



Code Complete, Second Edition

By Steve McConnell

Publisher: **Microsoft Press**

Pub Date: **June 09, 2004**

Print ISBN: **0-7356-1967-0**

Pages: **960**

[Table of Contents](#) | [Index](#)

Overview

Take a strategic approach to software construction—and produce superior products—with this fully updated edition of Steve McConnell's critically praised and award-winning guide to software development best practices.



Code Complete, Second Edition

By Steve McConnell

Publisher: Microsoft Press

Pub Date: June 09, 2004

Print ISBN: 0-7356-1967-0

Pages: 960

[Table of Contents](#) | [Index](#)

[Copyright](#)

[Preface](#)

[Who Should Read This Book?](#)

[Where Else Can You Find This Information?](#)

[Key Benefits of This Handbook](#)

[Why This Handbook Was Written](#)

[Author Note](#)

[Acknowledgments](#)

[About the Author](#)

[Steve McConnell](#)

[Part I: Laying the Foundation](#)

[In this part:](#)

[Chapter 1. Welcome to Software Construction](#)

[Section 1.1. What Is Software Construction?](#)

[Section 1.2. Why Is Software Construction Important?](#)

[Section 1.3. How to Read This Book](#)

[Key Points](#)

[Chapter 2. Metaphors for a Richer Understanding of Software Development](#)

[Section 2.1. The Importance of Metaphors](#)

[Section 2.2. How to Use Software Metaphors](#)

[Section 2.3. Common Software Metaphors](#)

[Key Points](#)

[Chapter 3. Measure Twice, Cut Once: Upstream Prerequisites](#)

[Section 3.1. Importance of Prerequisites](#)

[Section 3.2. Determine the Kind of Software You're Working On](#)

[Section 3.3. Problem-Definition Prerequisite](#)

[Section 3.4. Requirements Prerequisite](#)

[Section 3.5. Architecture Prerequisite](#)

[Section 3.6. Amount of Time to Spend on Upstream Prerequisites](#)

[Additional Resources](#)

[Key Points](#)

[Chapter 4. Key Construction Decisions](#)

[Section 4.1. Choice of Programming Language](#)

[Section 4.2. Programming Conventions](#)

[Section 4.3. Your Location on the Technology Wave](#)

[Section 4.4. Selection of Major Construction Practices](#)

[Key Points](#)

[Part II: Creating High-Quality Code](#)

[In this part:](#)

[Chapter 5. Design in Construction](#)

[Section 5.1. Design Challenges](#)

[Section 5.2. Key Design Concepts](#)
[Section 5.3. Design Building Blocks: Heuristics](#)
[Section 5.4. Design Practices](#)
[Section 5.5. Comments on Popular Methodologies](#)
[Additional Resources](#)
[Key Points](#)

[Chapter 6. Working Classes](#)

[Section 6.1. Class Foundations: Abstract Data Types \(ADTs\)](#)
[Section 6.2. Good Class Interfaces](#)
[Section 6.3. Design and Implementation Issues](#)
[Section 6.4. Reasons to Create a Class](#)
[Section 6.5. Language-Specific Issues](#)
[Section 6.6. Beyond Classes: Packages](#)
[Additional Resources](#)
[Key Points](#)

[Chapter 7. High-Quality Routines](#)

[Section 7.1. Valid Reasons to Create a Routine](#)
[Section 7.2. Design at the Routine Level](#)
[Section 7.3. Good Routine Names](#)
[Section 7.4. How Long Can a Routine Be?](#)
[Section 7.5. How to Use Routine Parameters](#)
[Section 7.6. Special Considerations in the Use of Functions](#)
[Section 7.7. Macro Routines and Inline Routines](#)
[Key Points](#)

[Chapter 8. Defensive Programming](#)

[Section 8.1. Protecting Your Program from Invalid Inputs](#)
[Section 8.2. Assertions](#)
[Section 8.3. Error-Handling Techniques](#)
[Section 8.4. Exceptions](#)
[Section 8.5. Barricade Your Program to Contain the Damage Caused by Errors](#)
[Section 8.6. Debugging Aids](#)
[Section 8.7. Determining How Much Defensive Programming to Leave in Production Code](#)
[Section 8.8. Being Defensive About Defensive Programming](#)
[Additional Resources](#)
[Key Points](#)

[Chapter 9. The Pseudocode Programming Process](#)

[Section 9.1. Summary of Steps in Building Classes and Routines](#)
[Section 9.2. Pseudocode for Pros](#)
[Section 9.3. Constructing Routines by Using the PPP](#)
[Section 9.4. Alternatives to the PPP](#)
[Key Points](#)

[Part III: Variables](#)

[In this part:](#)

[Chapter 10. General Issues in Using Variables](#)

[Section 10.1. Data Literacy](#)
[Section 10.2. Making Variable Declarations Easy](#)
[Section 10.3. Guidelines for Initializing Variables](#)
[Section 10.4. Scope](#)
[Section 10.5. Persistence](#)
[Section 10.6. Binding Time](#)
[Section 10.7. Relationship Between Data Types and Control Structures](#)
[Section 10.8. Using Each Variable for Exactly One Purpose](#)
[Key Points](#)

[Chapter 11. The Power of Variable Names](#)

[Section 11.1. Considerations in Choosing Good Names](#)

