to write
530 comments." The appropriate response is that, if you have to concentrate so hard
531 on writing code that commenting interrupts your thinking, you need to design in
532 pseudocode first and then convert the pseudocode to comments. Code that
533 requires that much concentration is a warning sign.

534 **KEY POINT**
535 If your design is hard to code, simplify the design before you worry about
536 comments or code. If you use pseudocode to clarify your thoughts, coding is
straightforward and the comments are automatic.

537 **Performance is not a good reason to avoid commenting**
538 One recurring attribute of the rolling wave of technology discussed in Section
539 4.3 is interpreted environments in which commenting imposes a measurable
540 performance penalty. In the 1980s, comments in Basic programs on the original
541 IBM PC slowed programs. In the 1990s, .asp pages did the same thing. In the
542 2000s, JavaScript code and other code that needs to be sent across network
543 connections presents a similar problem.

544 In each of these cases, the ultimate solution has not been to avoid commenting. It
545 has been to create a release version of the code that's different from the
546 development version. This is typically accomplished by running the code
547 through a tool that strips out comments as part of the build process.

548 **Optimum Number of Comments**

549 **HARD DATA**
550 Capers Jones points out that studies at IBM found that a commenting density of
551 one comment roughly every ten statements was the density at which clarity
552 seemed to peak. Fewer comments made the code hard to understand. More
comments also reduced code understandability (Jones 2000).

553 This kind of research can be abused, and projects sometimes adopt a standard
554 such as "programs must have one comment at least every five lines." This
555 standard addresses the symptom of programmers' not writing clear code, but it
556 doesn't address the cause.

557 If you use the Pseudocode Programming Process effectively, you'll probably end
558 up with a comment for every few lines of code. The number of comments,

559 however, will be a side effect of the process itself. Rather than focusing on the
560 number of comments, focus on whether each comment is efficient. If the
561 comments describe why the code was written and meet the other criteria
562 established in this chapter, you'll have enough comments.

563 32.5 Commenting Techniques

564 Commenting is amenable to several different techniques depending on the level
565 to which the comments apply: program, file, routine, paragraph, or individual
566 line.

567 Commenting Individual Lines

568 In good code, the need to comment individual lines of code is rare. Here are two
569 possible reasons a line of code would need a comment:

- 570
- The single line is complicated enough to need an explanation.
 - The single line once had an error and you want a record of the error.

572 Here are some guidelines for commenting a line of code:

573 *Avoid self-indulgent comments*

574 Many years ago, I heard the story of a maintenance programmer who was called
575 out of bed to fix a malfunctioning program. The program's author had left the
576 company and couldn't be reached. The maintenance programmer hadn't worked
577 on the program before, and after examining the documentation carefully, he
578 found only one comment. It looked like this:

579 MOV AX, 723h ; R. I. P. L. V. B.

580 After working with the program through the night and puzzling over the
581 comment, the programmer made a successful patch and went home to bed.
582 Months later, he met the program's author at a conference and found out that the
583 comment stood for "Rest in peace, Ludwig van Beethoven." Beethoven died in
584 1827 (decimal), which is 723 (hexadecimal). The fact that 723h was needed in
585 that spot had nothing to do with the comment. Aaarrrrghhhh!

586 Endline Comments and Their Problems

587 Endline comments are comments that appear at the ends of lines of code. Here's
588 an example:

589 Visual Basic Example of Endline Comments

```
590 For employeeId = 1 To employeeCount
591     GetBonus( employeeId, employeeType, bonusAmount )
```

```
592     If employeeType = EmployeeType_Manager Then
593         PayManagerBonus( employeeId, bonusAmount ) ' pay full amount
594     Else
595         If employeeType = EmployeeType_Programmer Then
596             If bonusAmount >= MANAGER_APPROVAL_LEVEL Then
597                 PayProgrammerBonus( employeeId, StdAmt() ) ' pay std. amount
598             Else
599                 PayProgrammerBonus( employeeId, bonusAmount ) ' pay full amount
600             End If
601         End If
602     End If
603 Next
```

Although useful in some circumstances, endline comments pose several problems. The comments have to be aligned to the right of the code so that they don't interfere with the visual structure of the code. If you don't align them neatly, they'll make your listing look like it's been through the washing machine.

Endline comments tend to be hard to format. If you use many of them, it takes time to align them. Such time is not spent learning more about the code; it's dedicated solely to the tedious task of pressing the spacebar or the tab key.

Endline comments are also hard to maintain. If the code on any line containing an endline comment grows, it bumps the comment farther out, and all the other endline comments will have to be bumped out to match. Styles that are hard to maintain aren't maintained, and the commenting deteriorates under modification rather than improving.

Endline comments also tend to be cryptic. The right side of the line usually doesn't offer much room, and the desire to keep the comment on one line means that the comment must be short. Work then goes into making the line as short as possible instead of as clear as possible. The comment usually ends up as cryptic as possible.

621 *Avoid endline comments on single lines*

622 In addition to their practical problems, endline comments pose several
623 conceptual problems. Here's an example of a set of endline comments:

624

625 *The comments merely repeat
626 the code.*

C++ Example of Useless Endline Comments

```
627 memoryToInitialize = MemoryAvailable(); // get amount of memory available
628 pointer = GetMemory( memoryToInitialize ); // get a ptr to the available memory
629 ZeroMemory( pointer, memoryToInitialize ); // set memory to 0
630 ...
631 FreeMemory( pointer ); // free memory allocated
```

630
631
632

A systemic problem with endline comments is that it's hard to write a meaningful comment for one line of code. Most endline comments just repeat the line of code, which hurts more than it helps.

633
634
635

If an endline comment is intended to apply to more than one line of code, the formatting doesn't show which lines the comment applies to. Here's an example:

CODING HORROR

Visual Basic Example of a Confusing Endline Comment on Multiple Lines of Code

```
636 For rateIdx = 1 to rateCount           ' Compute discounted rates
637     LookupRegularRate( rateIdx, regularRate )
638     rate( rateIdx ) = regularRate * discount( rateIdx )
639 Next
```

Even though the content of this particular comment is fine, its placement isn't. You have to read the comment and the code to know whether the comment applies to a specific statement or to the entire loop.

When to Use Endline Comments

Here are three exceptions to the recommendation against using endline comments:

Use endline comments to annotate data declarations

Endline comments are useful for annotating data declarations because they don't have the same systemic problems as endline comments on code, provided that you have enough width. With 132 columns, you can usually write a meaningful comment beside each data declaration. Here's an example:

Java Example of Good Endline Comments for Data Declarations

```
645 int boundary;          // upper index of sorted part of array
646 String insertVal;    // data elmt to insert in sorted part of array
647 int insertPos;        // position to insert elmt in sorted part of array
```

Avoid using endline comments for maintenance notes

Endline comments are sometimes used for recording modifications to code after its initial development. This kind of comment typically consists of a date and the programmer's initials, or possibly an error-report number. Here's an example:

```
653 for i = 1 to maxElmts - 1 -- fixed error #A423 10/1/92 (scm)
654 Adding such a comment can be gratifying after a late-night debugging session on
655 software that's in production, but such comments really have no place in
656 production code. Such comments are handled better by version-control software.
657 Comments should explain why the code works now, not why the code didn't
658 work at some point in the past.
```

667 **CROSS-REFERENCE** Use
668 of endline comments to mark
669 ends of blocks is described
670 further in “Commenting
Control Structures,” later in
this section.

671
672
673

674

675
676

677

678

679

680

681

682

683

684

685

686

687

688

689

690

691

692

693

694

695

696

697

698

699

700

701

702

703

704

Use endline comments to mark ends of blocks

An endline comment is useful for marking the end of a long block of code—the end of a *while* loop or an *if* statement, for example. This is described in more detail later in this chapter.

Aside from a couple of special cases, endline comments have conceptual problems and tend to be used for code that’s too complicated. They are also difficult to format and maintain. Overall, they’re best avoided.

Commenting Paragraphs of Code

Most comments in a well-documented program are one-or two-sentence comments that describe paragraphs of code. Here’s an example:

Java Example of a Good Comment for a Paragraph of Code

```
// swap the roots
oldRoot = root[0];
root[0] = root[1];
root[1] = oldRoot;
```

The comment doesn’t repeat the code. It describes the code’s intent. Such comments are relatively easy to maintain. Even if you find an error in the way the roots are swapped, for example, the comment won’t need to be changed. Comments that aren’t written at the level of intent are harder to maintain.

Write comments at the level of the code’s intent

Describe the purpose of the block of code that follows the comment. Here’s an example of a comment that’s ineffective because it doesn’t operate at the level of intent:

Java Example of an Ineffective Comment

```
/* check each character in "inputString" until a dollar sign
is found or all characters have been checked
*/
done = False;
maxLen = inputString.length();
i = 0;
while ( !done && ( i < maxLen ) ) {
    if ( inputString[ i ] == '$' ) {
        done = True;
    }
    else {
        i++;
    }
}
```

705 You can figure out that the loop looks for a \$ by reading the code, and it's
706 somewhat helpful to have that summarized in the comment. The problem with
707 this comment is that it merely repeats the code and doesn't give you any insight
708 into what the code is supposed to be doing. This comment would be a little
709 better:

710 // find '\$' in inputString

711 This comment is better because it indicates that the goal of the loop is to find a \$.
712 But it still doesn't give you much insight into why the loop would need to find a
713 \$—in other words, into the deeper intent of the loop. Here's a comment that's
714 better still:

715 // find the command-word terminator (\$)

716 This comment actually contains information that the code listing does not,
717 namely that the \$ terminates a command word. In no way could you deduce that
718 merely from reading the code fragment, so the comment is genuinely helpful.

719 Another way of thinking about commenting at the level of intent is to think about
720 what you would name a routine that did the same thing as the code you want to
721 comment. If you're writing paragraphs of code that have one purpose each, it
722 isn't difficult. The comment in the code above is a good example.

723 *FindCommandWordTerminator()* would be a decent routine name. The other
724 options, *Find\$InInputString()* and

725 *CheckEachCharacterInInputStrUntilADollarSignIsFoundOrAllCharactersHave*
726 *BeenChecked()*, are poor names (or invalid) for obvious reasons. Type the
727 description without shortening or abbreviating it, as you might for a routine
728 name. That description is your comment, and it's probably at the level of intent.

729 If the code is a subset of another routine, take the next step and put the code into
730 its own routine. If it performs a well-defined function and you name the routine
731 well, you'll add to the readability and maintainability of your code.

732

733 **KEY POINT**

734 For the record, the code itself is always the first documentation you should
735 check. In the case above, the literal, \$, should be replaced with a named constant,
736 and the variables should provide more of a clue about what's going on. If you
737 want to push the edge of the readability envelope, add a variable to contain the
738 result of the search. Doing that clearly distinguishes between the loop index and
739 the result of the loop. Here's the code rewritten with good comments and good
style:

740 Java Example of a Good Comment and Good Code

```
// find the command-word terminator  
foundTheTerminator = False;
```

```
743     maxCommandLength = inputString.length();
744     testCharPosition = 0;
745     while ( !foundTheTerminator && ( testCharPosition < maxCommandLength ) ) {
746         if ( inputString[ testCharPosition ] == COMMAND_WORD_TERMINATOR ) {
747             foundTheTerminator = True;
748             Here's the variable that
749             contains the result of the
750             search.
751             terminatorPosition = testCharPosition;
752         }
753     }
```

If the code is good enough, it begins to read at close to the level of intent, encroaching on the comment's explanation of the code's intent. At that point, the comment and the code might become somewhat redundant, but that's a problem few programs have.

```
758     Another good step for this code would be to create a routine called something
759     like FindCommandWordTerminator() and move the code from the sample into
760     that routine. A comment that describes that thought is useful but is more likely
761     than a routine name to become inaccurate as the software evolves.
```

Focus paragraph comments on the why rather than the how

Comments that explain how something is done usually operate at the programming-language level rather than the problem level. It's nearly impossible for a comment that focuses on how an operation is done to explain the intent of the operation, and comments that tell how are often redundant. What does the following comment tell you that the code doesn't?

CODING HORROR

Java Example of a Comment That Focuses on How

```
// if account flag is zero
if ( accountFlag == 0 ) ...
```

The comment tells you nothing more than the code itself does. What about this comment?

Java Example of a Comment That Focuses on Why

```
// if establishing a new account
if ( accountFlag == 0 ) ...
```

This comment is a lot better because it tells you something you couldn't infer from the code itself. The code itself could still be improved by use of a meaningful enumerated type name instead of *O* and a better variable name. Here's the best version of this comment and code:

Java Example of Using Good Style In Addition to a "Why" Comment

```
// if establishing a new account
```

```
782     if ( accountType == AccountType.NewAccount ) ...
```

783 When code attains this level of readability, it's appropriate to question the value
784 of the comment. In this case, the comment has been made redundant by the
785 improved code, and it should probably be removed. Alternatively, the purpose of
786 the comment could be subtly shifted, like this:

787 Java Example of Using a “Section Heading” Comment

```
788 // establish a new account
789 if ( accountType == AccountType.NewAccount ) {
790     ...
791 }
```

792 If this comment documents the whole block of code following the *if* test, then it
793 serves as a summary-level comment, and it's appropriate to retain it as a section
794 heading for the paragraph of code it references.

795 *Use comments to prepare the reader for what is to follow*

796 Good comments tell the person reading the code what to expect. A reader should
797 be able to scan only the comments and get a good idea of what the code does and
798 where to look for a specific activity. A corollary to this rule is that a comment
799 should always precede the code it describes. This idea isn't always taught in
800 programming classes, but it's a well-established convention in commercial
801 practice.

802 *Make every comment count*

803 There's no virtue in excessive commenting. Too many comments obscure the
804 code they're meant to clarify. Rather than writing more comments, put the extra
805 effort into making the code itself more readable.

806 *Document surprises*

807 If you find anything that isn't obvious from the code itself, put it into a
808 comment. If you have used a tricky technique instead of a straightforward one to
809 improve performance, use comments to point out what the straightforward
810 technique would be and quantify the performance gain achieved by using the
811 tricky technique. Here's an example:

812 C++ Example of Documenting a Surprise

```
813 for ( element = 0; element < elementCount; element++ ) {
814     // Use right shift to divide by two. Substituting the
815     // right-shift operation cuts the loop time by 75%.
816     elementList[ element ] = elementList[ element ] >> 1;
817 }
```

818 The selection of the right shift in this example is intentional. Among experienced
819 programmers, it's common knowledge that for integers, right shift is functionally
820 equivalent to divide-by-two.

If it's common knowledge, why document it? Because the purpose of the operation is not to perform a right shift; it is to perform a divide-by-two. The fact that the code doesn't use the technique most suited to its purpose is significant. Moreover, most compilers optimize integer division-by-two to be a right shift anyway, meaning that the reduced clarity is usually unnecessary. In this particular case, the compiler evidently doesn't optimize the divide-by-two, and the time saved will be significant. With the documentation, a programmer reading the code would see the motivation for using the nonobvious technique. Without the comment, the same programmer would be inclined to grumble that the code is unnecessarily "clever" without any meaningful gain in performance. Usually such grumbling is justified, so it's important to document the exceptions.

Avoid abbreviations

Comments should be unambiguous, readable without the work of figuring out abbreviations. Avoid all but the most common abbreviations in comments.

Unless you're using endline comments, using abbreviations isn't usually a temptation. If you are, and it is, realize that abbreviations are another strike against a technique that struck out several pitches ago.

Differentiate between major and minor comments

In a few cases, you might want to differentiate between different levels of comments, indicating that a detailed comment is part of a previous, broader comment. You can handle this in a couple of ways.

You can try underlining the major comment and not underlining the minor comment, as in the following:

C++ Example of Differentiating Between Major and Minor Comments with Underlines—Not Recommended

```
// copy the string portion of the table, along the way omitting  
// strings that are to be deleted
```

```
-----  
// determine number of strings in the table
```

```
...  
...
```

```
...  
// mark the strings to be deleted
```

861 The weakness of this approach is that it forces you to underline more comments
862 than you'd really like to. If you underline a comment, it's assumed that all the
863 nonunderlined comments that follow it are subordinate to it. Consequently, when
864 you write the first comment that isn't subordinate to the underlined comment, it
865 too must be underlined and the cycle starts all over. The result is too much
866 underlining, or inconsistently underlining in some places and not underlining in
867 others.

868 This theme has several variations that all have the same problem. If you put the
869 major comment in all caps and the minor comments in lowercase, you substitute
870 the problem of too many all-caps comments for the problem of too many
871 underlined comments. Some programmers use an initial cap on major statements
872 and no initial cap on minor ones, but that's a subtle visual cue that's too easily
873 overlooked.

874 A better approach is to use ellipses in front of the minor comments. Here's an
875 example:

C++ Example of Differentiating Between Major and Minor Comments with Ellipses

878 *The major comment is
879 formatted normally.*

```
// copy the string portion of the table, along the way omitting  
// strings that are to be deleted
```

881 *A minor comment that is part
882 of the action described by the
883 major comment is preceded
884 by an ellipsis here...*

```
// ... determine number of strings in the table
```

885 ...

887 *...and here.*

```
// ... mark the strings to be deleted
```

888 ...

891 Another approach that's often best is to put the major-comment operation into its
892 own routine. Routines should be logically "flat," with all their activities on about
893 the same logical level. If your code differentiates between major and minor
894 activities within a routine, the routine isn't flat. Putting the complicated group of
895 activities into its own routine makes for two logically flat routines instead of one
896 logically lumpy one.

897 This discussion of major and minor comments doesn't apply to indented code
898 within loops and conditionals. In such cases, you'll often have a broad comment
899 at the top of the loop and more detailed comments about the operations within
900 the indented code. In those cases, the indentation provides the clue to the logical

901 organization of the comments. This discussion applies only to sequential
902 paragraphs of code in which several paragraphs make up a complete operation
903 and some paragraphs are subordinate to others.

904 ***Comment anything that gets around an error or an undocumented feature
905 in a language or an environment***

906 If it's an error, it probably isn't documented. Even if it's documented
907 somewhere, it doesn't hurt to document it again in your code. If it's an
908 undocumented feature, by definition it isn't documented elsewhere, and it should
909 be documented in your code.

910 Suppose you find that the library routine *WriteData(data, numItems, blockSize)*
911 works properly except when *blockSize* equals 500. It works fine for 499, 501,
912 and every other value you've ever tried, but you have found that the routine has a
913 defect that appears only when *blockSize* equals 500. In code that uses
914 *WriteData()*, document why you're making a special case when *blockSize* is 500.
915 Here's an example of how it could look:

916 Java Example of Documenting the Workaround for an Error

```
917 blockSize = optimalBlockSize( numItems, sizePerItem );  
918  
919 /* The following code is necessary to work around an error in  
920 WriteData() that appears only when the third parameter  
921 equals 500. '500' has been replaced with a named constant  
922 for clarity.  
923 */  
924 if ( blockSize == WRITEDATA_BROKEN_SIZE ) {  
925     blockSize = WRITEDATA_WORKAROUND_SIZE;  
926 }  
927 WriteData ( file, data, blockSize );
```

928 ***Justify violations of good programming style***

929 If you've had to violate good programming style, explain why. That will prevent
930 a well-intentioned programmer from changing the code to a better style, possibly
931 breaking your code. The explanation will make it clear that you knew what you
932 were doing and weren't just sloppy—give yourself credit where credit is due!

933 **FURTHER READING** For

934 other perspectives on writing
good comments, see *The*

935 **CODING HORROR**

936 *Style* (Kernighan and Plauger
937 1978).

938

939

940 // die a horrible death.

941 This is a good example of one of the most prevalent and hazardous bits of
942 programming folklore: that comments should be used to document especially
943 “tricky” or “sensitive” sections of code. The reasoning is that people should
944 know they need to be careful when they’re working in certain areas.

945 This is a scary idea.

946 Commenting tricky code is exactly the wrong approach to take. Comments can’t
947 rescue difficult code. As Kernighan and Plauger emphasize, “Don’t document
948 bad code—rewrite it” (1978).

949 **HARD DATA**

950 One study found that areas of source code with large numbers of comments also
951 tended to have the most defects and to consume the most development effort
952 (Lind and Vairavan 1989). The authors hypothesized that programmers tended to
comment difficult code heavily.

953 **KEY POINT**

954 When someone says, “This is really *tricky* code,” I hear them say, “This is really
955 *bad* code.” If something seems tricky to you, it will be incomprehensible to
956 someone else. Even something that doesn’t seem all that tricky to you can seem
957 impossibly convoluted to another person who hasn’t seen the trick before. If you
958 have to ask yourself, “Is this tricky?”, it is. You can always find a rewrite that’s
959 not tricky, so rewrite the code. Make your code so good that you don’t need
comments, and then comment it to make it even better.

960 This advice applies mainly to code you’re writing for the first time. If you’re
961 maintaining a program and don’t have the latitude to rewrite bad code,
962 commenting the tricky parts is a good practice.

963

964 **CROSS-REFERENCE** For
965 details on formatting data,
966 see "Laying Out Data
967 Declarations" in Section 31.5.
968 For details on how to use data
969 effectively, see Chapters 10
through 13.

970

971

972

973

974

975

976

Commenting Data Declarations

Comments for variable declarations describe aspects of the variable that the variable name can’t describe. It’s important to document data carefully; at least one company that has studied its own practices has concluded that annotations on data are even more important than annotations on the processes in which the data is used (SDC, in Glass 1982). Here are some guidelines for commenting data:

Comment the units of numeric data

If a number represents length, indicate whether the length is expressed in inches, feet, meters, or kilometers. If it’s time, indicate whether it’s expressed in elapsed seconds since 1-1-1980, milliseconds since the start of the program, and so on. If it’s coordinates, indicate whether they represent latitude, longitude, and altitude and whether they’re in radians or degrees; whether they represent an X, Y, Z coordinate system with its origin at the earth’s center; and so on. Don’t assume

977
978
979

980

981 **CROSS-REFERENCE** A
982 stronger technique for
983 documenting allowable
984 ranges of variables is to use
985 assertions at the beginning
986 and end of a routine to assert
987 that the variable's values
988 should be within a prescribed
range. For more details, see
Section 8.2, "Assertions."

989

990

991

992

993

994

995

996

997

998

999

1000

1001

1002

1003

1004

1005

1006

1007

1008

1009

1010

1011

1012

1013

1014

1015

that the units are obvious. To a new programmer, they won't be. To someone who's been working on another part of the system, they won't be. After the program has been substantially modified, they won't be.

Comment the range of allowable numeric values

If a variable has an expected range of values, document the expected range. One of the powerful features of the Ada programming language was the ability to restrict the allowable values of a numeric variable to a range of values. If your language doesn't support that capability (which most languages don't), use a comment to document the expected range of values. For example, if a variable represents an amount of money in dollars, indicate that you expect it to be between \$1 and \$100. If a variable indicates a voltage, indicate that it should be between 105v and 125v.

Comment coded meanings

If your language supports enumerated types—as C++ and Visual Basic do—use them to express coded meanings. If it doesn't, use comments to indicate what each value represents—and use a named constant rather than a literal for each of the values. If a variable represents kinds of electrical current, comment the fact that 1 represents alternating current, 2 represents direct current, and 3 represents *undefined*.

Here's an example of documenting variable declarations that illustrates the three preceding recommendations:

Visual Basic Example of Nicely Documented Variable Declarations

```
Dim cursorX As Integer      ' horizontal cursor position; ranges from 1..MaxCols
Dim cursorY As Integer      ' vertical cursor position; ranges from 1..MaxRows

Dim antennaLength As Long    ' length of antenna in meters; range is >= 2
Dim signalStrength As Integer ' strength of signal in kilowatts; range is >= 1

Dim characterCode As Integer   ' ASCII character code; ranges from 0..255
Dim characterAttribute As Integer ' 0=Plain; 1=Italic; 2=Bold; 3=BoldItalic
Dim characterSize As Integer    ' size of character in points; ranges from 4..127

All the range information is given in comments.
```

Comment limitations on input data

Input data might come from an input parameter, a file, or direct user input. The guidelines above apply as much to routine-input parameters as to other kinds of data. Make sure that expected and unexpected values are documented.

Comments are one way of documenting that a routine is never supposed to receive certain data. Assertions are another way to document valid ranges, and if you use them the code becomes that much more self-checking.

1016
1017
10181019 **CROSS-REFERENCE** For
1020 details on naming flag
1021 variables, see "Naming Status
1022 Variables" in Section 11.2.1023
1024
1025
1026
1027
1028
1029
10301031
1032
1033
1034
1035
10361037 **CROSS-REFERENCE** For
1038 details on using global data,
1039 see Section 13.3, "Global
1040 Data."1041
1042
1043

1044

1045 **CROSS-REFERENCE** For
1046 other details on control
1047 structures, see Section 31.3,
1048 "Layout Styles," Section
31.4, "Laying Out Control
1049 Structures," and Chapters 14
through 19.1050 *Purpose of the following loop*
1051
1052
1053***Document flags to the bit level***

If a variable is used as a bit field, document the meaning of each bit, as in the next example.

Visual Basic Example of Documenting Flags to the Bit Level

```
' The meanings of the bits in StatusFlags are as follows:  
' MSB 0 error detected: 1=yes, 0=no  
' 1-2 kind of error: 0=syntax, 1=warning, 2=severe, 3=fatal  
' 3 reserved (should be 0)  
' 4 printer status: 1=ready, 0=not ready  
' ...  
' 14 not used (should be 0)  
' LSB 15-32 not used (should be 0)  
Dim StatusFlags As Integer
```

If the example were written in C++, it would call for bit-field syntax so that the bit-field meanings would be self-documenting.

Stamp comments related to a variable with the variable's name

If you have comments that refer to a specific variable, make sure that the comment is updated whenever the variable is updated. One way to improve the odds of a consistent modification is to stamp the comment with the name of the variable. That way, string searches for the variable name will find the comment as well as the variable.

Document global data

If global data is used, annotate each piece well at the point at which it is declared. The annotation should indicate the purpose of the data and why it needs to be global. At each point at which the data is used, make it clear that the data is global. A naming convention is the first choice for highlighting a variable's global status. If a naming convention isn't used, comments can fill the gap.

Commenting Control Structures

The space before a control structure is usually a natural place to put a comment. If it's an *if* or a *case* statement, you can provide the reason for the decision and a summary of the outcome. If it's a loop, you can indicate the purpose of the loop. Here are a couple of examples:

C++ Example of Commenting the Purpose of a Control Structure

```
// copy input field up to comma  
while ( (*inputString != ',') && (*inputString != END_OF_STRING) ) {  
    *field = *inputString;  
    field++;
```

```

1054           inputString++;
1055     } // End of the loop (useful for
1056     longer, nested loops—
1057   although the need for such a
1058   comment indicates overly
1059   Purpose of the conditional
1060
1061   Purpose of the loop. Position
1062   of comment makes it clear
1063   that inputString is being set up
1064   for the loop.
1065
1066
1067
    *field = END_OF_STRING;

    // if at end of string, all actions are complete
    if ( *inputString != END_OF_STRING ) {
      // read past comma and subsequent blanks to get to the next input field
      inputString++;
      while ( ( *inputString == ' ' ) && ( *inputString != END_OF_STRING ) ) {
        inputString++;
      }
    } // if -- at end of string
  
```

This example suggests some guidelines.

Put a comment before each block of statements, if, case, or loop

Such a place is a natural spot for a comment, and these constructs often need explanation. Use a comment to clarify the purpose of the control structure.

Comment the end of each control structure

Use a comment to show what ended—for example,

```

1073   } // for clientIndex – process record for each client
1074
1075   A comment is especially helpful at the end of long or nested loops. Use
1076   comments to clarify loop nesting. Here's a Java example of using comments to
1077   clarify the ends of loop structures:
  
```

Java Example of Using Comments to Show Nesting

```

1078   for ( tableIndex = 0; tableIndex < tableCount; tableIndex++ ) {
1079     while ( recordIndex < recordCount ) {
1080       if ( !IllegalRecordNumber( recordIndex ) ) {
1081         ...
1082       } // if
1083     } // while
1084   } // for
  
```

This commenting technique supplements the visual clues about the logical structure given by the code's indentation. You don't need to use the technique for short loops that aren't nested. When the nesting is deep or the loops are long, however, the technique pays off.

Treat end-of-loop comments as a warning indicating complicated code

If a loop is complicated enough to need an end-of-loop comment, treat the comment as a warning sign: the loop might need to be simplified. The same rule applies to complicated *if* tests and *case* statements.

1093
1094
1095
1096

1097

1098 **CROSS-REFERENCE** For
1099 details on formatting
1100 routines, see Section 31.7,
1101 "Laying Out Routines." For
1102 details on how to create high-
1103 quality routines, see Chapter
CODING HORROR

1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133

End-of-loop comments provide useful clues to logical structure, but writing them initially and then maintaining them can become tedious. The best way to avoid such tedious work is often to rewrite any code that's complicated enough to require tedious documentation.

Commenting Routines

Routine-level comments are the subject of some of the worst advice in typical computer-science textbooks. Many textbooks urge you to pile up a stack of information at the top of every routine, regardless of its size or complexity. Here's an example:

Visual Basic Example of a Monolithic, Kitchen-Sink Routine Prolog

```
'*****  
' Name: CopyString  
'  
' Purpose: This routine copies a string from the source  
' string (source) to the target string (target).  
'  
' Algorithm: It gets the length of "source" and then copies each  
' character, one at a time, into "target". It uses  
' the loop index as an array index into both "source"  
' and "target" and increments the loop/array index  
' after each character is copied.  
'  
' Inputs:      input    The string to be copied  
'  
' Outputs:     output   The string to receive the copy of "input"  
'  
' Interface Assumptions: None  
'  
' Modification History: None  
'  
' Author:      Dwight K. Coder  
' Date Created: 10/1/04  
' Phone:       (555) 222-2255  
' SSN:         111-22-3333  
' Eye Color:   Green  
' Maiden Name: None  
' Blood Type: AB-  
' Mother's Maiden Name: None  
' Favorite Car: Pontiac Aztek  
' Personalized License Plate: "Tek-ie"  
'*****
```

1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144

This is ridiculous. *CopyString* is presumably a trivial routine—probably fewer than five lines of code. The comment is totally out of proportion to the scale of the routine. The parts about the routine's *Purpose* and *Algorithm* are strained because it's hard to describe something as simple as *CopyString* at a level of detail that's between "copy a string" and the code itself. The boiler-plate comments *Interface Assumptions* and *Modification History* aren't useful either—they just take up space in the listing. Requiring the author's name is redundant with information that can be retrieved more accurately from the revision control system. To require all these ingredients for every routine is a recipe for inaccurate comments and maintenance failure. It's a lot of make-work that never pays off.

1145
1146
1147
1148
1149

Another problem with heavy routine headers is that they discourage good factoring of the code—the overhead to create a new routine is so high that programmers will tend to err on the side of creating fewer routines, not more. Coding conventions should encourage good practices; heavy routine headers do the opposite.

1150

Here are some guidelines for commenting routines:

1151
1152
1153
1154
1155
1156

Keep comments close to the code they describe

One reason that the prolog to a routine shouldn't contain voluminous documentation is that such a practice puts the comments far away from the parts of the routine they describe. During maintenance, comments that are far from the code tend not to be maintained with the code. The comments and the code start to disagree, and suddenly the comments are worthless.

1157
1158
1159

Instead, follow the Principle of Proximity and put comments as close as possible to the code they describe. They're more likely to be maintained, and they'll continue to be worthwhile.

1160
1161
1162
1163

Several components of routine prologs are described below and should be included as needed. For your convenience, create a boilerplate documentation prolog. Just don't feel obliged to include all the information in every case. Fill out the parts that matter and delete the rest.

1164 **CROSS-REFERENCE** Goo
1165 d routine names are key to
1166 routine documentation. For
1167 details on how to create them,
see Section 7.3, "Good
1168 Routine Names."
1169

Describe each routine in one or two sentences at the top of the routine

If you can't describe the routine in a short sentence or two, you probably need to think harder about what it's supposed to do. Difficulty in creating a short description is a sign that the design isn't as good as it should be. Go back to the design drawing board and try again. The short summary statement should be present in virtually all routines except for simple *Get* and *Set* accessor routines.

1170
1171
1172

Document parameters where they are declared

The easiest way to document input and output variables is to put comments next to the parameter declarations. Here's an example:

1173
1174
1175
1176
1177
1178
1179

1180 **CROSS-REFERENCE** Endline
1181 line comments are discussed
1182 in more detail in "Endline
1183 comments and their
1184 problems," earlier in this
1185 section.

1186
1187
1188
1189
1190
1191
1192
1193

Java Example of Documenting Input and Output Data Where It's Declared—Good Practice

```
public void InsertionSort(  
    int[] dataToSort, // elements to sort in locations firstElement..lastElement  
    int firstElement, // index of first element to sort (>=0)  
    int lastElement // index of last element to sort (<= MAX_ELEMENTS)  
)
```

This practice is a good exception to the rule of not using endline comments; they are exceptionally useful in documenting input and output parameters. This occasion for commenting is also a good illustration of the value of using standard indentation rather than endline indentation for routine parameter lists; you wouldn't have room for meaningful endline comments if you used endline indentation. The comments in the example are strained for space even with standard indentation. This example also demonstrates that comments aren't the only form of documentation. If your variable names are good enough, you might be able to skip commenting them. Finally, the need to document input and output variables is a good reason to avoid global data. Where do you document it? Presumably, you document the globals in the monster prolog. That makes for more work and unfortunately in practice usually means that the global data doesn't get documented. That's too bad because global data needs to be documented at least as much as anything else.

Differentiate between input and output data

It's useful to know which data is used as input and which is used as output. Visual Basic makes it relatively easy to tell because output data is preceded by the *ByRef* keyword and input data is preceded by the *ByVal* keyword. If your language doesn't support such differentiation automatically, put it into comments. Here's an example in C++:

1200 **CROSS-REFERENCE** The
1201 order of these parameters
1202 follows the standard order for
1203 C++ routines but conflicts
1204 with more general practices.
1205 For details, see "Put
1206 parameters in input-modify-
1207 output order" in Section 7.5.
1208 For details on using a naming
1209 convention to differentiate
between input and output
data, see Section 11.4,
"Informal Naming
1210 Conventions."
1211
1212
1213

1214 **CROSS-REFERENCE** For
1215 details on other
1216 considerations for routine
1217 interfaces, see Section 7.5,
1218 "How to Use Routine
1219 Parameters."
1220

1221
1222
1223

1224
1225

1226
1227
1228
1229
1230
1231
1232
1233

1234
1235
1236
1237

C++ Example of Differentiating Between Input and Output Data

```
void StringCopy(  
    char *target,      // out: string to copy to  
    char *source       // in: string to copy from  
)  
{  
    ...
```

C++-language routine declarations are a little tricky because some of the time the asterisk (*) indicates that the argument is an output argument, and a lot of the time it just means that the variable is easier to handle as a pointer than as a base type. You're usually better off identifying input and output arguments explicitly.

If your routines are short enough and you maintain a clear distinction between input and output data, documenting the data's input or output status is probably unnecessary. If the routine is longer, however, it's a useful service to anyone who reads the routine.

Document interface assumptions

Documenting interface assumptions might be viewed as a subset of the other commenting recommendations. If you have made any assumptions about the state of variables you receive—legal and illegal values, arrays being in sorted order, member data being initialized or containing only good data, and so on—document them either in the routine prolog or where the data is declared. This documentation should be present in virtually every routine.

Make sure that global data that's used is documented. A global variable is as much an interface to a routine as anything else and is all the more hazardous because it sometimes doesn't seem like one.

As you're writing the routine and realize that you're making an interface assumption, write it down immediately.

Comment on the routine's limitations

If the routine provides a numeric result, indicate the accuracy of the result. If the computations are undefined under some conditions, document the conditions. If the routine has a default behavior when it gets into trouble, document the behavior. If the routine is expected to work only on arrays or tables of a certain size, indicate that. If you know of modifications to the program that would break the routine, document them. If you ran into gotchas during the development of the routine, document them too.

Document the routine's global effects

If the routine modifies global data, describe exactly what it does to the global data. As mentioned in Section 13.3, modifying global data is at least an order of magnitude more dangerous than merely reading it, so modifications should be

1238 performed carefully, part of the care being clear documentation. As usual, if
1239 documenting becomes too onerous, rewrite the code to reduce the use of global
1240 data.

1241 ***Document the source of algorithms that are used***

1242 If you have used an algorithm from a book or magazine, document the volume
1243 and page number you took it from. If you developed the algorithm yourself,
1244 indicate where the reader can find the notes you've made about it.