[Section 11.2. Naming Specific Types of Data](#)
[Section 11.3. The Power of Naming Conventions](#)
[Section 11.4. Informal Naming Conventions](#)
[Section 11.5. Standardized Prefixes](#)
[Section 11.6. Creating Short Names That Are Readable](#)
[Section 11.7. Kinds of Names to Avoid](#)
[Key Points](#)

[Chapter 12. Fundamental Data Types](#)

[Section 12.1. Numbers in General](#)
[Section 12.2. Integers](#)
[Section 12.3. Floating-Point Numbers](#)
[Section 12.4. Characters and Strings](#)
[Section 12.5. Boolean Variables](#)
[Section 12.6. Enumerated Types](#)
[Section 12.7. Named Constants](#)
[Section 12.8. Arrays](#)
[Section 12.9. Creating Your Own Types \(Type Aliasing\)](#)
[Key Points](#)

[Chapter 13. Unusual Data Types](#)

[Section 13.1. Structures](#)
[Section 13.2. Pointers](#)
[Section 13.3. Global Data](#)
[Additional Resources](#)
[Key Points](#)

[Part IV: Statements](#)

[In this part:](#)

[Chapter 14. Organizing Straight-Line Code](#)

[Section 14.1. Statements That Must Be in a Specific Order](#)
[Section 14.2. Statements Whose Order Doesn't Matter](#)
[Key Points](#)

[Chapter 15. Using Conditionals](#)

[Section 15.1. if Statements](#)
[Section 15.2. case Statements](#)
[Key Points](#)

[Chapter 16. Controlling Loops](#)

[Section 16.1. Selecting the Kind of Loop](#)
[Section 16.2. Controlling the Loop](#)
[Section 16.3. Creating Loops Easily—From the Inside Out](#)
[Section 16.4. Correspondence Between Loops and Arrays](#)
[Key Points](#)

[Chapter 17. Unusual Control Structures](#)

[Section 17.1. Multiple Returns from a Routine](#)
[Section 17.2. Recursion](#)
[Section 17.3. goto](#)
[Section 17.4. Perspective on Unusual Control Structures](#)
[Additional Resources](#)
[Key Points](#)

[Chapter 18. Table-Driven Methods](#)

[Section 18.1. General Considerations in Using Table-Driven Methods](#)
[Section 18.2. Direct Access Tables](#)
[Section 18.3. Indexed Access Tables](#)
[Section 18.4. Stair-Step Access Tables](#)
[Section 18.5. Other Examples of Table Lookups](#)
[Key Points](#)

[Chapter 19. General Control Issues](#)

- [Section 19.1. Boolean Expressions](#)
- [Section 19.2. Compound Statements \(Blocks\)](#)
- [Section 19.3. Null Statements](#)
- [Section 19.4. Taming Dangerously Deep Nesting](#)
- [Section 19.5. A Programming Foundation: Structured Programming](#)
- [Section 19.6. Control Structures and Complexity](#)
- [Key Points](#)

[Part V: Code Improvements](#)

[In this part:](#)

- [Chapter 20. The Software-Quality Landscape](#)
 - [Section 20.1. Characteristics of Software Quality](#)
 - [Section 20.2. Techniques for Improving Software Quality](#)
 - [Section 20.3. Relative Effectiveness of Quality Techniques](#)
 - [Section 20.4. When to Do Quality Assurance](#)
 - [Section 20.5. The General Principle of Software Quality](#)
 - [Additional Resources](#)
 - [Key Points](#)

[Chapter 21. Collaborative Construction](#)

- [Section 21.1. Overview of Collaborative Development Practices](#)
- [Section 21.2. Pair Programming](#)
- [Section 21.3. Formal Inspections](#)
- [Section 21.4. Other Kinds of Collaborative Development Practices](#)
- [Comparison of Collaborative Construction Techniques](#)
- [Additional Resources](#)
- [Key Points](#)

[Chapter 22. Developer Testing](#)

- [Section 22.1. Role of Developer Testing in Software Quality](#)
- [Section 22.2. Recommended Approach to Developer Testing](#)
- [Section 22.3. Bag of Testing Tricks](#)
- [Section 22.4. Typical Errors](#)
- [Section 22.5. Test-Support Tools](#)
- [Section 22.6. Improving Your Testing](#)
- [Section 22.7. Keeping Test Records](#)
- [Additional Resources](#)
- [Key Points](#)

[Chapter 23. Debugging](#)

- [Section 23.1. Overview of Debugging Issues](#)
- [Section 23.2. Finding a Defect](#)
- [Section 23.3. Fixing a Defect](#)
- [Section 23.4. Psychological Considerations in Debugging](#)
- [Section 23.5. Debugging Tools—Obvious and Not-So-Obvious](#)
- [Additional Resources](#)
- [Key Points](#)

[Chapter 24. Refactoring](#)

- [Section 24.1. Kinds of Software Evolution](#)
- [Section 24.2. Introduction to Refactoring](#)
- [Section 24.3. Specific Refactorings](#)
- [Section 24.4. Refactoring Safely](#)
- [Section 24.5. Refactoring Strategies](#)
- [Additional Resources](#)
- [Key Points](#)

[Chapter 25. Code-Tuning Strategies](#)

- [Section 25.1. Performance Overview](#)
- [Section 25.2. Introduction to Code Tuning](#)
- [Section 25.3. Kinds of Fat and Molasses](#)