1245 ***Use comments to mark parts of your program***

1246 Some programmers use comments to mark parts of their program so that they
1247 can find them easily. One such technique in C++ and Java is to mark the top of
1248 each routine with a comment such as

1249 `/**`

1250 This allows you to jump from routine to routine by doing a string search for `/**`.

1251 A similar technique is to mark different kinds of comments differently,
1252 depending on what they describe.

1253 For example, in C++ you could use `@keyword`, where *keyword* is a code you use
1254 to indicate the kind of comment. The comment `@param` could indicate that the
1255 comment describes a parameter to a routine, `@version` could indicate file-version
1256 information, `@throws` could document the exceptions thrown by a routine, and
1257 so on. This technique allows you to use tools to extract different kinds of
1258 information from your source files. For example, you could search for `@throws`
1259 to retrieve documentation about all of the exceptions thrown by all of the
1260 routines in a program.

1261 CC2E.COM/3259
1262 This C++ convention is based on the JavaDoc convention, which is a well-
1263 established interface documentation convention for Java programs
1264 (java.sun.com/j2se/javadoc/). You can define your own conventions in other
languages.

1265

Commenting Classes, Files, and Programs

1266 **CROSS-REFERENCE** For
1267 layout details, see Section
1268 31.8, "Laying Out Classes."
1269 For details on using classes,
1270 see Chapter 6, "Working
1271 Classes."
1272 Classes as well.

Classes, files, and programs are all characterized by the fact that they contain multiple routines. A file or class should contain a collection of related routines. A program contains all the routines in a program. The documentation task in each case is to provide a meaningful, top-level view of the contents of the file, class, or program. The issues are similar in each case, so I'll just refer to documenting "files," and you can assume that the guidelines apply to classes and programs as well.

1273

1274

1275

1276

1277

1278

1279

1280

1281

1282

1283

1284

1285

1286

1287

1288

1289

1290

1291

1292

1293

1294

1295

1296

1297

1298

1299

1300

1301

1302

1303

1304

1305

1306

1307

General Guidelines for Class Documentation

For each class, use a block comment to describe general attributes of the class.

Describe the design approach to the class

Overview comments that provide information that can't readily be reverse engineered from coding details are especially useful. Describe the class's design philosophy, overall design approach, design alternatives that were considered and discarded, and so on.

Describe limitations, usage assumptions, and so on

Similar to routines, be sure to describe any limitations imposed by the class's design. Also describe assumptions about input and output data, error-handling responsibilities, global effects, sources of algorithms, and so on.

Comment the class interface

Can another programmer understand how to use a class without looking at the class's implementation? If not, then class encapsulation is seriously at risk. The class's interface should contain all the information anyone needs to use the class. The JavaDoc convention is to require, at a minimum, documentation for each parameter and each return value (Sun Microsystems 2000). This should be done for all exposed routines of each class (Bloch 2001).

Don't document implementation details in the class interface

A cardinal rule of encapsulation is that you expose information only on a need-to-know basis: if there is any question about whether information needs to be exposed, the default is to keep it hidden. Consequently, class interface files should contain information needed to use the class, but not information needed to implement or maintain the inner workings of the class.

General Guidelines for File Documentation

At the top of a file, use a block comment to describe the contents of the file.

Here are some guidelines for the block comment:

Describe the purpose and contents of each file

The file header comment should describe the classes or routines contained in a file. If all the routines for a program are in one file, the purpose of the file is pretty obvious—it's the file that contains the whole program. If the purpose of the file is to contain one specific class, the purpose is also pretty obvious—it's the file that contains the class with a similar name.

If the file contains more than one class, explain why the classes need to be combined into a single file.

1308 If the division into multiple source files is made for some reason other than
1309 modularity, a good description of the purpose of the file will be even more
1310 helpful to a programmer who is modifying the program. If someone is looking
1311 for a routine that does *x*, does the file's header comment help that person
1312 determine whether this file contains such a routine?

1313 ***Put your name, email address, and phone number in the block comment***
1314 Authorship and primary responsibility for specific areas of source code becomes
1315 important on large projects. Small projects (less than 10 people) can use
1316 collaborative development approaches such as shared code ownership in which
1317 all team members are equally responsible for all sections of code. Larger systems
1318 require that programmers specialize in different areas of code, which makes full-
1319 team-wide shared code ownership impractical.

1320 In that case, authorship is important information to have in a listing. It gives
1321 other programmers who work on the code a clue about the programming style,
1322 and it gives them someone to contact if they need help. Depending on whether
1323 you work on individual routines, classes, or programs, you should include author
1324 information at the routine, class, or program level.

1325 ***Include a copyright statement in the block comment***
1326 Many companies like to include copyright statements in their programs. If yours
1327 is one of them, include a line similar to this one:

1328 **Java Example of a Copyright Statement**

```
1329 // (c) Copyright 1993-2004 Steven C. McConnell. All Rights Reserved.  
1330 ...  
1331 (You would typically use your company's name rather than your name.)
```

1332 ***Give the file a name related to its contents***
1333 Normally, the name of the file should be closely related to the name of the public
1334 class contained in the file. For example, if the class is named *Employee*, the file
1335 should be named *Employee.cpp*.

1336

1337 **FURTHER READING** This
1338 discussion is adapted from
1339 "The Book Paradigm for
1340 Improved Maintenance"
1340 (Oman and Cook 1990a) and
1341 "Typographic Style Is More
1341 Than Cosmetic" (Oman and
1342 Cook 1990b). A similar
1343 analysis is presented in detail
1344 in *Human Factors and*
1345 *Typography for More*
1346 *Readable Programs* (Baecker
1346 and Marcus 1990).

1347

1348

1349

1350

1351

1352

1353

1354

1355

1356

1357

1358

1359

1360

1361

1362

1363

1364

1365 | HARD DATA

1366

1367

1368

1369

1370

1371

The Book Paradigm for Program Documentation

Most experienced programmers agree that the documentation techniques described in the previous section are valuable. The hard, scientific evidence for the value of any one of the techniques is still weak. When the techniques are combined, however, evidence of their effectiveness is strong.

In 1990, Paul Oman and Curtis Cook published a pair of studies on the "Book Paradigm" for documentation (1990a, 1990b). They looked for a coding style that would support several different styles of code reading. One goal was to support top-down, bottom-up, and focused searches. Another was to break up the code into chunks that programmers could remember more easily than a long listing of homogeneous code. Oman and Cook wanted the style to provide for both high-level and low-level clues about code organization.

They found that by thinking of code as a special kind of book and formatting it accordingly, they could achieve their goals. In the Book Paradigm, code and its documentation are organized into several components similar to the components of a book to help programmers get a high-level view of the program.

The "preface" is a group of introductory comments such as those usually found at the beginning of a file. It functions as the preface to a book does. It gives the programmer an overview of the program.

The "table of contents" shows the files, classes, and routines (chapters). They might be shown in a list, as a traditional book's chapters are, or graphically, in a structure chart.

The "sections" are the divisions within routines—routine declarations, data declarations, and executable statements, for example.

The "cross-references" are cross-reference maps of the code, including line numbers.

The low-level techniques that Oman and Cook use to take advantage of the similarities between a book and a code listing are similar to the techniques described in Chapter 31, "Layout and Style," and in this chapter.

The upshot of using their techniques to organize code was that when Oman and Cook gave a maintenance task to a group of experienced, professional programmers, the average time to perform a maintenance task in a 1000-line program was only about three-quarters of the time it took the programmers to do the same task in a traditional source listing (1990b). Moreover, the maintenance scores of programmers on code documented with the Book Paradigm averaged about 20 percent higher than on traditionally documented code. Oman and Cook

1372 concluded that by paying attention to the typographic principles of book design,
1373 you can get a 10 to 20 percent improvement in comprehension. A study with
1374 programmers at the University of Toronto produced similar results (Baecker and
1375 Marcus 1990).

1376 The Book Paradigm emphasizes the importance of providing documentation that
1377 explains both the high-level and the low-level organization of your program.

1378 22.6 IEEE Standards

1379 One of the most valuable sources of information on documenting software
1380 projects is contained the IEEE Software Engineering Standards. IEEE standards
1381 are developed by groups composed of practitioners and academicians who are
1382 expert in a particular area. Each standard contains a summary of the area covered
1383 by the standard and typically contains the outline for the appropriate
1384 documentation for work in that area.

1385 Several national and international organizations participate in standards work.
1386 The *IEEE* (Institute for Electric and Electrical Engineers) is a group that has
1387 taken the lead in defining software engineering standards. Some standards are
1388 jointly adopted by *ISO* (International Standards Organization), *EIA* (Electronic
1389 Industries Alliance), *IEC* (International Engineering Consortium), or both.

1390 Standards names are composed of the standards number, the year the standard
1391 was adopted, and the name of the standard. So, *IEEE/EIA Std 12207-1997*,
1392 *Information Technology—Software Life Cycle Processes*, refers to standard
1393 number 12207.2, which was adopted in 1997 by the IEEE and EIA.

1394 Here are some of the national and international standards most applicable to
1395 software projects.

1396 CC2E.COM/3266 The top-level standard is *ISO/IEC Std 12207, Information Technology—Software*
1397 *Life Cycle Processes*, which is the international standard that defines a lifecycle
1398 framework for developing and managing software projects. This standard was
1399 adopted in the United States as *IEEE/EIA Std 12207, Information Technology—*
1400 *Software Life Cycle Processes*.

1401 Software Development Standards

1402 *IEEE Std 830-1998, Recommended Practice for Software Requirements*
1403 *Specifications*

1404 *IEEE Std 1233-1998, Guide for Developing System Requirements Specifications*

1405 *IEEE Std 1016-1998, Recommended Practice for Software Design Descriptions*

1406 *IEEE Std 828-1998, Standard for Software Configuration Management Plans*

1407 *IEEE Std 1063-2001, Standard for Software User Documentation*

1408 *IEEE Std 1219-1998, Standard for Software Maintenance*

1409 CC2E.COM/3280

Software Quality Assurance Standards

1410 *IEEE Std 730-2002, Standard for Software Quality Assurance Plans*

1411 *IEEE Std 1028-1997, Standard for Software Reviews*

1412 *IEEE Std 1008-1987 (R1993), Standard for Software Unit Testing*

1413 *IEEE Std 829-1998, Standard for Software Test Documentation*

1414 *IEEE Std 1061-1998, Standard for a Software Quality Metrics Methodology*

1415 CC2E.COM/3287

Management Standards

1416 *IEEE Std 1058-1998, Standard for Software Project Management Plans*

1417 *IEEE Std 1074-1997, Standard for Developing Software Life Cycle Processes*

1418 *IEEE Std 1045-1992, Standard for Software Productivity Metrics*

1419 *IEEE Std 1062-1998, Recommended Practice for Software Acquisition*

1420 *IEEE Std 1540-2001, Standard for Software Life Cycle Processes- Risk Management*

1422 *IEEE Std 1490-1998, Guide - Adoption of PMI Standard - A Guide to the Project Management Body of Knowledge*

1424 CC2E.COM/3294

Overview of Standards

1425 CC2E.COM/3201

IEEE Software Engineering Standards Collection, 2003 Edition. New York: Institute of Electrical and Electronics Engineers (IEEE). This comprehensive volume contains 40 of the most recent ANSI/IEEE standards for software development as of 2003. Each standard includes a document outline, a description of each component of the outline, and a rationale for that component. The document includes standards for quality-assurance plans, configuration-management plans, test documents, requirements specifications, verification and validation plans, design descriptions, project management plans, and user

1433 documentation. The book is a distillation of the expertise of hundreds of people
1434 at the top of their fields, and would be a bargain at virtually any price. Some of
1435 the standards are also available individually. All are available from the IEEE
1436 Computer Society in Los Alamitos, California and from
1437 www.computer.org/cspress.

1438 Moore, James W. *Software Engineering Standards: A User's Road Map*, Los
1439 Alamitos, Ca.: IEEE Computer Society Press, 1997. Moore provides an
1440 overview of IEEE software engineering standards.

CC2E.COM/3208

1441

1442 CC2E.COM/3215 *I wonder how many great
1443 novelists have never read
1444 someone else's work, how
1445 many great painters have
1446 never studied another's
1447 brush strokes, how many
1448 skilled surgeons never
1449 learned by looking over a
1450 colleague's shoulder ...
1451 And yet that's what we
1452 expect programmers to
1453 do.*

1454 —Dave Thomas

1455

1456

1457

1458

1459

1460 CC2E.COM/3222

1461

1462

1463

1464

1465

1466

1467

Additional Resources on Documentation

SourceForge.net. For decades, a perennial problem in teaching software development has been finding lifesize examples of production code to share with students. Many people learn quickest from studying real-life examples, but most lifesize code bases are treated as proprietary information by the companies that created them. This situation has improved dramatically through the combination of the Internet and open source software. The Source Forge website contains code for thousands of programs in C, C++, Java, Visual Basic, PHP, Perl, Python, and many other languages, all which you can download for free.

Programmers can benefit from wading through the code on this website to see much larger real-world examples than *Code Complete* is able to show in the short code examples in this book. Junior programmers who haven't previously seen extensive examples of production code will find this website especially valuable.

Spinellis, Diomidis. *Code Reading: The Open Source Perspective*, Boston, Mass.: Addison Wesley, 2003. This book is a pragmatic exploration of techniques for reading code—including where to find code to read, tips for reading large code bases, tools that support code reading, and many other useful suggestions.

Sun Microsystems. "How to Write Doc Comments for the Javadoc™ Tool," 2000. Available from <http://java.sun.com/j2se/javadoc/writingdoccomments/>. This article describes how to use Javadoc to document Java programs. It includes detailed advice about how to tag comments using an @tag style notation. It also includes many specific details about how to wordsmith the comments themselves. The Javadoc conventions are probably the most fully developed code-level documentation standards currently available.

Here are sources of information on other topics in software documentation:

1468 **CROSS-REFERENCE** For
1469 Additional Resources on
1470 programming style, see the
1471 references in “Additional
Resources” in Chapter 31.

1472 CC2E.COM/3229

1473

1474

1475 CC2E.COM/3236

1476

1477

1478

1479 CC2E.COM/3243

1480

1481

1482

1483

1484

1485

1486

1487

1488

1489

1490

1491

1492

1493

1494

1495

1496

1497

1498

1499

1500

1501

McConnell, Steve. *Software Project Survival Guide*, Redmond, Wa: Microsoft Press, 1998. This book describes the documentation required by a medium-sized business-critical project. A related website provides numerous related document templates.

www.construx.com. This website (my company’s website) contains numerous document templates, coding conventions, and other resources related to all aspects of software development, including software documentation.

Post, Ed. “Real Programmers Don’t Use Pascal”, *Datamation*, July 1983, pp. 263-265. This tongue-in-cheek paper argues for a return to the “good old days” of Fortran programming when programmers didn’t have to worry about pesky issues like readability.

CHECKLIST: Good Commenting Technique

General

- Can someone pick up the code and immediately start to understand it?
- Do comments explain the code’s intent or summarize what the code does, rather than just repeating the code?
- Is the Pseudocode Programming Process used to reduce commenting time?
- Has tricky code been rewritten rather than commented?
- Are comments up to date?
- Are comments clear and correct?
- Does the commenting style allow comments to be easily modified?

Statements and Paragraphs

- Does the code avoid endline comments?
- Do comments focus on *why* rather than *how*?
- Do comments prepare the reader for the code to follow?
- Does every comment count? Have redundant, extraneous, and self-indulgent comments been removed or improved?
- Are surprises documented?
- Have abbreviations been avoided?
- Is the distinction between major and minor comments clear?
- Is code that works around an error or undocumented feature commented?

Data Declarations

- Are units on data declarations commented?
- Are the ranges of values on numeric data commented?

- 1502 Are coded meanings commented?
- 1503 Are limitations on input data commented?
- 1504 Are flags documented to the bit level?
- 1505 Has each global variable been commented where it is declared?
- 1506 Has each global variable been identified as such at each usage, by a naming convention, a comment, or both?
- 1507 Are magic numbers replaced with named constants or variables rather than just documented?
- 1508
- 1509

1510 **Control Structures**

- 1511 Is each control statement commented?
- 1512 Are the ends of long or complex control structures commented or, when possible, simplified so that they don't need comments?
- 1513

1514 **Routines**

- 1515 Is the purpose of each routine commented?
- 1516 Are other facts about each routine given in comments, when relevant, including input and output data, interface assumptions, limitations, error corrections, global effects, and sources of algorithms?
- 1517
- 1518

1519 **Files, Classes, and Programs**

- 1520 Does the program have a short document such as that described in the Book Paradigm that gives an overall view of how the program is organized?
- 1521 Is the purpose of each file described?
- 1522 Are the author's name, email address, and phone number in the listing?
- 1523
- 1524
-

1525 **Key Points**

- 1526 • The question of whether to comment is a legitimate one. Done poorly, commenting is a waste of time and sometimes harmful. Done well, commenting is worthwhile.
- 1527
- 1528
- 1529 • The source code should contain most of the critical information about the program. As long as the program is running, the source code is more likely than any other resource to be kept current, and it's useful to have important information bundled with the code.
- 1530
- 1531
- 1532
- 1533 • Good code is its own best documentation. If the code is bad enough to require extensive comments, try first to improve the code so that it doesn't need extensive comments.
- 1534
- 1535

- 1536 ● Comments should say things about the code that the code can't say about
1537 itself—at the summary level or the intent level.
1538
1539 ● Some commenting styles require a lot of tedious clerical work. Develop a
 style that's easy to maintain.

33

Personal Character

3

CC2E.COM/3313

4

5

6

7

8

9

10

11

12

Related Topics

Themes in software craftsmanship: Chapter 34

Complexity: Sections 5.2 and 19.6

PERSONAL CHARACTER HAS RECEIVED A RARE DEGREE of attention in software development. Ever since Edsger Dijkstra's landmark 1965 article "Programming Considered as a Human Activity," programmer character has been regarded as a legitimate and fruitful area of inquiry. Although titles such as *The Psychology of Bridge Construction* and "Exploratory Experiments in Attorney Behavior" might seem absurd, in the computer field *The Psychology of Computer Programming*, "Exploratory Experiments in Programmer Behavior," and similar titles are classics.

Engineers in every discipline learn the limits of the tools and materials they work with. If you're an electrical engineer, you know the conductivity of various metals and a hundred ways to use a voltmeter. If you're a structural engineer, you know the load-bearing properties of wood, concrete, and steel.

If you're a software engineer, your basic building material is human intellect and your primary tool is *you*. Rather than designing a structure to the last detail and then handing the blueprints to someone else for construction, you know that once you've designed a piece of software to the last detail, it's done. The whole job of

32 programming is building air castles—it's one of the most purely mental activities
33 you can do. Consequently, when software engineers study the essential
34 properties of their tools and raw materials, they find that they're studying
35 people—intellect, character, and other attributes that are less tangible than wood,
36 concrete, and steel.

37 If you're looking for concrete programming tips, this chapter might seem too
38 abstract to be useful. Once you've absorbed the specific advice in the rest of the
39 book, however, this chapter spells out what you need to do to continue
40 improving. Read the next section, and then decide whether you want to skip the
41 chapter.

42 **33.1 Isn't Personal Character Off the Topic?**

43 The intense inwardness of programming makes personal character especially
44 important. You know how difficult it is to put in eight concentrated hours in one
45 day. You've probably had the experience of being burned out one day from
46 concentrating too hard the day before, or burned out one month from
47 concentrating too hard the month before. You've probably had days on which
48 you've worked well from 8:00 A.M. to 2:00 P.M. and then felt like quitting. You
49 didn't quit, though; you pushed on from 2:00 P.M. to 5:00 P.M. and then spent
50 the rest of the week fixing what you wrote from 2:00 to 5:00.

51 Programming work is essentially unsupervisable because no one ever really
52 knows what you're working on. We've all had projects in which we spent 80
53 percent of the time working on a small piece we found interesting and 20 percent
54 of the time building the other 80 percent of the program.

55 Your employer can't force you to be a good programmer; a lot of times your
56 employer isn't even in a position to judge whether you're good. If you want to be
57 great, you're responsible for making yourself great. It's a matter of your personal
58 character.

59 **HARD DATA**

60 Once you decide to make yourself a superior programmer, the potential for
61 improvement is huge. Study after study has found differences on the order of 10
62 to 1 in the time required to create a program. They have also found differences
63 on the order of 10 to 1 in the time required to debug a program and 10 to 1 in the
64 resulting size, speed, error rate, and number of errors detected (Sackman,
65 Erikson, and Grant 1968; Curtis 1981; Mills 1983; DeMarco and Lister 1985;
Curtis et al. 1986; Card 1987; Valett and McGarry 1989).

66
67
68

You can't do anything about your intelligence, so the classical wisdom goes, but you can do something about your character. It turns out that character is the more decisive factor in the makeup of a superior programmer.

69

33.2 Intelligence and Humility

70 *We become authorities
71 and experts in the
72 practical and scientific
73 spheres by so many
74 separate acts and hours
75 of work. If a person keeps
76 faithfully busy each hour
77 of the working day, he
78 can count on waking up
79 some morning to find
80 himself one of the
81 competent ones of his
82 generation.*

—William James

83
84
85
86
87

Intelligence doesn't seem like an aspect of personal character, and it isn't. Coincidentally, great intelligence is only loosely connected to being a good programmer.

What? You don't have to be superintelligent?

No, you don't. Nobody is really smart enough to program computers. Fully understanding an average program requires an almost limitless capacity to absorb details and an equal capacity to comprehend them all at the same time. The way you focus your intelligence is more important than how much intelligence you have.

As Chapter 5 mentioned, at the 1972 Turing Award Lecture, Edsger Dijkstra delivered a paper titled “The Humble Programmer.” He argued that most of programming is an attempt to compensate for the strictly limited size of our skulls. The people who are best at programming are the people who realize how small their brains are. They are humble. The people who are the worst at programming are the people who refuse to accept the fact that their brains aren't equal to the task. Their egos keep them from being great programmers. The more you learn to compensate for your small brain, the better a programmer you'll be. The more humble you are, the faster you'll improve.

88
89

The purpose of many good programming practices is to reduce the load on your gray cells. Here are a few examples:

90
91
92
93
94
95
96
97
98
99

- The point of “decomposing” a system is to make it simpler to understand. (See Section TBD for more details.)
- Conducting reviews, inspections, and tests is a way of compensating for anticipated human fallibilities. These review techniques originated as part of “egoless programming” (Weinberg 1998). If you never made mistakes, you wouldn't need to review your software. But you know that your intellectual capacity is limited, so you augment it with someone else's.
- Keeping routines short reduces the load on your brain.
- Writing programs in terms of the problem domain rather than in terms of low-level implementation-level details reduces your mental workload.

- 100 • Using conventions of all sorts frees your brain from the relatively mundane
101 aspects of programming, which offer little payback.

102 You might think that the high road would be to develop better mental abilities so
103 that you wouldn't need these programming crutches. You might think that a
104 programmer who uses mental crutches is taking the low road. Empirically,
105 however, it's been shown that humble programmers who compensate for their
106 fallibilities write code that's easier for themselves and others to understand and
107 that has fewer errors. The real low road is the road of errors and delayed
108 schedules.

109 33.3 Curiosity

110 Once you admit that your brain is too small to understand most programs and
111 you realize that effective programming is a search for ways to offset that fact,
112 you begin a career-long search for ways to compensate.

113 In the development of a superior programmer, curiosity about technical subjects
114 must be a priority. The relevant technical information changes continually. Many
115 web programmers have never had to program in Windows, and many Windows
116 programmers never had to deal with DOS, or Unix, or punch cards. Specific
117 features of the technical environment change every 5 to 10 years. If you aren't
118 curious enough to keep up with the changes, you may find yourself down at the
119 old-programmers' home playing cards with T-Bone Rex and the Brontosaurus
120 sisters.

121 Programmers are so busy working they often don't have time to be curious about
122 how they might do their jobs better. If this is true for you, you're not alone. The
123 following subsections describe a few specific actions you can take to exercise
124 your curiosity and make learning a priority.

125 **CROSS-REFERENCE** For 126 a fuller discussion of the 127 importance of process in 128 software development, see 129 Section 34.2, "Pick Your 130 Process."

131 The more aware you are of the development process, whether from reading or
132 from your own observations about software development, the better position
133 you're in to understand changes and to move your group in a good direction.

134 If your workload consists entirely of short-term assignments that don't develop
135 your skills, be dissatisfied. If you're working in a competitive software market,
136 half of what you now need to know in order to do your job will be out of date in
137 three years. If you're not learning, you're turning into a dinosaur.

138 You're in too much demand to spend time working for management that doesn't
139 have your interests in mind. Despite some ups and downs, the average number of
140 software jobs available in the U.S. is expected to increase dramatically between

136
137
138
139
140
2000 and 2010. Jobs for systems analysts are expected to increase by 60 percent,
for software engineers by 95 percent, and for computer programmers by 16
percent. For all computer-job categories combined, about 2 million new jobs will
be created beyond the 2.9 million that currently exist (Hecker 2001). If you can't
learn at your job, find a new one.

141 **CROSS-REFERENCE** Several key aspects of
142 programming revolve around
143 the idea of experimentation.
144 For details, see
145 "Experimentation" in Section
146 34.9.
147

148
149
150
151
152

153 **FURTHER READING** A great
154 book that teaches problem
155 solving is James Adams's
156 *Conceptual Blockbusting*
(2001).

157
158
159

160
161
162
163
164
165

166
167 CC2E.COM/3320
168
169
170
171
172
173

Experiment

One effective way to learn about programming is to experiment with
programming and the development process. If you don't know how a feature of
your language works, write a short program to exercise the feature, and see how
it works. Prototype! Watch the program execute in the debugger. You're better
off working with a short program to test a concept than you are writing a larger
program with a feature you don't quite understand.

What if the short program shows that the feature doesn't work the way you want
it to? That's what you wanted to find out. Better to find it out in a small program
than a large one. One key to effective programming is learning to make mistakes
quickly, learning from them each time. Making a mistake is no sin. Failing to
learn from a mistake is.

Read about problem solving

Problem solving is the core activity in building computer software. Herbert
Simon reported a series of experiments on human problem solving. They found
that human beings don't always discover clever problem-solving strategies
themselves, even though the same strategies could readily be taught to the same
people (Simon 1996). The implication is that even if you want to reinvent the
wheel, you can't count on success. You might reinvent the square instead.

Analyze and plan before you act

You'll find that there's a tension between analysis and action. At some point you
have to quit gathering data and act. The problem for most programmers,
however, isn't an excess of analysis. The pendulum is currently so far on the
"acting" side of the arc that you can wait until it's at least partway to the middle
before worrying about getting stuck on the "analysis-paralysis" side.

Learn about successful projects

One especially good way to learn about programming is to study the work of the
great programmers. Jon Bentley thinks that you should be able to sit down with a
glass of brandy and a good cigar and read a program the way you would a good
novel. That might not be as far-fetched as it sounds. Most people wouldn't want
to use their recreational time to scrutinize a 500-page source listing, but many
people would enjoy studying a high-level design and dipping into more detailed
source listings for selected areas.

174 The software-engineering field makes extraordinarily limited use of examples of
175 past successes and failures. If you were interested in architecture, you'd study
176 the drawings of Louis Sullivan, Frank Lloyd Wright, and I. M. Pei. You'd
177 probably visit some of their buildings. If you were interested in structural
178 engineering, you'd study the Brooklyn bridge, the Tacoma Narrows bridge, and
179 a variety of other concrete, steel, and wood structures. You would study
180 examples of successes and failures in your field.

181 Thomas Kuhn points out that a part of any mature science is a set of solved
182 problems that are commonly recognized as examples of good work in the field
183 and serve as examples for future work (Kuhn 1996). Software engineering is
184 only beginning to mature to this level. In 1990, the Computer Science and
185 Technology Board concluded that there were few documented case studies of
186 either successes or failures in the software field (CSTB 1990). An article in the
187 *Communications of the ACM* argued for learning from case studies of
188 programming problems (Linn and Clancy 1992). The fact that someone has to
189 argue for this is significant.

190 That one of the most popular computing columns, "Programming Pearls," was
191 built around case studies of programming problems is suggestive. One of the
192 most popular books in software engineering is *The Mythical Man-Month*, a
193 postmortem on the IBM OS/360 project, a case study in programming
194 management.

195 With or without a book of case studies in programming, find code written by
196 superior programmers and read it. Ask to look at the code of programmers you
197 respect. Ask to look at the code of programmers you don't. Compare their code,
198 and compare their code to your own. What are the differences? Why are they
199 different? Which way is better? Why?

200 In addition to reading other people's code, develop a desire to know what expert
201 programmers think about your code. Find world-class programmers who'll give
202 you their criticism. As you listen to the criticism, filter out points that have to do
203 with their personal idiosyncrasies and concentrate on the points that matter. Then
204 change your programming so that it's better.

205 **Read!**

206 Documentation phobia is rampant among programmers. Computer
207 documentation tends to be poorly written and poorly organized, but for all its
208 problems, there's much to gain from overcoming an excessive fear of computer-
209 screen photons or paper products. Documentation contains the keys to the castle,
210 and it's worth spending time reading it. Overlooking information that's readily
211 available is such a common oversight that a familiar acronym on newsgroups
212 and bulletin boards is "RTFM!," which stands for "Read the !#*%*@ Manual!"

213
214
215
216
217

A modern language product is usually bundled with an enormous set of library code. Time spent browsing through the library documentation is well invested. Often the company that provides the language product has already created many of the classes you need. If it has, make sure you know about them. Skim the documentation every couple of months.

218 **CROSS-REFERENCE** For
219 books you can use in a
220 personal reading program,
221 see Section 35.4, "A
222 Software Developer's
223 Reading Plan."
224
225

226
227 **FURTHER READING** For
228 other discussions of
229 programmer levels, see
"Construx's Professional
230 Development Program"
231 (Chapter 16) in *Professional
Software Development*
232 (McConnell 2004).

233
234
235

236
237
238
239
240

241
242
243
244
245
246
247
248
249

250
251

Read other books and periodicals

Pat yourself on the back for reading this book. You're already learning more than most people in the software industry because one book is more than most programmers read each year (DeMarco and Lister 1999). A little reading goes a long way toward professional advancement. If you read even one good programming book every two months, roughly 35 pages a week, you'll soon have a firm grasp on the industry and distinguish yourself from nearly everyone around you.

Make a commitment to professional development

Good programmers constantly look for ways to become better. Consider the following professional development ladder used at my company and several others:

- Level 1: Beginning. A beginner is a programmer capable of using the basic capabilities of one language. Such a person can write classes, routines, loops, and conditionals and use many of the features of a language.
- Level 2: Introductory. An intermediate programmer who has moved past the beginner phase is capable of using the basic capabilities of multiple languages and is very comfortable in at least one language.
- Level 3: Competency. A competent programmer has expertise in a language or an environment or both. A programmer at this level might know all the intricacies of J2EE or have the *C++ Annotated C++ Reference Manual* memorized. Programmers at this level are valuable to their companies, and many programmers never move beyond this level.
- Level 4: Leadership. A leader has the expertise of a Level 3 programmer and recognizes that programming is only 15 percent communicating with the computer, that it's 85 percent communicating with people. Only 30 percent of an average programmer's time is spent working alone (McCue 1978). Even less time is spent communicating with the computer. The guru writes code for an audience of people rather than machines. True guru-level programmers write code that's crystal-clear, and they document it too. They don't want to waste their valuable gray cells reconstructing the logic of a section of code that they could have read in a one-sentence comment.

A great coder who doesn't emphasize readability is probably stuck at Level 3, but even that isn't usually the case. In my experience, the main reason people

252 write unreadable code is that their code is bad. They don't say to themselves,
253 "My code is bad, so I'll make it hard to read." They just don't understand their
254 code well enough to make it readable, which locks them into one of the lower
255 levels.

256 The worst code I've ever seen was written by someone who wouldn't let anyone
257 go near her programs. Finally, her manager threatened to fire her if she didn't
258 cooperate. Her code was uncommented and littered with variables like *x*, *xx*, *xxx*,
259 *xx1*, and *xx2*, all of which were global. Her manager's boss thought she was a
260 great programmer because she fixed errors quickly. The quality of her code gave
261 her abundant opportunities to demonstrate her error-correcting ability.

262 It's no sin to be a beginner or an intermediate. It's no sin to be a competent
263 programmer instead of a leader. The sin is in how long you remain a beginner or
264 intermediate after you know what you have to do to improve.

265 33.4 Intellectual Honesty

266 Part of maturing as a programming professional is developing an
267 uncompromising sense of intellectual honesty. Intellectual honesty commonly
268 manifests itself in several ways:

- 269 • Refusing to pretend you're an expert when you're not
- 270 • Readily admitting your mistakes
- 271 • Trying to understand a compiler warning rather than suppressing the
272 message
- 273 • Clearly understanding your program—not compiling it to see if it works
- 274 • Providing realistic status reports
- 275 • Providing realistic schedule estimates and holding your ground when
276 management asks you to adjust them

277 The first two items on this list—admitting that you don't know something or that
278 you made a mistake—echo the theme of intellectual humility discussed earlier.
279 How can you learn anything new if you pretend that you know everything
280 already? You'd be better off pretending that you don't know anything. Listen to
281 people's explanations, learn something new from them, and assess whether *they*
282 know what *they* are talking about.

283 Be ready to quantify your degree of certainty on any issue. If it's usually 100
284 percent, that's a warning sign.

285 *Any fool can defend his
286 or her mistakes—and
287 most fools do.*
288 —Dale Carnegie

289
290

291 Refusing to admit mistakes is a particularly annoying habit. If Sally refuses to
292 admit a mistake, she apparently believes that not admitting the mistake will trick
293 others into believing that she didn't make it. The opposite is true. Everyone will
294 know she made a mistake. Mistakes are accepted as part of the ebb and flow of
complex intellectual activities, and as long as she hasn't been negligent, no one
will hold mistakes against her.

If she refuses to admit a mistake, the only person she'll fool is herself. Everyone
else will learn that they're working with a prideful programmer who's not
completely honest. That's a more damning fault than making a simple error. If
you make a mistake, admit it quickly and emphatically.

295 Pretending to understand compiler messages when you don't is another common
296 blind spot. If you don't understand a compiler warning or if you think you know
297 what it means but are too pressed for time to check it, guess what's really a
298 waste of time? You'll probably end up trying to solve the problem from the
299 ground up while the compiler waves the solution in your face. I've had several
300 people ask for help in debugging programs. I'll ask if they have a clean compile,
301 and they'll say yes. Then they'll start to explain the symptoms of the problem,
302 and I'll say, "Hmmmm. That sounds like it would be an uninitialized pointer, but
303 the compiler should have warned you about that." Then they'll say, "Oh yeah—it
304 did warn about that. We thought it meant something else." It's hard to fool other
305 people about your mistakes. It's even harder to fool the computer, so don't waste
306 your time trying.

307 A related kind of intellectual sloppiness occurs when you don't quite understand
308 your program and "just compile it to see if it works." In that situation, it doesn't
309 really matter whether the program works because you don't understand it well
310 enough to know whether it works or not. Remember that testing can only show
the presence of errors, not their absence. If you don't understand the program,
311 you can't test it thoroughly. Feeling tempted to compile a program to "see what
312 happens" is a warning sign. It might mean that you need to back up to design or
313 that you began coding before you were sure you knew what you were doing.
314 Make sure you have a strong intellectual grip on the program before you
315 relinquish it to the compiler.

317 ***The first 90 percent of the***
318 ***code accounts for the first***
319 ***90 percent of the***
320 ***development time. The***
321 ***remaining 10 percent of***
322 ***the code accounts for the***
323 ***other 90 percent of the***
324 ***development time.***

325 —Tom Cargill

326

327

328

329

330

331

332

333

334

335

336

337

338

339

340

341

342

343

344

345

346

347

348

349

350

351

352

353

354

355

356

357

Status reporting is an area of scandalous duplicity. Programmers are notorious for saying that a program is “90 percent complete” during the last 50 percent of the project. If your problem is that you have a poor sense of your own progress, you can solve it by learning more about how you work. But if your problem is that you don’t speak your mind because you want to give the answer your manager wants to hear, that’s a different story. A manager usually appreciates honest observations about the status of a project, even if they’re not the opinions the manager wants to hear. If your observations are well thought out, give them as dispassionately as you can and in private. Management needs to have accurate information to coordinate development activities, and full cooperation is essential.