- [Section 25.4. Measurement](#)
- [Section 25.5. Iteration](#)
- [Section 25.6. Summary of the Approach to Code Tuning](#)
- [Additional Resources](#)
- [Key Points](#)
- [Chapter 26. Code-Tuning Techniques](#)
 - [Section 26.1. Logic](#)
 - [Section 26.2. Loops](#)
 - [Section 26.3. Data Transformations](#)
 - [Section 26.4. Expressions](#)
 - [Section 26.5. Routines](#)
 - [Section 26.6. Recoding in a Low-Level Language](#)
 - [Section 26.7. The More Things Change, the More They Stay the Same](#)
 - [Additional Resources](#)
 - [Key Points](#)
- [Part VI: System Considerations](#)
 - [In this part:](#)
 - [Chapter 27. How Program Size Affects Construction](#)
 - [Section 27.1. Communication and Size](#)
 - [Section 27.2. Range of Project Sizes](#)
 - [Section 27.3. Effect of Project Size on Errors](#)
 - [Section 27.4. Effect of Project Size on Productivity](#)
 - [Section 27.5. Effect of Project Size on Development Activities](#)
 - [Additional Resources](#)
 - [Key Points](#)
 - [Chapter 28. Managing Construction](#)
 - [Section 28.1. Encouraging Good Coding](#)
 - [Section 28.2. Configuration Management](#)
 - [Section 28.3. Estimating a Construction Schedule](#)
 - [Section 28.5. Treating Programmers as People](#)
 - [Section 28.6. Managing Your Manager](#)
 - [Key Points](#)
 - [Chapter 29. Integration](#)
 - [Section 29.1. Importance of the Integration Approach](#)
 - [Section 29.2. Integration Frequency—Phased or Incremental?](#)
 - [Section 29.3. Incremental Integration Strategies](#)
 - [Section 29.4. Daily Build and Smoke Test](#)
 - [Additional Resources](#)
 - [Key Points](#)
 - [Chapter 30. Programming Tools](#)
 - [Section 30.1. Design Tools](#)
 - [Section 30.2. Source-Code Tools](#)
 - [Section 30.3. Executable-Code Tools](#)
 - [Section 30.4. Tool-Oriented Environments](#)
 - [Section 30.5. Building Your Own Programming Tools](#)
 - [Section 30.6. Tool Fantasyland](#)
 - [Additional Resources](#)
 - [Key Points](#)
 - [Part VII: Software Craftsmanship](#)
 - [In this part:](#)
 - [Chapter 31. Layout and Style](#)
 - [Section 31.1. Layout Fundamentals](#)
 - [Section 31.2. Layout Techniques](#)
 - [Section 31.3. Layout Styles](#)
 - [Section 31.4. Laying Out Control Structures](#)

[Section 31.5. Laying Out Individual Statements](#)

[Section 31.6. Laying Out Comments](#)

[Section 31.7. Laying Out Routines](#)

[Section 31.8. Laying Out Classes](#)

[Additional Resources](#)

[Key Points](#)

[Chapter 32. Self-Documenting Code](#)

[Section 32.1. External Documentation](#)

[Section 32.2. Programming Style as Documentation](#)

[Section 32.3. To Comment or Not to Comment](#)

[Section 32.4. Keys to Effective Comments](#)

[Section 32.5. Commenting Techniques](#)

[Section 32.6. IEEE Standards](#)

[Additional Resources](#)

[Key Points](#)

[Chapter 33. Personal Character](#)

[Section 33.1. Isn't Personal Character Off the Topic?](#)

[Section 33.2. Intelligence and Humility](#)

[Section 33.3. Curiosity](#)

[Section 33.4. Intellectual Honesty](#)

[Section 33.5. Communication and Cooperation](#)

[Section 33.6. Creativity and Discipline](#)

[Section 33.7. Laziness](#)

[Section 33.8. Characteristics That Don't Matter As Much As You Might Think](#)

[Section 33.9. Habits](#)

[Additional Resources](#)

[Key Points](#)

[Chapter 34. Themes in Software Craftsmanship](#)

[Section 34.1. Conquer Complexity](#)

[Section 34.2. Pick Your Process](#)

[Section 34.3. Write Programs for People First, Computers Second](#)

[Section 34.4. Program into Your Language, Not in It](#)

[Section 34.5. Focus Your Attention with the Help of Conventions](#)

[Section 34.6. Program in Terms of the Problem Domain](#)

[Section 34.7. Watch for Falling Rocks](#)

[Section 34.8. Iterate, Repeatedly, Again and Again](#)

[Section 34.9. Thou Shalt Rend Software and Religion Asunder](#)

[Key Points](#)

[Chapter 35. Where to Find More Information](#)

[Section 35.1. Information About Software Construction](#)

[Section 35.2. Topics Beyond Construction](#)

[Section 35.3. Periodicals](#)

[Section 35.4. A Software Developer's Reading Plan](#)

[Section 35.5. Joining a Professional Organization](#)

[Bibliography](#)

[Index](#)

[◀ PREVIOUS](#)

[*< Free Open Study >*](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Copyright

PUBLISHED BY

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

Copyright ± 2004 by Steven C. McConnell

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

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

Code Complete / Steve McConnell.—2nd ed.

p. cm.

Includes index.

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

QA76.76.D47M39 2004

005.1—dc22 2004049981

1 2 3 4 5 6 7 8 9 QWT 8 7 6 5 4 3

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

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

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

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

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

Acquisitions Editors: Linda Engelman and Robin Van Steenburgh

Project Editor: Devon Musgrave

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Preface

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

—Fred Brooks

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

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

Who Should Read This Book?