An issue related to inaccurate status reporting is inaccurate estimation. The typical scenario goes like this: Management asks Bert for an estimate of how long it would take to develop a new database product. Bert talks to a few programmers, crunches some numbers, and comes back with an estimate of eight programmers and six months. His manager says, “That’s not really what we’re looking for. Can you do it in a shorter time, with fewer programmers?” Bert goes away and thinks about it and decides that for a short period he could cut training and vacation time and have everyone work a little overtime. He comes back with an estimate of six programmers and four months. His manager says, “That’s great. This is a relatively low-priority project, so try to keep it on time without any overtime because the budget won’t allow it.”

The mistake Bert made was not realizing that estimates aren’t negotiable. He can revise an estimate to be more accurate, but negotiating with his boss won’t change the time it takes to develop a software project. IBM’s Bill Weimer says, “We found that technical people, in general, were actually very good at estimating project requirements and schedules. The problem they had was defending their decisions; they needed to learn how to hold their ground” (Weimer in Metzger and Boddie 1996). Bert’s not going to make his manager any happier by promising to deliver a project in four months and delivering it in six than he would by promising and delivering it in six. In the long run, he’ll lose credibility by compromising. In the short run, he’ll gain respect by standing firm on his estimate.

If management applies pressure to change your estimate, realize that ultimately the decision whether to do a project rests with management. Say “Look. This is how much it’s going to cost. I can’t say whether it’s worth this price to the company—that’s your job. But I can tell you how long it takes to develop a piece of software—that’s my job. I can’t ‘negotiate’ how long it will take; that’s like negotiating how many feet are in a mile. You can’t negotiate laws of nature. We can, however, negotiate other aspects of the project that affect the schedule and then reestimate the schedule. We can eliminate features, reduce performance,

358 develop the project in increments, or use fewer people and a longer schedule or
359 more people and a shorter schedule."

360 One of the scariest exchanges I've ever heard was at a lecture on managing
361 software projects. The speaker was the author of a best-selling software-project-
362 management book. A member of the audience asked, "What do you do if
363 management asks for an estimate and you know that if you give them an accurate
364 estimate they'll say it's too high and decide not to do the project?" The speaker
365 responded that that was one of those tricky areas in which you had to get
366 management to buy into the project by underestimating it. He said that once
367 they'd invested in the first part of the project, they'd see it through to the end.

368 Wrong answer! Management is responsible for the big-picture issues of running
369 a company. If a certain software capability is worth \$250K to a company and
370 you estimate it will cost \$750K to develop, the company shouldn't develop the
371 software. It's management's responsibility to make such judgments. When the
372 speaker advocated lying about the project's cost, telling management it would
373 cost less than it really would, he advocated covertly stealing management's
374 authority. If you think a project is interesting, breaks important new ground for
375 the company, or provides valuable training, say so. Management can weigh those
376 factors too. But tricking management into making the wrong decision could
377 literally cost the company hundreds of thousands of dollars. If it costs you your
378 job, you'll have gotten what you deserve.

379 33.5 Communication and Cooperation

380 Truly excellent programmers learn how to work and play well with others.
381 Writing readable code is part of being a team player.

382 The computer probably reads your program as often as other people do, but it's a
383 lot better at reading poor code than people are. As a readability guideline, keep
384 the person who has to modify your code in mind. Programming is
385 communicating with another programmer first, communicating with the
386 computer second.

387 Most good programmers enjoy making their programs readable, given sufficient
388 time to do so. There are a few holdouts, though, and some of them are good
389 coders.

390

391 *When I got out of school,*
392 *I thought I was the best*
393 *programmer in the world.*
394 *I could write an*
395 *unbeatable tic-tac-toe*
396 *program, use five*
397 *different computer*
languages, and create
398 *1000-line programs that*
399 *WORKED. (really!)*
400 *Then I got out into the*
401 *Real World. My first task*
402 *in the Real World was to*
403 *read and understand a*
200,000-line Fortran
404 *program, then speed it up*
405 *by a factor of two. Any*
406 *Real Programmer will tell*
407 *you that all the*
408 *Structured Coding in the*
409 *world won't help you*
410 *solve a problem like*
411 *that—it takes actual*
412 *talent.*

413 —Ed Post, from “Real
414 Programmers Don’t Use
415 Pascal”

416

417

418

419

420

421

422

423

33.6 Creativity and Discipline

It’s hard to explain to a fresh computer-science graduate why you need conventions and engineering discipline. When I was an undergraduate, the largest program I wrote was about 500 lines of executable code. As a professional, I’ve written dozens of utilities that have been smaller than 500 lines, but the average main-project size has been 5,000 to 25,000 lines, and I’ve participated in projects with over a half million lines of code. This type of effort requires not the same skills on a larger scale, but a new set of skills altogether.

Some creative programmers view the discipline of standards and conventions as stifling to their creativity. The opposite is true. Without standards and conventions on large projects, project completion itself is impossible. Creativity isn’t even imaginable. Don’t waste your creativity on things that don’t matter. Establish conventions in noncritical areas so that you can focus your creative energies in the places that count.

In a 15-year retrospective on work at NASA’s Software Engineering Laboratory, McGarry and Pajerski reported that methods and tools that emphasize human discipline have been especially effective (1990). Many highly creative people have been extremely disciplined. “Form is liberating,” as the saying goes. Great architects work within the constraints of physical materials, time, and cost. Great artists do too. Anyone who has examined Leonardo’s drawings has to admire his disciplined attention to detail. When Michelangelo designed the ceiling of the Sistine Chapel, he divided it into symmetric collections of geometric forms such as triangles, circles, and squares. He designed it in three zones corresponding to three Platonic stages. Without this self-imposed structure and discipline, the 300 human figures would have been merely chaotic rather than the coherent elements of an artistic masterpiece.

A programming masterpiece requires just as much discipline. If you don’t try to analyze requirements and design before you begin coding, much of your learning about the project will occur during coding, and the result of your labors will look more like a three-year-old’s finger painting than a work of art.

33.7 Laziness

Laziness manifests itself in several ways:

- Deferring an unpleasant task
- Doing an unpleasant task quickly to get it out of the way

- 424 • Writing a tool to do the unpleasant task so that you never have to do the task
425 again

426 Some of these manifestations of laziness are better than others. The first is hardly
427 ever beneficial. You've probably had the experience of spending several hours
428 futzing with jobs that didn't really need to be done so that you wouldn't have to
429 face a relatively minor job that you couldn't avoid. I detest data entry, and many
430 programs require a small amount of data entry. I've been known to delay
431 working on a program for days just to delay the inevitable task of entering
432 several pages of numbers by hand. This habit is "true laziness." It manifests
433 itself again in the habit of compiling a class to see if it works so that you can
434 avoid the exercise of checking the class with your mind.

435 The small tasks are never as bad as they seem. If you develop the habit of doing
436 them right away, you can avoid the procrastinating kind of laziness. This habit is
437 "enlightened laziness"—the second kind of laziness. You're still lazy, but you're
438 getting around the problem by spending the smallest possible amount of time on
439 something that's unpleasant.

440 The third option is to write a tool to do the unpleasant task. This is "long-term
441 laziness." It is undoubtedly the most productive kind of laziness (provided that
442 you ultimately save time by having written the tool). In these contexts, a certain
443 amount of laziness is beneficial.

444 When you step through the looking glass, you see the other side of the laziness
445 picture. "Hustle" or "making an effort" doesn't have the rosy glow it does in
446 high-school phys-ed class. Hustle is extra, unnecessary effort. It shows that
447 you're eager but not that you're getting your work done. It's easy to confuse
448 motion with progress; busy-ness with being productive. The most important
449 work in effective programming is thinking, and people tend not to look busy
450 when they're thinking. If I worked with a programmer who looked busy all the
451 time, I'd assume that he was not a good programmer because he wasn't using his
452 most valuable tool, his brain.

453 **33.8 Characteristics That Don't Matter As
454 Much As You Might Think**

455 Hustle isn't the only characteristic that you might admire in other aspects of your
456 life but that doesn't work very well in software development.

457 Persistence

458 Depending on the situation, persistence can be either an asset or a liability. Like
459 most value-laden concepts, it's identified by different words depending on
460 whether you think it's a good quality or a bad one. If you want to identify
461 persistence as a bad quality, you say it's "stubbornness" or "pigheadedness." If
462 you want it to be a good quality, you call it "tenacity" or "perseverance."

463 Most of the time, persistence in software development is pigheadedness—it has
464 little value. Persistence when you're stuck on a piece of new code is hardly ever
465 a virtue. Try redesigning the class, try an alternative coding approach, or try
466 coming back to it later. When one approach isn't working, that's a good time to
467 try an alternative (Pirsig 1974).

468 **CROSS-REFERENCE** For
469 a more detailed discussion of
470 persistence in debugging, see
471 "Tips for Finding Defects" in
472 Section 23.2.
473
474

In debugging, it can be mighty satisfying to track down the error that has been annoying you for four hours, but it's often better to give up on the error after a certain amount of time with no progress—say 15 minutes. Let your subconscious chew on the problem for a while. Try to think of an alternative approach that would circumvent the problem altogether. Rewrite the troublesome section of code from scratch. Come back to it later when your mind is fresh. Fighting computer problems is no virtue. Avoiding them is better.

475 It's hard to know when to give up, but it's essential that you ask. When you
476 notice that you're frustrated, that's a good time to ask the question. Asking
477 doesn't necessarily mean that it's time to give up, but it probably means that it's
478 time to set some parameters on the activity: "If I don't solve the problem using
479 this approach within the next 30 minutes, I'll take 10 minutes to brainstorm
480 about different approaches and try the best one for the next hour."

481 Experience

482 The value of hands-on experience as compared to book learning is smaller in
483 software development than in many other fields for several reasons. In many
484 other fields, basic knowledge changes slowly enough that someone who
485 graduated from college 10 years after you did probably learned the same basic
486 material that you did. In software development, even basic knowledge changes
487 rapidly. The person who graduated from college 10 years after you did probably
488 learned twice as much about effective programming techniques. Older
489 programmers tend to be viewed with suspicion not just because they might be
490 out of touch with specific technology but because they might never have been
491 exposed to basic programming concepts that became well known after they left
492 school.

493 In other fields, what you learn about your job today is likely to help you in your
494 job tomorrow. In software, if you can't shake the habits of thinking you
495 developed while using your former programming language or the code-tuning
496 techniques that worked on your old machine, your experience will be worse than
497 none at all. A lot of software people spend their time preparing to fight the last
498 war rather than the next one. If you can't change with the times, experience is
499 more a handicap than a help.

500 Aside from the rapid changes in software development, people often draw the
501 wrong conclusions from their experiences. It's hard to view your own life
502 objectively. You can overlook key elements of your experience that would cause
503 you to draw different conclusions if you recognized them. Reading studies of
504 other programmers is helpful because the studies reveal other people's
505 experience—filtered enough that you can examine it objectively.

506 People also put an absurd emphasis on the *amount* of experience programmers
507 have. "We want a programmer with five years of C programming experience" is
508 a silly statement. If a programmer hasn't learned C after a year or two, the next
509 three years won't make much difference. This kind of "experience" has little
510 relationship to performance.

511 The fact that information changes quickly in programming makes for weird
512 dynamics in the area of "experience." In many fields, a professional who has a
513 history of achievement can coast—relaxing and enjoying the respect earned by a
514 string of successes. In software development, anyone who coasts quickly
515 becomes out of touch. To stay valuable, you have to stay current. For young,
516 hungry programmers, this is an advantage. Older programmers sometimes feel
517 they've already earned their stripes and resent having to prove themselves year
518 after year.

519 The bottom line on experience is this: If you work for 10 years, do you get 10
520 years of experience or do you get 1 year of experience 10 times? You have to
521 reflect on your activities to get true experience. If you make learning a
522 continuous commitment, you'll get experience. If you don't, you won't, no
523 matter how many years you have under your belt.

524 Gonzo Programming

525 *If you haven't spent at least a month working on the*
526 *same program—working 16 hours a day, dreaming about it*
527 *during the remaining 8 hours of restless sleep, working several*
528 *nights straight through trying to eliminate that "one last bug"*
529 *from the program—then you haven't really written a*

530 *complicated computer program. And you may not have the*
531 *sense that there is something exhilarating about programming.*

532 *Edward Yourdon*

533 This lusty tribute to programming machismo is pure B.S. and an almost certain
534 recipe for failure. Those all-night programming stints make you feel like the
535 greatest programmer in the world, but then you have to spend several weeks
536 correcting the defects you installed during your blaze of glory. By all means, get
537 excited about programming. But excitement is no substitute for competency.
538 Remember which is more important.

539 **33.9 Habits**

540 *The moral virtues, then, are engendered in us neither by*
541 *nor contrary to nature...their full development in us is due to*
542 *habit....Anything that we have to learn to do we learn by the*
543 *actual doing of it....Men will become good builders as a result*
544 *of building well and bad ones as a result of building*
545 *badly....So it is a matter of no little importance what sort of*
546 *habits we form from the earliest age—it makes a vast*
547 *difference, or rather all the difference in the world.*

548 *Aristotle*

549 Good habits matter because most of what you do as a programmer you do
550 without consciously thinking about it. For example, at one time, you might have
551 thought about how you wanted to format indented loops, but now you don't
552 think about it again each time you write a new loop. You do it the way you do it
553 out of habit. This is true of virtually all aspects of program formatting. When
554 was the last time you seriously questioned your formatting style? Chances are
555 good that if you've been programming for five years, you last questioned it four
556 and a half years ago. The rest has been habit.

557 **CROSS-REFERENCE** For
558 details on errors in
559 assignment statements, see
560 “Errors by Classification” in
561 Section 22.4.
562
563
564 You have habits in many areas. For example, programmers tend to check loop
indexes carefully and not to check assignment statements, making errors in
assignment statements much harder to find than errors in loop indexes (Gould
1975). You respond to criticism in a friendly way or in an unfriendly way.
You're always looking for ways to make code readable or fast, or you're not. If
you have to choose between making code fast and making it readable, and you
make the same choice every time, you're not really choosing; you're responding
out of habit.

565 Study the quotation from Aristotle and substitute “programming virtues” for
566 “moral virtues.” He points out that you are not predisposed to either good or bad
567 behavior but are constituted in such a way that you can become either a good or
568 a bad programmer. The main way you become good or bad at what you do is by
569 doing—builders by building and programmers by programming. What you do
570 becomes habit, and what you do by habit determines whether you have the
571 “programming virtues.” Over time, your good and bad habits determine whether
572 you’re a good or a bad programmer.

573 Bill Gates says that any programmer who will ever be good is good in the first
574 few years. After that, whether a programmer is good or not is cast in concrete
575 (Lammers 1986). After you’ve been programming a long time, it’s hard to
576 suddenly start saying, “How do I make this loop faster?” or “How do I make this
577 code more readable?” These are habits that good programmers develop early.

578 When you first learn something, learn it the right way. When you first do it,
579 you’re actively thinking about it and you still have an easy choice between doing
580 it in a good way and doing it in a bad way. After you’ve done it a few times, you
581 pay less attention to what you’re doing and “force of habit” takes over. Make
582 sure that the habits that take over are the ones you want to have.

583 What if you don’t already have the most effective habits? How do you change a
584 bad habit? If I had the definitive answer to that, I could sell self-help tapes on
585 late-night TV. But here’s at least part of an answer. You can’t replace a bad habit
586 with no habit at all. That’s why people who suddenly stop smoking or swearing
587 or overeating have such a hard time unless they substitute something else, like
588 chewing gum. It’s easier to replace an old habit with a new one than it is to
589 eliminate one altogether. In programming, try to develop new habits that work.
590 Develop the habit of writing a class in pseudocode before coding it and carefully
591 reading the code before compiling it, for instance. You won’t have to worry
592 about losing the bad habits; they’ll naturally drop by the wayside as new habits
593 take their places.

CC2E.COM/3327

594 Additional Resources

595 CC2E.COM/3334 Dijkstra, Edsger. “The Humble Programmer.” Turing Award Lecture.
596 *Communications of the ACM* 15, no. 10 (October 1972): 859–66. This classic
597 paper helped begin the inquiry into how much computer programming depends
598 on the programmer’s mental abilities. Dijkstra has persistently stressed the
599 message that the essential task of programming is mastering the enormous
600 complexity of computer science. He argues that programming is the only activity
601 in which humans have to master nine orders of magnitude of difference between
602 the lowest level of detail and the highest. This paper would be interesting reading

603 solely for its historical value, but many of its themes sound fresh 20 years later.
604 It also conveys a good sense of what it was like to be a programmer in the early
605 days of computer science.

606 Weinberg, Gerald M. *The Psychology of Computer Programming: Silver
607 Anniversary Edition*. New York: Dorset House, 1998. This classic book contains
608 a detailed exposition of the idea of egoless programming and of many other
609 aspects of the human side of computer programming. It contains many
610 entertaining anecdotes and is one of the most readable books yet written about
611 software development.

612 Pirsig, Robert M.. *Zen and the Art of Motorcycle Maintenance : An Inquiry into
613 Values*, William Morrow, 1974. Pirsig provides an extended discussion of
614 “quality,” ostensibly as it relates to motorcycle maintenance. Pirsig was working
615 as a software technical writer when he wrote *ZAMM*, and his insightful
616 comments apply as much to software projects as motorcycle maintenance.

617 Curtis, Bill, ed. *Tutorial: Human Factors in Software Development*. Los
618 Angeles: IEEE Computer Society Press, 1985. This is an excellent collection of
619 papers that address the human aspects of creating computer programs. The 45
620 papers are divided into sections on mental models of programming knowledge,
621 learning to program, problem solving and design, effects of design
622 representations, language characteristics, error diagnosis, and methodology. If
623 programming is one of the most difficult intellectual challenges that humankind
624 has ever faced, learning more about human mental capacities is critical to the
625 success of the endeavor. These papers about psychological factors also help you
626 to turn your mind inward and learn about how you individually can program
627 more effectively.

628 McConnell, Steve. *Professional Software Development*, Boston, MA: Addison
629 Wesley, 2004. Chapter 7, “Orphans Preferred,” provides more details on
630 programmer personalities and the role of personal character.

631 Key Points

- 632 • Your personal character directly affects your ability to write computer
633 programs.
- 634 • The characteristics that matter most are humility, curiosity, intellectual
635 honesty, creativity and discipline, and enlightened laziness.
- 636 • The characteristics of a superior programmer have almost nothing to do with
637 talent and everything to do with a commitment to personal development.