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

Experienced Programmers

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

Technical Leads

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

Self-Taught Programmers

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

Students

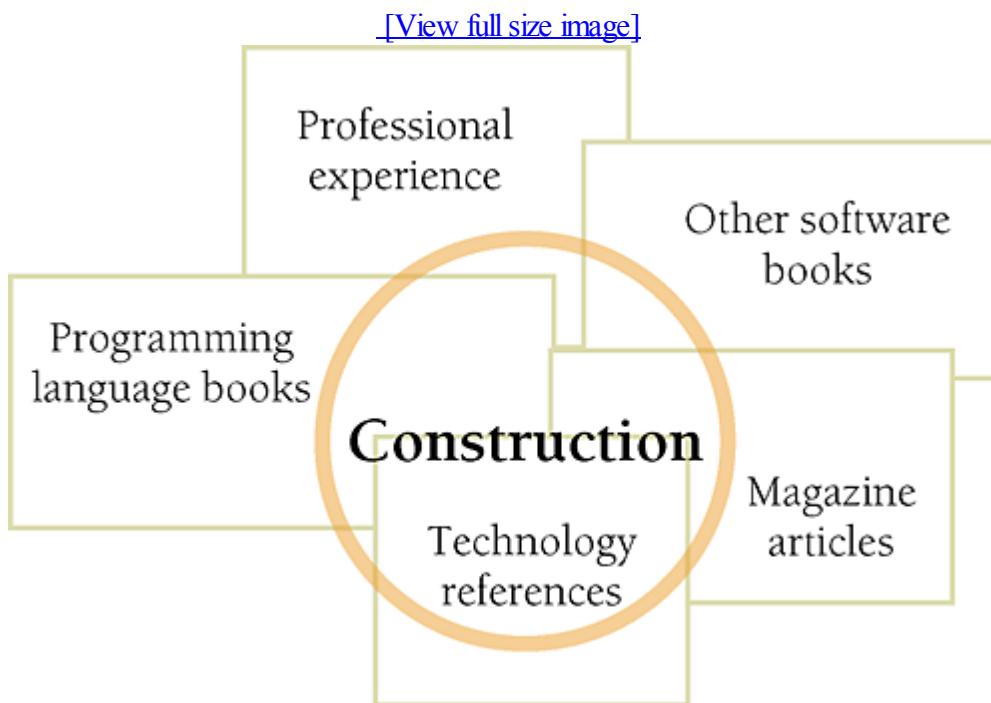
The counterpoint to the programmer with experience but little formal training is the fresh college graduate. The recent graduate is often rich in theoretical knowledge but poor in the practical know-how that goes into building production programs. The practical lore of good coding is often passed down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

Where Else Can You Find This Information?

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

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

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



[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Benefits of This Handbook

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

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

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

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

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

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

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

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

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

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

Access to other sources of information This book collects much of the available information on software construction, but it's hardly the last word. Throughout the chapters, "[Additional Resources](#)" sections describe other books and articles you can read as you pursue the topics you find most interesting.

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Why This Handbook Was Written

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

The Topic of Construction Has Been Neglected

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

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

Construction Is Important

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

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

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

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

No Comparable Book Is Available

In light of construction's obvious importance, I was sure when I conceived this book that someone else would already have written a book on effective construction practices. The need for a book about how to program effectively seemed obvious. But I found that only a few books had been written about construction and then only on parts of the topic. Some had been written 15 years or more earlier and employed relatively esoteric languages such as ALGOL, PL/I, Ratfor, and Smalltalk. Some were written by professors who were not working on production code. The professors wrote about techniques that worked for student projects, but they often had little idea of how the techniques could be applied to real-world projects. Still, the lack of comparable books is

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Author Note

I welcome your inquiries about the topics discussed in this book, your error reports, or other related subjects. Please contact me at steveme@construx.com, or visit my website at <http://www.stevemeconnell.com>.

Bellevue, Washington

Memorial Day, 2004

Microsoft Learning Technical Support

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

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

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

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

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

Postal Mail:

Microsoft Press

Attn: Code Complete 2E Editor

One Microsoft Way

Redmond, WA 98052-6399

E-mail:

mspinput@microsoft.com

Acknowledgments

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

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

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

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

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

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

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

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

About the Author

[Steve McConnell](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Steve McConnell

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



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

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Part I: Laying the Foundation

[In this part:](#)

[Chapter 1. Welcome to Software Construction](#)

[Chapter 2. Metaphors for a Richer Understanding of Software Development](#)

[Chapter 3. Measure Twice, Cut Once: Upstream Prerequisites](#)

[Chapter 4. Key Construction Decisions](#)

In this part:

[Chapter 1](#): Welcome to Software Construction

[Chapter 2](#): Metaphors for a Richer Understanding of Software Development

[Chapter 3](#): Measure Twice, Cut Once: Upstream Prerequisites

[Chapter 4](#): Key Construction Decisions

Chapter 1. Welcome to Software Construction

cc2e.com/0178

Contents

- [What Is Software Construction? page 3](#)
- [Why Is Software Construction Important? page 6](#)
- [How to Read This Book page 8](#)

Related Topics

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

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

1.1. What Is Software Construction?

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

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

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

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

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

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

1.2. Why Is Software Construction Important?

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

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

Here's why:

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

Cross-Reference

For details on the relationship between project size and the percentage of time consumed by construction, see "[Activity Proportions and Size](#)" in [Section 27.5](#).

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

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.