- 638 • Surprisingly, raw intelligence, experience, persistence, and guts hurt as
639 much as they help.
- 640 • Many programmers don't actively seek new information and techniques and
641 instead rely on accidental, on-the-job exposure to new information. If you
642 devote a small percentage of your time to reading and learning about
643 programming, after a few months or years you'll dramatically distinguish
644 yourself from the programming mainstream.
- 645 • Good character is mainly a matter of having the right habits. To be a great
646 programmer, develop the right habits, and the rest will come naturally.

34

Themes in Software Craftsmanship

Contents

- 34.1 Conquer Complexity
- 34.2 Pick Your Process
- 34.3 Write Programs for People First, Computers Second
- 34.4 Program Into Your Language, Not In It
- 34.5 Focus Your Attention with the Help of Conventions
- 34.6 Program in Terms of the Problem Domain
- 34.7 Watch for Falling Rocks
- 34.8 Iterate, Repeatedly, Again and Again
- 34.9 Thou Shalt Rend Software and Religion Asunder

Related Topics

The whole book

THIS BOOK IS MOSTLY ABOUT the details of software construction: high-quality classes, variable names, loops, source-code layout, system integration, and so on. This book has de-emphasized abstract topics in order to emphasize subjects that are more concrete.

Once the earlier parts of the book have put the concrete topics on the table, all you have to do to appreciate the abstract concepts is to pick up the topics from the various chapters and see how they're related. This chapter makes the abstract themes explicit: complexity, abstraction, process, readability, iteration, and so on. These themes account in large part for the difference between hacking and software craftsmanship.

26

27 **CROSS-REFERENCE** For
28 details on the importance of
29 attitude in conquering
30 complexity, see Section 33.2,
31 “Intelligence and Humility.”
32
33
34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

34.1 Conquer Complexity

The drive to reduce complexity is at the heart of software development—to such a degree that Chapter 5 described managing complexity as The Major Technical Imperative in Software. Although it’s tempting to try to be a hero and deal with computer-science problems at all levels, no one’s brain is really capable of spanning nine orders of magnitude of detail. Computer science and software engineering have developed many intellectual tools for handling such complexity, and discussions of other topics in this book have brushed up against several of them.

- Dividing a system into subsystems at the architecture level so that your brain can focus on a smaller amount of the system at one time.
- Carefully defining class interfaces so that you can ignore the internal workings of the class
- Preserving the abstraction represented by the class interface so that your brain doesn’t have to remember arbitrary details.
- Avoiding global data, because global data vastly increases the percentage of the code you need to juggle in your brain at any one time.
- Avoiding deep inheritance hierarchies because they are intellectually demanding
- Avoiding deep nesting of loops and conditionals because they can be replaced by simpler control structures that burn up fewer gray cells.
- Avoiding gotos because they introduce non-linearity that has been found to be difficult for most people to follow.
- Carefully defining your approach to error handling rather than using an arbitrary proliferation of different error-handling techniques.
- Being systematic about the use of the built-in exception mechanism, which can become a non-linear control structure that is about as hard to understand as gotos if not used with discipline.
- Not allowing classes to grow into monster classes that amount to whole programs in themselves.
- Keeping routines short.
- Using clear, self-explanatory variable names so that your brain doesn’t have to waste cycles remembering details like “*i* stands for the account index, and *j* stands for the customer index, or was it the other way around?”

- 60 • Minimizing the number of parameters passed to a routine, or, more
61 important, passing only the parameters needed to preserve the routine
62 interface's abstraction.
- 63 • Using conventions to spare your brain the challenge of remembering
64 arbitrary, accidental differences between different sections of code.
- 65 • In general, attacking what Chapter 5 describes as "accidental details"
66 wherever possible.

67 When you put a complicated test into a boolean function and abstract the
68 purpose of the test, you make the code less complex. When you substitute a table
69 lookup for a complicated chain of logic, you do the same thing. When you create
70 a well-defined, consistent class interface, you eliminate the need to worry about
71 implementation details of the class and simplify your job overall.

72 The point of having coding conventions is also mainly to reduce complexity.
73 When you can standardize decisions about formatting, loops, variable names,
74 modeling notations, and so on, you release mental resources that you need to
75 focus on more challenging aspects of the programming problem. One reason
76 coding conventions are so controversial is that choices among the options have
77 some limited aesthetic base but are essentially arbitrary. People have the most
78 heated arguments over their smallest differences. Conventions are the most
79 useful when they spare you the trouble of making and defending arbitrary
80 decisions. They're less valuable when they impose restrictions in more
81 meaningful areas.

82 Abstraction in its various forms is a particularly powerful tool for managing
83 complexity. Programming has advanced largely through increasing the
84 abstractness of program components. Fred Brooks argues that the biggest single
85 gain ever made in computer science was in the jump from machine language to
86 higher-level languages—it freed programmers from worrying about the detailed
87 quirks of individual pieces of hardware and allowed them to focus on
88 programming (Brooks 1995). The idea of routines was another big step, followed
89 by classes and packages.

90 Naming variables functionally, for the "what" of the problem rather than the
91 "how" of the implementation-level solution, increases the level of abstraction. If
92 you say, "OK, I'm popping the stack and that means that I'm getting the most
93 recent employee," abstraction can save you the mental step "I'm popping the
94 stack." You simply say, "I'm getting the most recent employee." This is a small
95 gain, but when you're trying to reduce a range in complexity of 1 to 10^9 , every
96 step counts. Using named constants rather than literals also increases the level of
97 abstraction. Object-oriented programming provides a level of abstraction that

98 applies to algorithms and data at the same time, a kind of abstraction that
99 functional decomposition alone didn't provide.

100 In summary, a primary goal of software design and construction is conquering
101 complexity. The motivation behind many programming practices is to reduce a
102 program's complexity. Reducing complexity is arguably the most important key
103 to being an effective programmer.

104 34.2 Pick Your Process

105 A second major thread in this book is the idea that the process you use to
106 develop software matters a surprising amount.

107 On a small project, the talents of the individual programmer are the biggest
108 influence on the quality of the software. Part of what makes an individual
109 programmer successful is his or her choice of processes.

110 On projects with more than one programmer, organizational characteristics make
111 a bigger difference than the skills of the individuals involved do. Even if you
112 have a great team, its collective ability isn't simply the sum of the team
113 members' individual abilities. The way in which people work together
114 determines whether their abilities are added to each other or subtracted from
115 each other. The process the team uses determines whether one person's work
116 supports the work of the rest of the team or undercuts it.

117 **CROSS-REFERENCE** For
118 details on making
119 requirements stable, see
120 Section 3.4, "Requirements
121 Prerequisite." For details on
122 variations in development
123 approaches, see Section 3.2,
124 "Determine the Kind of
125 Software You're Working
126 On."

127 One example of the way in which process matters is the consequence of not
128 making requirements stable before you begin designing and coding. If you don't
129 know what you're building, you can't very well create a superior design for it. If the
130 requirements and subsequently the design change while the software is under
131 development, the code must change too, which risks degrading the quality of the
132 system.

133 "Sure," you say, "but in the real world, you never really have stable
134 requirements, so that's a red herring." Again, the process you use determines
135 both how stable your requirements are and how stable they need to be. If you
136 want to build more flexibility into the requirements, you can set up an
137 incremental development approach in which you plan to deliver the software in
138 several increments rather than all at once. This is an attention to process, and it's
139 the process you use that ultimately determines whether your project succeeds or
140 fails. Table 3-1 in Section 3.1 makes it clear that requirements errors are far
141 more costly than construction errors, so focusing on that part of the process also
142 affects cost and schedule.

133 *My message to the serious
134 programmer is: spend a
135 part of your working day
136 examining and refining
137 your own methods. Even
138 though programmers are
139 always struggling to meet
140 some future or past
141 deadline, methodological
142 abstraction is a wise long
143 term investment.*

144 — Robert W. Floyd

145
146
147
148
149
150
151
152
153
154

155 **CROSS-REFERENCE** For
156 details on iteration, see
157 Section 34.8, “Iterate,
158 Repeatedly, Again and
Again,” later in this chapter.

159
160
161
162
163
164
165
166

The same principle of consciously attending to process applies to design. You have to lay a solid foundation before you can begin building on it. If you rush to coding before the foundation is complete, it will be harder to make fundamental changes in the system’s architecture. People will have an emotional investment in the design because they will have already written code for it. It’s hard to throw away a bad foundation once you’ve started building a house on it.

The main reason the process matters is that in software, quality must be built in from the first step onward. This flies in the face of the folk wisdom that you can code like hell and then test all the mistakes out of the software. That idea is dead wrong. Testing merely tells you the specific ways in which your software is defective. Testing won’t make your program more usable, faster, smaller, more readable, or more extensible.

Premature optimization is another kind of process error. In an effective process, you make coarse adjustments at the beginning and fine adjustments at the end. If you were a sculptor, you’d rough out the general shape before you started polishing individual features. Premature optimization wastes time because you spend time polishing sections of code that don’t need to be polished. You might polish sections that are small enough and fast enough as they are; you might polish code that you later throw away; you might fail to throw away bad code because you’ve already spent time polishing it. Always be thinking, “Am I doing this in the right order? Would changing the order make a difference?”

Consciously follow a good process.

Low-level processes matter too. If you follow the process of writing pseudocode and then filling in the code around the pseudocode, you reap the benefits of designing from the top down. You’re also guaranteed to have comments in the code without having to put them in later.

Observing large processes and small processes means pausing to pay attention to how you create software. It’s time well spent. Saying that “code is what matters; you have to focus on how good the code is, not some abstract process” is shortsighted and ignores mountains of experimental and practical evidence to the contrary. Software development is a creative exercise. If you don’t understand the creative process, you’re not getting the most out of the primary tool you use to create software—your brain. A bad process wastes your brain cycles. A good process leverages them to maximum advantage.

167

34.3 Write Programs for People First, Computers Second

169

170

your program *n.* A maze of non sequiturs littered with clever-clever tricks and irrelevant comments. *Compare MY PROGRAM.*

171

172

173

174

my program *n.* A gem of algorithmic precision, offering the most sublime balance between compact, efficient coding on the one hand and fully commented legibility for posterity on the other. *Compare YOUR PROGRAM.*

175

Stan Kelly-Bootle

176

177

178

Another theme that runs throughout this book is an emphasis on code readability. Communication with other people is the motivation behind the quest for the Holy Grail of self-documenting code.

179

180

104

181

The computer doesn't care whether your code is readable. It's better at reading binary machine instructions than it is at reading high-level-language statements. You write readable code because it helps other people to read your code. Readability has a positive effect on all these aspects of a program:

183

184

185

186

187

188

- Comprehensibility
 - Reviewability
 - Error rate
 - Debugging
 - Modifiability
 - Development time—a consequence of all of the above
 - External quality—a consequence of all of the above

190 *In the early years of
191 programming, a program
192 was regarded as the
193 private property of the
194 programmer. One would
195 no more think of reading
a colleague's program
196 unbidden than of picking
197 up a love letter and
198 reading it. This is
199 essentially what a
200 program was, a love letter
from the programmer to
the hardware, full of the
202 intimate details known
only to partners in an
affair. Consequently,
205 programs became larded
with the pet names and
206 verbal shorthand so
207 popular with lovers who
208 live in the blissful
209 abstraction that assumes
210 that theirs is the only
existence in the universe.
Such programs are
212 unintelligible to those
213 outside the partnership.*

214 — Michael Marcotty

215

216

217
218
219
220
221
222
223

224 **HARD DATA**

225
226
227

Readable code doesn't take any longer to write than confusing code does, at least not in the long run. It's easier to be sure your code works if you can easily read what you wrote. That should be a sufficient reason to write readable code. But code is also read during reviews. It's read when you or someone else fixes an error. It's read when the code is modified. It's read when someone tries to use part of your code in a similar program.

Making code readable is not an optional part of the development process, and favoring write-time convenience over read-time convenience is a false economy. You should go to the effort of writing good code, which you can do once, rather than the effort of reading bad code, which you'd have to do again and again.

“What if I’m just writing code for myself? Why should I make it readable?” Because a week or two from now you’re going to be working on another program and think, “Hey! I already wrote this class last week. I’ll just drop in my old tested, debugged code and save some time.” If the code isn’t readable, good luck!

The idea of writing unreadable code because you’re the only person working on a project sets a dangerous precedent. Your mother used to say, “What if your face froze in that expression?” Habits affect all your work; you can’t turn them on and off at will, so be sure that what you’re doing is something you want to become a habit. A professional programmer writes readable code, period.

It’s also good to recognize that whether a piece of code ever belongs exclusively to you is debatable. Douglas Comer came up with a useful distinction between private and public programs (Comer 1981): “Private programs” are programs for a programmer’s own use. They aren’t used by others. They aren’t modified by others. Others don’t even know the programs exist. They are usually trivial, and they are the rare exception. “Public programs” are programs used or modified by someone other than the author.

Standards for public and for private programs can be different. Private programs can be sloppily written and full of limitations without affecting anyone but the author. Public programs must be written more carefully: Their limitations should be documented; they should be reliable; and they should be modifiable. Beware of a private program’s becoming public, as private programs often do. You need to convert the program to a public program before it goes into general circulation. Part of making a private program public is making it readable.

Even if you think you’re the only one who will read your code, in the real world chances are good that someone else will need to modify your code. One study found that 10 generations of maintenance programmers work on an average program before it gets rewritten (Thomas 1984). Maintenance programmers

228 spend 50 to 60 percent of their time trying to understand the code they have to
229 maintain, and they appreciate the time you put into documenting it (Parikh and
230 Zvegintzov 1983).

231 Earlier chapters examined the techniques that help you achieve readability: good
232 class, routine, and variable names, careful formatting, small routines, hiding
233 complex boolean tests in boolean functions, assigning intermediate results to
234 variables for clarity in complicated calculations, and so on. No individual
235 application of a technique can make the difference between a readable program
236 and an illegible one. But the accumulation of many small readability
237 improvements will be significant.

238 If you think you don't need to make your code readable because no one else ever
239 looks at it, make sure you're not confusing cause and effect.

240 34.4 Program Into Your Language, Not In It

241 Don't limit your programming thinking only to the concepts that are supported
242 automatically by your language. The best programmers think of what they want
243 to do, and then they assess how to accomplish their objectives with the
244 programming tools at their disposal.

245 Should you use a class member routine that's inconsistent with the class's
246 abstraction just because it's more convenient than using one that provides more
247 consistency? You should write code in a way that preserves the abstraction
248 represented by the class's interface as much as possible. You don't need to use
249 global data or *gosub*s just because your language supports them. You can choose
250 not to use those hazardous programming capabilities—use programming
251 conventions to make up for weaknesses of the language. The fact that your
252 language has a *try-catch* structure doesn't automatically mean that exception
253 handling is the best error-handling approach. Programming using the most
254 obvious path amounts to programming *in* a language rather than programming
255 *into* a language; it's the programmer's equivalent of, "If Freddie jumped off a
256 bridge, would you jump off a bridge too?" Think about your technical goals,
257 then decide how best to accomplish those goals by programming *into* your
258 language.

259 Your language doesn't support assertions? Write your own *assert()* routine. It
260 might not function exactly the same as a built-in *assert()*, but you can still realize
261 most of *assert()*'s benefits by writing your own routine. Your language doesn't
262 support enumerated types or named constants? That's fine; you can define your
263 own enumerations and named constants with a disciplined use of global
264 variables supported by clear naming conventions.

265
266
267
268
269
270
271
272
273

In extreme cases, especially in new-technology environments, your tools might be so primitive that you're forced to change your desired programming approach significantly. In such cases, you might have to balance your desire to program into the language with the accidental difficulties that are created when the language makes your desired approach too cumbersome. But in such cases, you will benefit even more from programming conventions that help you steer clear of those environments' most hazardous features. In more typical cases, the gap between what you want to do and what your tools will readily support will require you to make only relatively minor concessions to your environment.

274

34.5 Focus Your Attention with the Help of Conventions

275

276 **CROSS-REFERENCE** For
277 an analysis of the value of
278 conventions as they apply to
279 program layout, see "How
280 Much Is Good Layout
281 Worth?" and "Objectives of
282 Good Layout" in Section
283 31.1.

284
285
286
287

288 A convention conveys important information concisely. In naming conventions,
289 a single character can differentiate among local, class, and global variables;
290 capitalization can concisely differentiate among types, named constants, and
291 variables. Indentation conventions can concisely show the logical structure of a
292 program. Alignment conventions can indicate concisely that statements are
293 related.

294 Conventions protect against known hazards. You can establish conventions to
295 eliminate the use of dangerous practices, to restrict such practices to cases in
296 which they're needed, or to compensate for their known hazards. You could
297 eliminate a dangerous practice, for example, by prohibiting global variables or
298 prohibiting multiple statements on a line. You could compensate for a hazardous
299 practice by requiring parentheses around complicated expressions or requiring
300 pointers to be set to *NULL* immediately after they're deleted to help prevent
301 dangling pointers.

302 Conventions add predictability to low-level tasks. Having conventional ways of
303 handling memory requests, error processing, input/output, and class interfaces
304 adds a meaningful structure to your code and makes it easier for another
305 programmer to figure out—as long as the programmer knows your conventions.
306 As mentioned in an earlier chapter, one of the biggest benefits of eliminating
307 global data is that you eliminate potential interactions among different classes
308 and subsystems. A reader knows roughly what to expect from local and class
309 data. But it's hard to tell when changing global data will break some bit of code
310 four subsystems away. Global data increases the reader's uncertainty. With good
311 conventions, you and your readers can take more for granted. The amount of
312 detail that has to be assimilated will be reduced, and that in turn will improve
313 program comprehension.

314 Conventions can compensate for language weaknesses. In languages that don't
315 support named constants (like Python, Perl, Unix shell script, and so on), a
316 convention can differentiate between variables intended to be both read and
317 written and those that are intended to emulate read-only constants. Conventions
318 for the disciplined use of global data and pointers are other examples of
319 compensating for language weaknesses with conventions.

320 Programmers on large projects sometimes go overboard with conventions. They
321 establish so many standards and guidelines that remembering them becomes a
322 full-time job. But programmers on small projects tend to go “underboard,” not
323 realizing the full benefits of intelligently conceived conventions. Understand
324 their real value and take advantage of them. Use them to provide structure in
325 areas in which structure is needed.