Cross-Reference

For data on variations among programmers, see "[Individual Variation](#)" in [Section 28.5](#).

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 KEY POINT 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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

1.3. How to Read This Book

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

Key Points

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

Chapter 2. Metaphors for a Richer Understanding of Software Development

cc2e.com/0278

Contents

- [The Importance of Metaphors page 9](#)
- [How to Use Software Metaphors page 11](#)
- [Common Software Metaphors page 13](#)

Related Topic

- Heuristics in design: "[Design Is a Heuristic Process](#)" in [Section 5.1](#)

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

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

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

2.1. The Importance of Metaphors

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

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

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

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

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

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

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

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

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

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).

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

2.2. How to Use Software Metaphors



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

KEY POINT

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

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

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

Cross-Reference

For details on how to use heuristics in designing software, see "[Design Is a Heuristic Process](#)" in [Section 5.1](#).

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

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

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

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

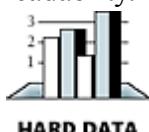
2.3. Common Software Metaphors

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

Software Penmanship: Writing Code

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

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

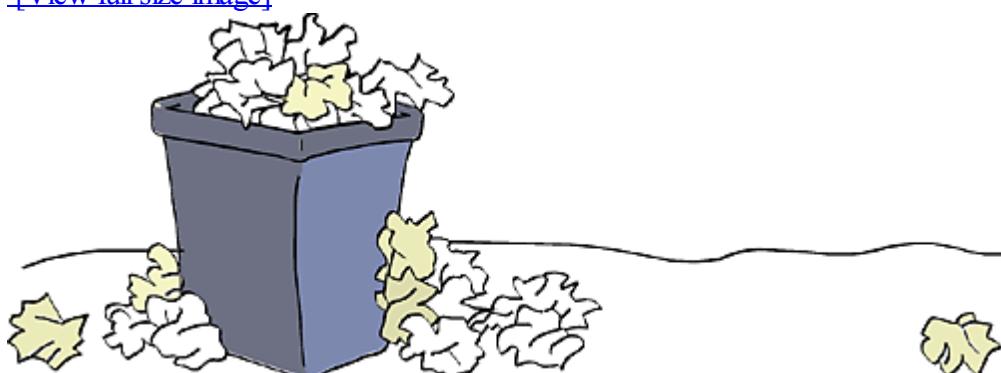


For an individual's work or for small-scale projects, the letter-writing metaphor works adequately, but for other purposes it leaves the party early—it doesn't describe software development fully or adequately. Writing is usually a one-person activity, whereas a software project will most likely involve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

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

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

[\[View full size image\]](#)



[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 3. Measure Twice, Cut Once: Upstream Prerequisites

cc2e.com/0309

Contents

- [Importance of Prerequisites page 24](#)
- [Determine the Kind of Software You're Working On page 31](#)
- [Problem-Definition Prerequisite page 36](#)
- [Requirements Prerequisite page 38](#)
- [Architecture Prerequisite page 43](#)
- [Amount of Time to Spend on Upstream Prerequisites page 55](#)

Related Topics

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

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

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

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

3.1. Importance of Prerequisites

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.

Cross-Reference

Paying attention to quality is also the best way to improve productivity. For details, see [Section 20.5, "The General Principle of Software Quality."](#)

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.

KEY POINT

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

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

Do Prerequisites Apply to Modern Software Projects?

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

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

—Harlan Mills



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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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

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

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

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

	Kind of Software		
	Business Systems	Mission-Critical Systems	Embedded Life-Critical Systems
Typical applications	Internet site	Embedded software	Avionics software
	Intranet site	Games	Embedded software
	Inventory management	Internet site	Medical devices
	Games	Packaged software	Operating systems
	Management information systems	Software tools	Packaged software
	Payroll system	Web services	
Life-cycle models	Agile development (Extreme Programming, Scrum, timebox 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
Requirements	Informal requirements specification	Semiformal requirements specification	Formal requirements specification
		As-needed requirements reviews	Formal requirements inspections

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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," "vision statement," "mission statement," or "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.

If the "box" is the boundary of constraints and conditions, then the trick is to find the box.... Don't think outside the box—find the box.

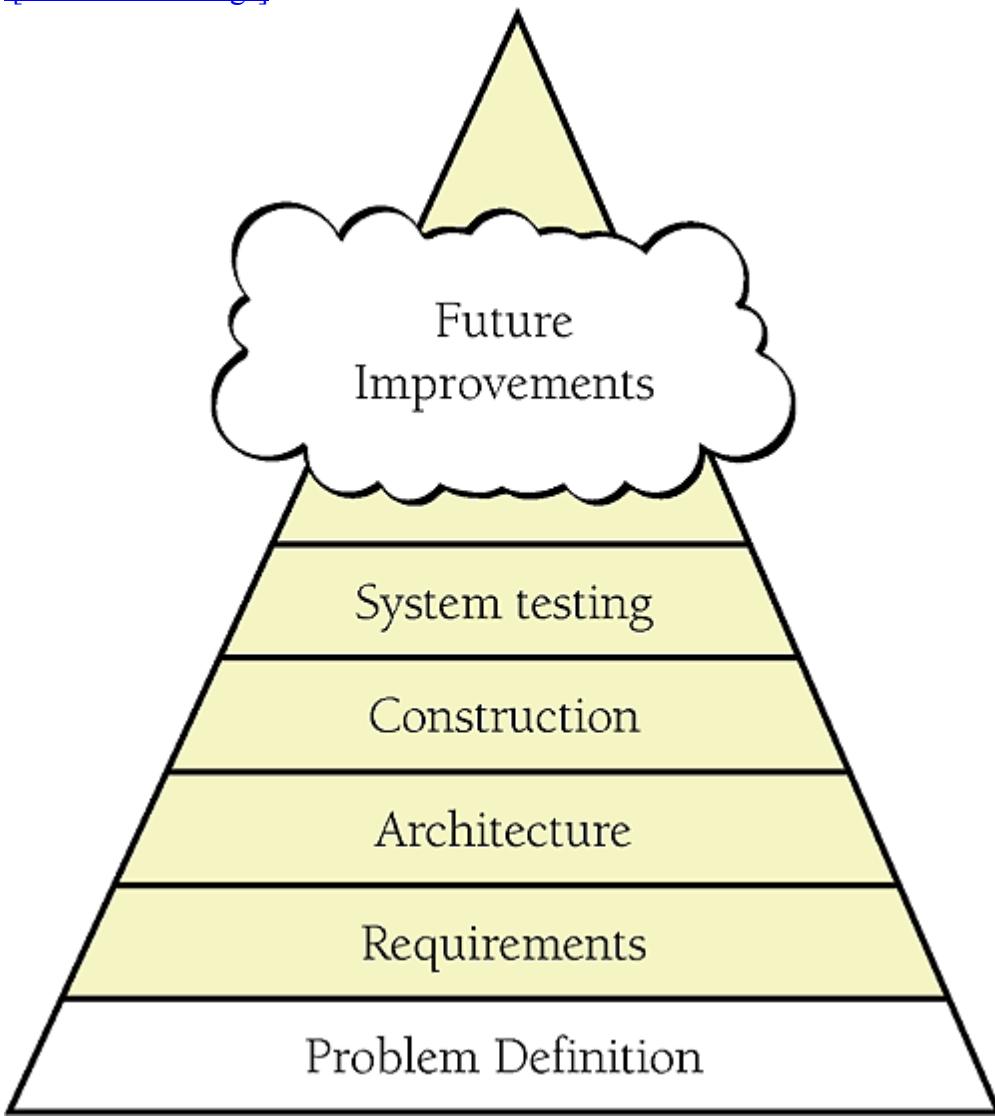
—Andy Hunt and Dave Thomas

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

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

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

[\[View full size image\]](#)



[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

3.4. Requirements Prerequisite

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

Why Have Official Requirements?

An explicit set of requirements is important for several reasons.

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



KEY POINT Explicit requirements also help to avoid arguments. You decide on the scope of the system before you begin programming. If you have a disagreement with another programmer about

what the program is supposed to do, you can resolve it by looking at the written requirements.

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

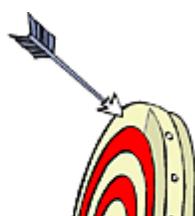


As [Table 3-1](#) reported, data from numerous organizations indicates that on large projects an error in requirements detected during the architecture stage is typically 3 times as expensive to correct as it would be if it were detected during the requirements stage. If detected during coding, it's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

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

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

[\[View full size image\]](#)



[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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

Cross-Reference

For more information on design at all levels, see [Chapters 5 through 9](#).

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

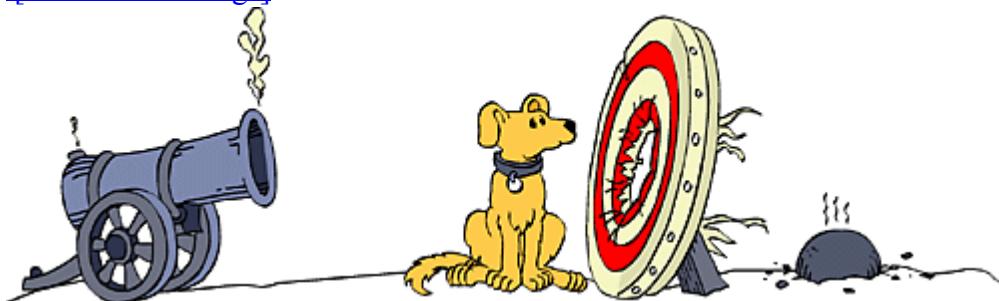


KEY POINT Why have architecture as a prerequisite? Because the quality of the architecture determines the conceptual integrity of the system. That in turn determines the ultimate quality of the system. A well-thought-out architecture provides the structure needed to maintain a system's conceptual integrity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

Good architecture makes construction easy. Bad architecture makes construction almost impossible. [Figure 3-7](#) illustrates another problem with bad architecture.

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

[\[View full size image\]](#)



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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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.

Cross-Reference

The amount of time you spend on prerequisites will depend on your project type. For details on adapting prerequisites to your specific project, see [Section 3.2, "Determine the Kind of Software You're Working On,"](#) earlier in this chapter.

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, you'll probably need to resolve requirements issues yourself. 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.

Cross-Reference

For approaches to handling changing requirements, see "[Handling Requirements Changes During Construction](#)" in [Section 3.4,](#) earlier in this chapter.