326 **34.6 Program in Terms of the Problem 327 Domain**

328 Another specific method of dealing with complexity is to work at the highest
329 possible level of abstraction. One way of working at a high level of abstraction is
330 to work in terms of the programming problem rather than the computer-science
331 solution.

332 Top-level code shouldn't be filled with details about files and stacks and queues
333 and arrays and characters whose parents couldn't think of better names for them
334 than *i*, *j*, and *k*. Top-level code should describe the problem that's being solved.
335 It should be packed with descriptive class names and routine calls that indicate
336 exactly what the program is doing, not cluttered with details about opening a file
337 as “read only.” Top-level code shouldn't contain clumps of comments that say “*i*

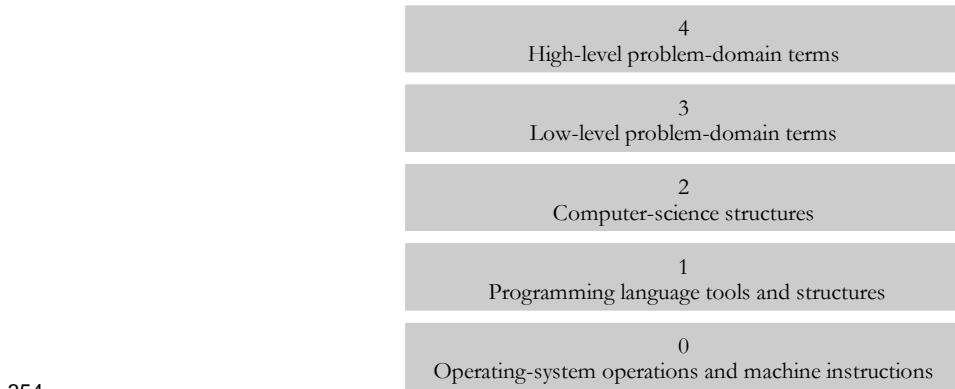
338 is a variable that represents the index of the record from the employee file here,
339 and then a little later it's used to index the client account file there..."

340 That's clumsy programming practice. At the top level of the program, you don't
341 need to know that the employee data comes as records or that it's stored as a file.
342 Information at that level of detail should be hidden. At the highest level, you
343 shouldn't have any idea how the data is stored. Nor do you need to read a
344 comment that explains what *i* means and that it's used for two purposes. You
345 should see a variable named something like *employeeIndex* so that you don't
346 need a verbose comment about *i*. If *i* has been used for two purposes, you should
347 see different variable names for the two purposes instead, and they should also
348 have distinctive names such as *employeeIndex* and *clientIndex*.

349 **Separating a Program into Levels of Abstraction**

350 Obviously, you have to work in implementation-level terms at some level, but
351 you can isolate the part of the program that works in implementation-level terms
352 from the part that works in problem-domain terms.

353 If you're designing a program, consider these levels of abstraction:



354

355

356

357

358

359

360

361

362

363

364

365

F34xx01

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.

Level 0: Operating System Operations and Machine Instructions

If you're working in a high-level language, you don't have to worry about the lowest level—your language takes care of it automatically. If you're working in a low-level language, you should try to create higher layers for yourself to work in, even though many programmers don't do that.

366
367
368
369
370
371
372

Level 1: Programming-Language Structures and Tools

Programming-language structures are the language's primitive data types, control structures, and so on. Most common languages also provide additional libraries, access to operating system calls, and so on. Using these structures and tools comes naturally since you can't program without them. Many programmers never work above this level of abstraction, which makes their lives much harder than they need to be.

373
374
375
376
377
378
379

Level 2: Low-Level Implementation Structures

Low-level implementation structures are slightly higher-level structures than those provided by the language itself. They tend to be the operations and data types you learn about in college courses in algorithms and data types—stacks, queues, linked lists, trees, indexed files, sequential files, sort algorithms, search algorithms, and so on. If your program consists entirely of code written at this level, you'll be awash in too much detail to win the battle against complexity.

380
381
382
383
384
385
386
387
388
389

Level 3: Low-Level Problem-Domain Terms

At this level, you have the primitives you need in order to work in terms of the problem domain. It's a glue layer between the computer-science structures below and the high-level problem-domain code above. To write code at this level, you need to figure out the vocabulary of the problem area and create building blocks you can use to work with the problem the program solves. In many applications, this will be the business objects layer or a services layer. Classes at this level provide the vocabulary and the building blocks. The classes might be too primitive to be used to solve the problem directly at this level, but they provide an Erector set that higher-level classes can use to build a solution to the problem.

390
391
392
393
394
395
396
397
398

Level 4: High-Level Problem-Domain Terms

This level provides the abstractive power to work with a problem on its own terms. Your code at this level should be somewhat readable by someone who's not a computer-science whiz—perhaps even by your non-technical customer. Code at this level won't depend much on the specific features of your programming language because you'll have built your own set of tools to work with the problem. Consequently, at this level your code depends more on the tools you've built for yourself at Level 3 than on the capabilities of the language you're using.

399
400
401
402
403

Implementation details should be hidden two layers below this one, in a layer of computer-science structures, so that changes in hardware or the operating system don't affect this layer at all. Embody the user's view of the world in the program at this level because when the program changes, it will change in terms of the user's view. Changes in the problem domain should affect this layer a lot, but

404 they should be easy to accommodate by programming in the problem-domain
405 building blocks from the layer below.

406 In addition to these conceptual layers, many programmers find it useful to break
407 a program up into other “layers” that cut across the layers described here. For
408 example, the typical 3-tier architecture cuts across the levels described here, and
409 provides further tools for making the design and code intellectually manageable.

410 **Low-Level Techniques for Working in the Problem 411 Domain**

412 Even without a complete, architectural approach to working in the problem
413 area’s vocabulary, you can use many of the techniques in this book to work in
414 terms of the real-world problem rather than the computer-science solution:

- 415 • Use classes to implement structures that are meaningful in problem-domain
416 terms.
- 417 • Hide information about the low-level data types and their implementation
418 details.
- 419 • Use named constants to document the meanings of strings and of numeric
420 literals.
- 421 • Assign intermediate variables to document the results of intermediate
422 calculations.
- 423 • Use boolean functions to clarify complex boolean tests.

424 **34.7 Watch for Falling Rocks**

425 Programming is neither fully an art nor fully a science. As it’s typically
426 practiced, it’s a “craft” that’s somewhere between art and science. At its best, it’s
427 an engineering discipline that arises from the synergistic fusion of art and
428 science (McConnell 2004). Whether art, science, craft, or engineering, it still
429 takes plenty of individual judgment to create a working software product. And
430 part of having good judgment in computer programming is being sensitive to a
431 wide array of warning signs, subtle indications of problems in your program.
432 Warning signs in programming alert you to the possibility of problems, but
433 they’re usually not as blatant as a road sign that says “Watch for falling rocks.”

434 When you or someone else says “This is really tricky code,” that’s a warning
435 sign, usually of poor code. “Tricky code” is a code phrase for “bad code.” If you
436 think code is tricky, think about rewriting it so that it’s not.

437 A class's having more errors than average is a warning sign. A few error-prone
438 classes tend to be the most expensive part of a program. If you have a class that
439 has had more errors than average, it will probably continue to have more errors
440 than average. Think about rewriting it.

441 If programming were a science, each warning sign would imply a specific, well-
442 defined corrective action. Because programming is still a craft, however, a
443 warning sign merely points to an issue that you should consider. You can't
444 necessarily rewrite tricky code or improve an error-prone class.

445 Just as an abnormal number of defects in a class warns you that the class has low
446 quality, an abnormal number of defects in a program implies that your process is
447 defective. A good process wouldn't allow error-prone code to be developed. It
448 would include the checks and balances of architecture followed by architecture
449 reviews, design followed by design reviews, and code followed by code reviews.
450 By the time the code was ready for testing, most errors would have been
451 eliminated. Exceptional performance requires working smart in addition to
452 working hard. Lots of debugging on a project is a warning sign that implies
453 people aren't working smart. Writing a lot of code in a day and then spending
454 two weeks debugging it is not working smart.

455 You can use design metrics as another kind of warning sign. Most design metrics
456 are heuristics that give an indication of the quality of a design. The fact that a
457 class contains more than 7 members doesn't necessarily mean that it's poorly
458 designed, but it's a warning that the class is complicated. Similarly, more than
459 about 10 decision points in a routine, more than three levels of logical nesting, an
460 unusual number of variables, high coupling to other classes, or low class or
461 routine cohesion should raise a warning flag. None of these signs necessarily
462 means that a class is poorly designed, but the presence of any of them should
463 cause you to look at the class skeptically.

464 Any warning sign should cause you to doubt the quality of your program. As
465 Charles Saunders Peirce says, "Doubt is an uneasy and dissatisfied state from
466 which we struggle to free ourselves and pass into the state of belief." Treat a
467 warning sign as an "irritation of doubt" that prompts you to look for the more
468 satisfied state of belief.

469 If you find yourself working on repetitious code or making similar modifications
470 in several areas, you should feel "uneasy and dissatisfied," doubting that control
471 has been adequately centralized in classes or routines. If you find it hard to create
472 scaffolding for test cases because you can't use an individual class easily, you
473 should feel the "irritation of doubt" and ask whether the class is coupled too
474 tightly to other classes. If you can't reuse code in other programs because some

475 classes are too interdependent, that's another warning sign that the classes are
476 coupled too tightly.

477 When you're deep into a program, pay attention to warning signs that indicate
478 that part of the program design isn't defined well enough to code. Difficulties in
479 writing comments, naming variables, and decomposing the problem into
480 cohesive classes with clear interfaces all indicate that you need to think harder
481 about the design before coding. Wishy-washy names and difficulty in describing
482 sections of code in concise comments are other signs of trouble. When the design
483 is clear in your mind, the low-level details come easily.

484 Be sensitive to indications that your program is hard to understand. Any
485 discomfort is a clue. If it's hard for you, it will be even harder for the next
486 programmers. They'll appreciate the extra effort you make to improve it. If
487 you're figuring out code instead of reading it, it's too complicated. If it's hard,
488 it's wrong. Make it simpler.

489 **HARD DATA**
490 If you want to take full advantage of warning signs, program in such a way that
491 you create your own warnings. This is useful because even after you know what
492 the signs are, it's surprisingly easy to overlook them. Glenford Myers conducted
493 a study of defect correction in which he found that the single most common
494 cause of not finding errors was simply overlooking them. The errors were visible
on test output but not noticed (Myers 1978b).

495 Make it hard to overlook problems in your program. One example is setting
496 pointers to NULL after you free them so that they'll cause ugly problems if you
497 mistakenly use one. A freed pointer might point to a valid memory location even
498 after it's been freed. Setting it to NULL guarantees that it points to an invalid
499 location, making the error harder to overlook.

500 Compiler warnings are literal warning signs that are often overlooked. If your
501 program generates warnings or errors, fix it so that it doesn't. You don't have
502 much chance of noticing subtle warning signs when you're ignoring those that
503 have "WARNING" printed directly on them.

504 Why is paying attention to intellectual warning signs especially important in
505 software development? The quality of the thinking that goes into a program
506 largely determines the quality of the program, so paying attention to warnings
507 about the quality of thinking directly affects the final product.

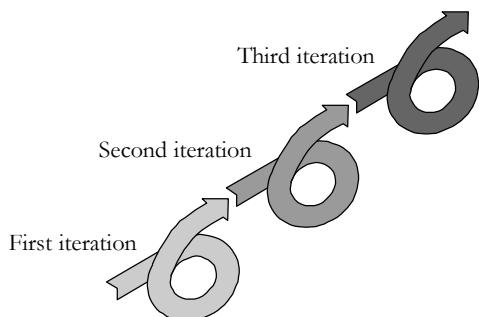
34.8 Iterate, Repeatedly, Again and Again

Iteration is appropriate for many software-development activities. During your initial specification of a system, you work with the user through several versions of requirements until you're sure you agree on them. That's an iterative process. When you build flexibility into your process by building and delivering a system in several increments, that's an iterative process. If you use prototyping to develop several alternative solutions quickly and cheaply before crafting the final product, that's another form of iteration. Iterating on requirements is perhaps as important as any other aspect of the software development process. Projects fail because they commit themselves to a solution before exploring alternatives. Iteration provides a way to learn about a product before you build it.

As Chapter 28 on managing construction points out, during initial project planning, schedule estimates can vary greatly depending on the estimation technique you use. Using an iterative approach for estimation produces a more accurate estimate than relying on a single technique.

Software design is a heuristic process and, like all heuristic processes, is subject to iterative revision and improvement. Software tends to be validated rather than proven, which means that it's tested and developed iteratively until it answers questions correctly. Both high-level and low-level design attempts should be repeated. A first attempt might produce a solution that works, but it's unlikely to produce the best solution. Taking several repeated and different approaches produces insight into the problem that's unlikely with a single approach.

The idea of iteration appears again in code tuning. Once the software is operational, you can rewrite small parts of it to greatly improve overall system performance. Many of the attempts at optimization, however, hurt the code more than they help it. It's not an intuitive process, and some techniques that seem likely to make a system smaller and faster actually make it larger and slower. The uncertainty about the effect of any optimization technique creates a need for tuning, measuring, and tuning again. If a bottleneck is critical to system performance, you can tune the code several times, and several of your later attempts may be more successful than your first.



F34xx02

Figure 34-2

Iteration helps improve requirements, planning, design, code quality, and performance.

Reviews cut across the grain of the development process, inserting iterations at any stage in which they're conducted. The purpose of a review is to check the quality of the work at a particular point. If the product fails the review, it's sent back for rework. If it succeeds, it doesn't need further iteration.

When you take iteration to the extreme, you get Fred Brooks's advice: "Build one to throw away; you will, anyhow" (Brooks 1995). One definition of engineering is to do for a dime what anyone can do for a dollar. Throwing a whole system away is doing for two dollars what anyone can do for one dollar. The trick of software engineering is to build the disposable version as quickly and inexpensively as possible, which is the point of iterating in the early stages.

34.9 Thou Shalt Rend Software and Religion Asunder

Religion appears in software development in numerous incarnations—as dogmatic adherence to a single design method, as unswerving belief in a specific formatting or commenting style, as a zealous avoidance of global data. It's always inappropriate.

Software Oracles

Unfortunately, the religious attitude is decreed from on high by some of the more prominent people in the profession. It's important to publicize innovations so that practitioners can try out promising new methods. Methods have to be tried before they can be fully proven or disproved. The dissemination of research results to practitioners is called "technology transfer" and is important for advancing the state of the practice of software development. There's a

539

540

541

542

543

544

545

546

547

548

549

550

551

552

553

554

555

556

557

558

559

560

561 **CROSS-REFERENCE** For details on handling programming religion as a manager, see "Religious Issues" in Section 28.5.

565

566

567 difference, however, between disseminating a new methodology and selling
568 software snake oil. The idea of technology transfer is poorly served by dogmatic
569 methodology peddlers who try to convince you that their new one-size-fits-all,
570 high-tech cow pies will solve all your problems. Forget everything you've
571 already learned because this new method is so great it will improve your
572 productivity 100 percent in everything!

573 Rather than latching on to the latest miracle fad, use a mixture of methods.
574 Experiment with the exciting, recent methods, but bank on the old and
575 dependable ones.

576 Eclecticism

577 **CROSS-REFERENCE** For
578 more on the difference
579 between algorithmic and
580 heuristic approaches, see
581 Section 2.2, "How to Use
582 Software Metaphors." For
583 information on eclecticism in
584 design, see "Iterate" in
585 Section 5.4.
586

Blind faith in one method precludes the selectivity you need if you're to find the most effective solutions to programming problems. If software development were a deterministic, algorithmic process, you could follow a rigid methodology to your solution. Software development isn't a deterministic process, however. It's heuristic—which means that rigid processes are inappropriate and have little hope of success. In design, for example, sometimes top-down decomposition works well. Sometimes an object-oriented approach, a bottom-up composition, or a data-structure approach works better. You have to be willing to try several approaches, knowing that some will fail and some will succeed but not knowing which ones will work until after you try them. You have to be eclectic.

587 Adherence to a single method is also harmful in that it makes you force-fit the
588 problem to the solution. If you decide on the solution method before you fully
589 understand the problem, you act prematurely. You over-constrain the set of
590 possible solutions, and you might rule out the most effective solution.

591 You'll be uncomfortable with any new methodology initially, and the advice that
592 you avoid religion in programming isn't meant to suggest that you should stop
593 using a new method as soon as you have a little trouble solving a problem with
594 it. Give the new method a fair shake, but give the old methods their fair shakes
595 too.

596 **CROSS-REFERENCE** For
597 a more detailed description of
598 the toolbox metaphor, see
599 "Applying Software
600 Techniques: The Intellectual
601 Toolbox" in Section 2.3.
602
603
604

Eclecticism is a useful attitude to bring to the techniques presented in this book as much as to techniques described in other sources. Discussions of several topics have advanced alternative approaches that you can't use at the same time. You have to choose one or the other for each specific problem. You have to treat the techniques as tools in a toolbox and use your own judgment to select the best tool for the job. Most of the time, the tool choice doesn't matter very much. You can use a box wrench, vise-grip pliers, or a crescent wrench. In some cases, however, the tool selection matters a lot, so you should always make your selection carefully. Engineering is in part a discipline of making trade-offs

605 among competing techniques. You can't make a trade-off if you've prematurely
606 limited your choices to a single tool.

607 The toolbox metaphor is useful because it makes the abstract idea of eclecticism
608 concrete. Suppose you were a general contractor and your buddy Simple Simon
609 always used vise-grip pliers. Suppose he refused to use a box wrench or a
610 crescent wrench. You'd probably think he was odd because he wouldn't use all
611 the tools at his disposal. The same is true in software development. At a high
612 level, you have alternative design methods. At a more detailed level, you can
613 choose one of several data types to represent any given design. At an even more
614 detailed level, you can choose several different schemes for formatting and
615 commenting code, naming variables, defining class interfaces, and passing
616 routine parameters.

617 A dogmatic stance conflicts with the eclectic toolbox approach to software
618 construction. It's incompatible with the attitude needed to build high-quality
619 software.

620 **Experimentation**

621 Eclecticism has a close relative in experimentation. You need to experiment
622 throughout the development process, but the religious attitude hobbles the
623 impulse. To experiment effectively, you must be willing to change your beliefs
624 based on the results of the experiment. If you're not willing, experimentation is a
625 gratuitous waste of time.