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/0344

Following are more resources on requirements:

Requirements

cc2e.com/0351

Here are a few books that give much more detail on requirements development:

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

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

Gilb, Tom. Competitive Engineering. Reading, MA: Addison-Wesley, 2004. This book describes Gilb's requirements language, known as "Planguage." The book covers Gilb's specific approach to requirements engineering, design and design evaluation, and evolutionary project management. This book can be downloaded from Gilb's website at <http://www.gilb.com>.

cc2e.com/0358

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

Abran, Alain, et al. Swebok: Guide to the Software Engineering Body of Knowledge. Los Alamitos, CA: IEEE Computer Society Press, 2001. This contains a detailed description of the body of software-requirements knowledge. It can also be downloaded from <http://www.swebok.org>.

cc2e.com/0365

Other good alternatives include the following:

Lauesen, Soren. Software Requirements: Styles and Techniques. Boston, MA: Addison-Wesley, 2002.

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

Cockburn, Alistair. Writing Effective Use Cases. Boston, MA: Addison-Wesley, 2000.

Software Architecture

cc2e.com/0372

Numerous books on software architecture have been published in the past few years. Here are some of the best:

Bass, Len, Paul Clements, and Rick Kazman. Software Architecture in Practice, 2d ed. Boston, MA: Addison-Wesley, 2003.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 on 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 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.

Chapter 4. Key Construction Decisions

cc2e.com/0489

Contents

- [Choice of Programming Language page 61](#)
- [Programming Conventions page 66](#)
- [Your Location on the Technology Wave page 66](#)
- [Selection of Major Construction Practices page 69](#)

Related Topics

- Upstream prerequisites: [Chapter 3](#)
- Determine the kind of software you're working on: [Section 3.2](#)
- How program size affects construction: [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](#), "[Measure Twice, Cut Once: Upstream Prerequisites](#)," discussed the software equivalent of blueprints and construction permits. You might not have had much control over those preparations, so the focus of that chapter was on assessing what you have to work with when 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 job site.

If you feel you've read enough about construction preparations already, you might skip ahead to [Chapter 5](#), "[Design in Construction](#)."

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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

—Alfred North Whitehead

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

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

Programmers are more productive using a familiar language than an unfamiliar one. Data from the Cocomo II estimation model shows that programmers working in a language they've used for three years or more are about 30 percent more productive than programmers with equivalent experience who are new to a language (Boehm et al. 2000). An earlier study at IBM found that programmers who had extensive experience with a programming language were more than three times as productive as those with minimal experience (Walston and Felix 1977). (Cocomo II is more careful to isolate effects of individual factors, which accounts for the different results of the two studies.)



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

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

Language	Level Relative to C
C	1
C++	2.5
Fortran 95	2
Java	2.5
Perl	6
Python	6

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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.

Cross-Reference

For more details on the power of conventions, see [Sections 11.3](#) through [11.5](#).

In a complex program, architectural guidelines give the program structural balance and construction guidelines provide low-level harmony, articulating each class as a faithful part of a comprehensive design. Any large program requires a controlling structure that unifies its programming-language details. Part of the beauty of a large structure is the way in which its detailed parts bear out the implications of its architecture. Without a unifying discipline, your creation will be a jumble of sloppy variations in style. Such variations tax your brain—and only for the sake of understanding coding-style differences that are essentially arbitrary. One key to successful programming is avoiding arbitrary variations so that your brain can be free to focus on the variations that are really needed. For more on this, see "[Software's Primary Technical Imperative: Managing Complexity](#)" in [Section 5.2](#).

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



Before construction begins, spell out the programming conventions you'll use.

KEY POINT Coding-convention details are at such a level of precision that they're nearly impossible to retrofit into software after it's written. Details of such conventions are provided throughout the book.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

4.3. Your Location on the Technology Wave

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

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

In early-wave environments—Web programming in the mid-1990s, for example—the situation is the opposite. Few programming language choices are available, and those languages tend to be buggy and poorly documented. Programmers spend significant amounts of time simply trying to figure out how the language works instead of writing new code. Programmers also spend countless hours working around bugs in the language products, underlying operating system, and other tools. Programming tools in early-wave environments tend to be primitive. Debuggers might not exist at all, and compiler optimizers are still only a gleam in some programmer's eye. Vendors revise their compiler version often, and it seems that each new version breaks significant parts of your code. Tools aren't integrated, and so you tend to work with different tools for UI, database, reports, and business logic. The tools tend not to be very compatible, and you can expend a significant amount of effort just to keep existing functionality working against the onslaught of compiler and library releases. If you run into trouble, reference literature exists on the Web in some form, but it isn't always reliable and, if the available literature is any guide, every time you encounter a problem it seems as though you're the first one to do so.

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

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

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

Example of Programming into a Language

In the early days of Visual Basic, I was frustrated because I wanted to keep the business logic, the UI, and the database separate in the product I was developing, but there wasn't any built-in way to do that in the language. I knew that if I wasn't careful, over time some of my Visual Basic "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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 combination of techniques 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 these practices are contained throughout the book.

cc2e.com/0496

Checklist: Major Construction Practices

Coding

- Have you defined how much design will be done up front and how much will be done at the keyboard, while the code is being written?
- Have you defined coding conventions for names, comments, and layout?
- 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, what conventions will be used for class interfaces, what standards will apply to reused code, how much to consider performance while coding, 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?

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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.
- Ask yourself whether the programming practices you're using are a response to the programming language you're using or controlled by it. Remember to program into the language, rather than programming in it.
- 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.