626 Many of the religious approaches to software development are based on a fear of
627 making mistakes. A blanket attempt to avoid mistakes is the biggest mistake of
628 all. Design is a process of carefully planning small mistakes in order to avoid
629 making big ones. Experimentation in software development is a process of
630 setting up tests so that you learn whether an approach fails or succeeds—the
631 experiment itself is a success as long as it resolves the issue.

632 Experimentation is appropriate at as many levels as eclecticism is. At each level
633 at which you are ready to make an eclectic choice, you can probably come up
634 with a corresponding experiment to determine which approach works best. At
635 the architectural-design level, an experiment might consist of sketching software
636 architectures using three different design approaches. At the detailed-design
637 level, an experiment might consist of following the implications of a higher-level
638 architecture using three different low-level design approaches. At the
639 programming-language level, an experiment might consist of writing a short
640 experimental program to exercise the operation of a part of the language you're
641 not completely familiar with. The experiment might consist of tuning a piece of
642 code and benchmarking it to verify that it's really smaller or faster. At the

643 overall software-development–process level, an experiment might consist of
644 collecting quality and productivity data so that you can see whether inspections
645 really find more errors than walkthroughs.

646 The point is that you have to keep an open mind about all aspects of software
647 development. Rather than being religious, you have to get technical about your
648 process as well as your product. Open-minded experimentation and religious
649 adherence to a predefined approach don't mix.

650 Key Points

- 651 • One primary goal of programming is managing complexity.
- 652 • The programming process significantly affects the final product.
- 653 • Team programming is more an exercise in communicating with people than
654 in communicating with a computer. Individual programming is more an
655 exercise in communicating with yourself than with a computer.
- 656 • When abused, a programming convention can be a cure that's worse than the
657 disease. Used thoughtfully, a convention adds valuable structure to the
658 development environment and helps with managing complexity and
659 communication.
- 660 • Programming in terms of the problem rather than the solution helps to
661 manage complexity.
- 662 • Paying attention to intellectual warning signs like the “irritation of doubt” is
663 especially important in programming because programming is almost purely
664 a mental activity.
- 665 • The more you iterate in each development activity, the better the product of
666 that activity will be.
- 667 • Dogmatic methodologies and high-quality software development don't mix.
668 Fill your intellectual toolbox with programming alternatives and improve
669 your skill at choosing the right tool for the job.

35

Where to Find More Information

Contents

- 35.1 Information About Software Construction
- 35.2 Topics Beyond Construction
- 35.3 Periodicals
- 35.4 A Software Developer's Reading Plan
- 35.5 Joining a Professional Organization

Related Topics

List of Additional Resource sections: page TBD

Web resources: www.cc2e.com

IF YOU'VE READ THIS FAR, you already know that a lot has been written about effective software development practices. Much more information is available than most people realize. People have already made all the mistakes that you're making now, and unless you're a glutton for punishment, you'll prefer reading their books and avoiding their mistakes to inventing new versions of old problems.

Because this book describes hundreds of other books and articles that contain articles on software development, it's hard to know what to read first. A software-development library is made up of several kinds of information. A core of programming books explains fundamental concepts of effective programming. Related books explain the larger technical, management, and intellectual context within which programming goes on. And detailed references on the languages, operating systems, environments, and hardware contain information that's useful for specific projects.

Books in the last category generally have a life span of about one project; they're more or less temporary and aren't discussed here.

1

2

3

4 CC2E.COM/3560

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

CC2E.COM/3581

29 Of the other kinds of books, it's useful to have a core set that discusses each of
30 the major software-development activities in depth—books on requirements,
31 design, construction, management, testing, and so on. The following sections
32 describe construction resources in depth, and then provide an overview of mate-
33 rials available in other software knowledge areas. Section 35.4 wraps these re-
34 sources into a neat package by defining a software developer's reading program.

35.1 Information About Software Construction

I originally wrote this book because I couldn't find a thorough discussion of software construction. In the years since I published the first edition, several good books have appeared.

Pragmatic Programmer (Hunt and Thomas 2000) focuses on the activities most closely associated with coding including testing, debugging, use of assertions, and so on. It does not dive deeply into code itself, but contains numerous principles related to creating good code.

Jon Bentley's *Programming Pearls, 2d Ed* (Bentley 2000) discusses the art and science of software design in the small. The book is organized as a set of essays that are very well written and express a great deal of insight into effective construction techniques as well as genuine enthusiasm for software construction. I use something I learned from Bentley's essays nearly every day that I program.

Kent Beck's *Extreme Programming Explained: Embrace Change* (Beck 2000) defines a construction-centric approach to software development. As Section 3.1 explained, the book's assertions about the economics of extreme programming are not borne out by industry research, but many of its recommendations are useful during construction regardless of whether a team is using extreme programming or some other approach.

A more specialized book is Steve Maguire's *Writing Solid Code – Microsoft's Techniques for Developing Bug-Free C Software* (Maguire 1993). It focuses on construction practices for commercial-quality software applications, mostly based on the author's experiences working on Microsoft's Office applications. It focuses on techniques applicable in C. It is largely oblivious to object-oriented programming issues, but most of the topics it addresses are relevant in any environment.

Another more specialized book is *The Practice of Programming* by Brian Kernighan and Rob Pike (Kernighan and Pike 1999). This book focuses on nitty gritty, practical aspects of programming, bridging the gap between academic

65 computer science knowledge and hands-on lessons. It includes discussions of
66 programming style, design, debugging, and testing. It assumes familiarity with
67 C/C++.

68 Although it's out of print and hard to find, *Programmers at Work* by Susan
69 Lammers (1986) is worth the effort if can find it. It contains interviews with the
70 industry's high-profile programmers. The interviews explore their personalities,
71 work habits, and programming philosophies. The luminaries interviewed include
72 Bill Gates (founder of Microsoft), John Warnock (founder of Adobe), Andy
73 Hertzfeld (principal developer of the Macintosh operating system), Butler
74 Lampson (a senior engineer at DEC, now at Microsoft), Wayne Ratliff (inventor
75 of dBase), Dan Bricklin (inventor of VisiCalc), and a dozen others.

76 **35.2 Topics Beyond Construction**

77 Beyond the core books described in the last section, here are some books that
78 range further afield from the topic of software construction.

79 CC2E.COM/3595

Overview Material

80 Robert L. Glass's *Facts and Fallacies of Software Engineering* (2003) provides a
81 readable introduction to the conventional wisdom of software development dos
82 and don'ts. The book is well researched and provides numerous pointers to addi-
83 tional resources.

84 My own *Professional Software Development* (2004) surveys the field of software
85 development as it is practiced now and as it could be if it were routinely prac-
86 ticed at its best.

87 The *Swebok: Guide to the Software Engineering Body of Knowledge* (Abran
88 2001) provides a detailed decomposition of the software engineering body of
89 knowledge. This book has dived into detail in the software construction area.
90 The Guide to the Swebok shows just how much more knowledge exists in the
91 field.

92 Gerald Weinberg's *The Psychology of Computer Programming* (Weinberg 1998)
93 is packed with fascinating anecdotes about programming. It's far-ranging be-
94 cause it was written at a time when anything related to software was considered
95 to be about programming. The advice in the original review of the book in the
96 *ACM Computing Reviews* is as good today as it was when the review was writ-
97 ten:

98 *Every manager of programmers should have his own
99 copy. He should read it, take it to heart, act on the precepts,
100 and leave the copy on his desk to be stolen by his program-
101 mers. He should continue replacing the stolen copies until
102 equilibrium is established (Weiss 1972).*

103 If you can't find *The Psychology of Computer Programming*, look for *The
104 Mythical Man-Month* (Brooks 1995) or *PeopleWare* (DeMarco and Lister 1999).
105 They both drive home the theme that programming is first and foremost some-
106 thing done by people and only secondarily something that happens to involve
107 computers.

108 A final excellent overview of issues in software development is *Software Crea-
109 tivity* (Glass 1995). This book should have been a breakthrough book on soft-
110 ware creativity the way that *Peopleware* was on software teams. Glass discusses
111 creativity versus discipline, theory versus practice, heuristics versus methodol-
112 ogy, process versus product, and many of the other dichotomies that define the
113 software field. After years of discussing this book with programmers who work
114 for me, I have concluded that the difficulty with the book is that it is a collection
115 of essays edited by Glass, but not entirely written by him. For some readers, this
116 gives the book an unfinished feel. Nonetheless, I still require every developer in
117 my company to read it. The book is out of print and hard to find, but worth the
118 effort if you are able to find it.

119 **Software-Engineering Overviews**

120 Every practicing computer programmer or software engineer should have a high-
121 level reference on software engineering. Such books survey the methodological
122 landscape rather than painting specific features in detail. They provide an over-
123 view of effective software-engineering practices and capsule descriptions of spe-
124 cific software-engineering techniques. The capsule descriptions aren't detailed
125 enough to train you in the techniques, but a single book would have to be several
126 thousand pages long to do that. They provide enough information so that you can
127 learn how the techniques fit together and can choose techniques for further in-
128 vestigation.

129 Roger S. Pressman's *Software Engineering: A Practitioner's Approach, 6th Ed.*
130 (Pressman 2004) is a balanced treatment of requirements, design, quality valida-
131 tion, and management. Its 700 pages pay little attention to programming prac-
132 tices, but that is a minor limitation, especially if you already have a book on con-
133 struction such as the one you're reading.

134
135
136
The 6th edition of Ian Sommerville's *Software Engineering* (Sommerville 2000)
is comparable to Pressman's book, and it also provides a good high-level over-
view of the software-development process.

137 CC2E.COM/3502

Other Annotated Bibliographies

138 Good computing bibliographies are rare. Here are a few that justify the effort it
139 takes to obtain them.

140 *ACM Computing Reviews* is a special-interest publication of the ACM that's
141 dedicated to reviewing books about all aspects of computers and computer pro-
142 gramming. The reviews are organized according to an extensive classification
143 scheme, making it easy to find books in your area of interest. For information on
144 this publication and on membership in the ACM, write: ACM, PO Box 12114,
145 Church Street Station, New York, NY 10257.

146 CC2E.COM/3509
147 Construx Software's Professional Development Ladder
148 (www.construx.com/ladder/). This website provides recommended reading pro-
grams for software developers, testers, and managers.

35.3 Periodicals

Lowbrow Programmer Magazines

150 These magazines are often available at local newsstands.

152 CC2E.COM/3516
153 *Software Development*. www.sdmagazine.com. This magazine focuses on pro-
154 gramming issues—less on tips for specific environments than on the general is-
155 sues you face as a professional programmer. The quality of the articles is quite
good. It also includes product reviews.

156 CC2E.COM/3523
157 *Dr. Dobb's Journal*. www.ddj.com. This magazine is oriented toward hard-core
158 programmers. Its articles tend to deal with detailed issues and include lots of
code.

159 If you can't find these magazines at your local newsstand, many publishers will
160 send you a complimentary issue, and many articles are available on line.

Highbrow Programmer Journals

161 You don't usually buy these magazines at the newsstand. You usually have to go
162 to a major university library or subscribe to them for yourself or your company.
163

CC2E.COM/3530

164
165
166
167
168
169
170

IEEE Software. www.computer.org/software/. This bimonthly magazine focuses on software construction, management, requirements, design and other leading-edge software topics. Its mission is to “build the community of leading software practitioners.” In 1993, I wrote that it’s “the most valuable magazine a programmer can subscribe to.” Since I wrote that, I’ve been Editor in Chief of the magazine, and I still believe it’s the best periodical available for a serious software practitioner.

171 CC2E.COM/3537
172
173
174
175

IEEE Computer. www.computer.org/computer/. This monthly magazine is the flagship publication of the IEEE Computer Society. It publishes articles on a wide spectrum of computer topics and has scrupulous review standards to ensure the quality of the articles it publishes. Because of its breadth, you’ll probably find fewer articles that interest you than you will in *IEEE Software*.

176 CC2E.COM/3544
177
178
179
180
181
182
183
184

Communications of the ACM. www.acm.org/cacm/. This magazine is one of the oldest and most respected computer publications available. It has the broad charter of publishing about the length and breadth of computerology, a subject that’s much vaster than it was even a few years ago. As with *IEEE Computer*, because of its breadth, you’ll probably find that many of the articles are outside your area of interest. The magazine tends to have an academic flavor, which has both a bad side and a good side. The bad side is that some of the authors write in an obfuscatory academic style. The good side is that it contains leading-edge information that won’t filter down to the lowbrow magazines for years.

185 186 **Special-Interest Publications**

Several publications provide in-depth coverage of specialized topics.

187 188 CC2E.COM/3551 189 190 191 **Professional Publications**

The IEEE Computer Society publishes specialized journals on software engineering, security and privacy, computer graphics and animation, internet development, multimedia, intelligent systems, the history of computing, and other topics. See www.computer.org for more details.

192 CC2E.COM/3558
193
194
195

The ACM also publishes special-interest publications in artificial intelligence, computers and human interaction, databases, embedded systems, graphics, programming languages, mathematical software, networking, software engineering, and other topics. See www.acm.org for more information.

196 197 **Popular-Market Publications**

These magazines all cover what their names suggest they cover.

198 CC2E.COM/3565
The C/C++ Users Journal. www.cuj.com.

- 199 CC2E.COM/3572 *Java Developer's Journal.* www.sys-con.com/java/.
- 200 CC2E.COM/3579 *Embedded Systems Programming.* www.embedded.com.
- 201 CC2E.COM/3586 *Linux Journal.* www.linuxjournal.com.
- 202 CC2E.COM/3593 *Unix Review.* www.unixreview.com
- 203 CC2E.COM/3500 *Windows Developer's Network.* www.wd-mag.com.

204 35.4 A Software Developer's Reading Plan

205 CC2E.COM/3507 This section describes the reading program that a software developer needs to
206 work through to achieve full professional standing at my company, Construx
207 Software. The plan described is a generic baseline plan for a software profes-
208 sional who wants to focus on development. Our mentoring program provides for
209 further tailoring of the generic plan to support an individual's interests, and
210 within Construx this reading is also supplemented with training and directed pro-
211 fessional experiences.

212 **Introductory Level**

213 To move beyond "introductory" level at Construx, a developer must read the
214 following books.

215 Adams, James L. *Conceptual Blockbusting: A Guide to Better Ideas*, 4th ed.
216 Cambridge, Mass.: Perseus Publishing.

217 Bentley, Jon. *Programming Pearls*, 2d Ed. Reading, Mass.: Addison-Wesley,
218 2000.

219 Glass, Robert L. *Facts and Fallacies of Software Engineering*, Boston, Mass.:
220 Addison Wesley, 2003.

221 McConnell, Steve. *Software Project Survival Guide*. Redmond, WA: Microsoft
222 Press, 1998.

223 McConnell, Steve. *Code Complete*, 2d Ed.. Redmond, WA: Microsoft Press,
224 2004.

225 **Practitioner Level**

226 To achieve "intermediate" status at Construx, a programmer needs to read the
227 following additional materials.

- 228 Berczuk, Stephen P. and Brad Appleton. *Software Configuration Management*
229 *Patterns: Effective Teamwork, Practical Integration*, Boston, Mass.: Addison
230 Wesley, 2003.
- 231 Fowler, Martin. *UML Distilled: A Brief Guide to the Standard Object Modeling*
232 *Language, 3d Ed*, Boston, Mass.: Addison Wesley, 2003.
- 233 Glass, Robert L. *Software Creativity*, Reading, Mass.: Addison Wesley, 1995.
- 234 Kaner, Cem, Jack Falk, Hung Q. Nguyen. *Testing Computer Software, 2d Ed.*,
235 New York: John Wiley & Sons, 1999.
- 236 Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented*
237 *Analysis and Design and the Unified Process*, 2d Ed., Englewood Cliffs, N.J.:
238 Prentice Hall, 2001.
- 239 McConnell, Steve. *Rapid Development*. Redmond, WA: Microsoft Press, 1996.
- 240 Wiegers, Karl. *Software Requirements*, 2d Ed. Redmond, WA: Microsoft Press,
241 2003.
- 242 CC2E.COM/3514 “Manager’s Handbook for Software Development”, NASA Goddard Space
243 Flight Center. Downloadable from sel.gsfc.nasa.gov/website/documents/online-doc.htm.
- 244

245 **Professional Level**

246 A software developer must read the following materials to achieve full profes-
247 sional standing at Construx (“leadership” level). Additional requirements are
248 tailored to each individual developer; this section describes the generic require-
249 ments.

250 Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*,
251 Second Edition, Boston, Mass.: Addison Wesley, 2003.

252 Fowler, Martin. *Refactoring: Improving the Design of Existing Code*, Reading,
253 Mass.: Addison Wesley, 1999.

254 Gamma, Erich, et al. *Design Patterns*, Reading, Mass.: Addison Wesley, 1995.

255 Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, Eng-
256 land: Addison-Wesley.

257 Maguire, Steve. *Writing Solid Code*. Redmond, WA: Microsoft Press, 1993.

- 258 Meyer, Bertrand. *Object-Oriented Software Construction, 2d Ed.* New York:
259 Prentice Hall PTR, 1997.
- 260 CC2E.COM/3521 “Software Measurement Guidebook”, NASA Goddard Space Flight Center.
261 Available from sel.gsfc.nasa.gov/website/documents/online-doc.htm.
- 262 CC2E.COM/3528 For more details on this professional development program, as well as for up-to-
263 date reading lists, see our professional development website at
264 www.construx.com/professionaldev/.

265 35.5 Joining a Professional Organization

- 266 CC2E.COM/3535 One of the best ways to learn more about programming is to get in touch with
267 other programmers who are as dedicated to the profession as you are. Local user
268 groups for specific hardware and language products are one kind of group. Other
269 kinds are national and international professional organizations. The most practi-
270 tioner-oriented organization is the Computer Society of the IEEE (Institute of
271 Electrical and Electronics Engineers). The IEEE Computer Society publishes the
272 *IEEE Computer* and *IEEE Software* magazines. For membership information,
273 see www.computer.org.
- 274 CC2E.COM/3542 The original professional organization was the Association for Computing Ma-
275 chinery, or ACM. The ACM publishes *Communications of the ACM* and many
276 special-interest magazines. It tends to be somewhat more academically oriented
277 than the IEEE Computer Society. For membership information, see
278 www.acm.org.