Part II: Creating High-Quality Code

[In this part:](#)

[Chapter 5. Design in Construction](#)

[Chapter 6. Working Classes](#)

[Chapter 7. High-Quality Routines](#)

[Chapter 8. Defensive Programming](#)

[Chapter 9. The Pseudocode Programming Process](#)

[◀ PREVIOUS](#)[< Free Open Study >](#)[NEXT ▶](#)**In this part:**

[Chapter 5](#): Design in Construction

[Chapter 6](#): Working Classes

[Chapter 7](#): High-Quality Routines

[Chapter 8](#): Defensive Programming

[Chapter 9](#): The Pseudocode Programming Process

[◀ PREVIOUS](#)[< Free Open Study >](#)[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 5. Design in Construction

cc2e.com/0578

Contents

- [Design Challenges page 74](#)
- [Key Design Concepts page 77](#)
- [Design Building Blocks: Heuristics page 87](#)
- [Design Practices page 110](#)
- [Comments on Popular Methodologies page 118](#)

Related Topics

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

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

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

5.1. Design Challenges

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

Cross-Reference

The difference between heuristic and deterministic processes is described in [Chapter 2, "Metaphors for a Richer Understanding of Software Development."](#)

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

Design Is a Wicked Problem

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

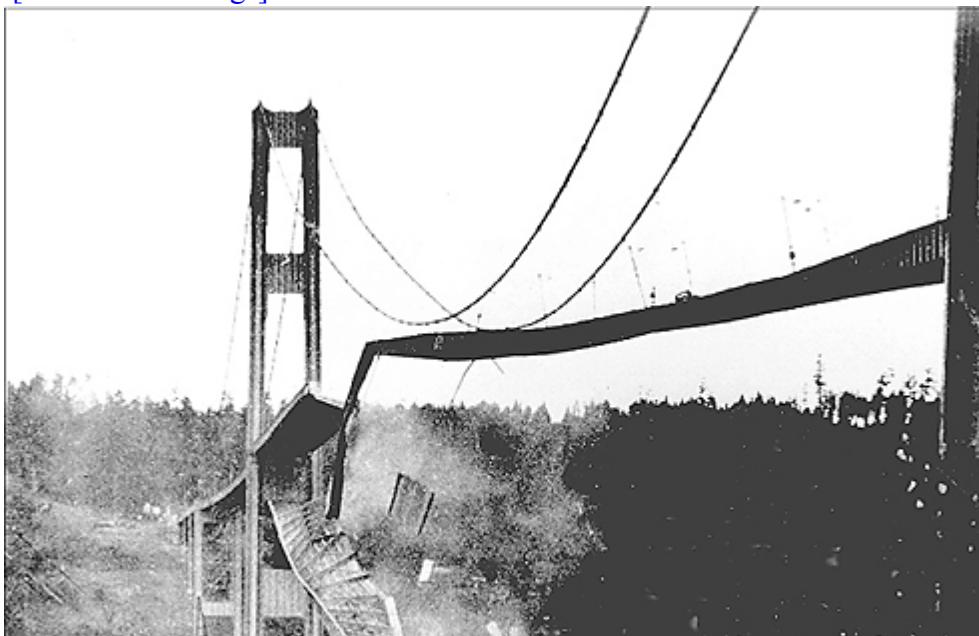
The picture of the software designer deriving his design in a rational, error-free way from a statement of requirements is quite unrealistic. No system has ever been developed in that way, and probably none ever will. Even the small program developments shown in textbooks and papers are unreal. They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

—David Parnas and Paul Clements

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

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

[\[View full size image\]](#)



[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 Brooks's landmark paper, "No Silver Bullets: Essence and Accidents of Software Engineering" (1987).

Cross-Reference

For discussion of the way complexity affects programming issues other than design, see [Section 34.1, "Conquer Complexity."](#)

Accidental and Essential Difficulties

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

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

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

Cross-Reference

Accidental difficulties are more prominent in early-wave development than in late-wave development. For details, see [Section 4.3, "Your Location on the Technology Wave."](#)

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

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

Importance of Managing Complexity

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

5.3. Design Building Blocks: Heuristics

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

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

Find Real-World Objects

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.

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

—Bertrand Meyer

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 is allowed to do to other objects.
- Determine the parts of each object that will be visible to other objects—which parts will be public and which will be private.
- Define each object's public interface.

Cross-Reference

For more details on designing using classes, see [Chapter 6, "Working Classes."](#)

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

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

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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.

KEY POINT

As you cycle through candidate designs and try different approaches, you'll look at both high-level and low-level views. The big picture you get from working with high-level issues will help you to put the low-level details in perspective. The details you get from working with low-level issues will provide a foundation in solid reality for the high-level decisions. The tug and pull between top-level and bottom-level considerations is a healthy dynamic; it creates a stressed structure that's more stable than one built wholly from the top down or the bottom up.

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

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.

Cross-Reference

Refactoring is a safe way to try different alternatives in code. For more on this, see [Chapter 24, "Refactoring."](#)

Divide and Conquer

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

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

Top-Down and Bottom-Up Design Approaches

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 *i* and crossing every *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!"

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

—P. J. Plauger

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

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

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/0520

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

Software Design, General

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

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

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

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

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

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

Software Design Theory

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

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

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

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

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

Design Patterns

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 6. Working Classes

cc2e.com/0665

Contents

- [Class Foundations: Abstract Data Types \(ADTs\) page 126](#)
- [Good Class Interfaces page 133](#)
- [Design and Implementation Issues page 143](#)
- [Reasons to Create a Class page 152](#)
- [Language-Specific Issues page 156](#)
- [Beyond Classes: Packages page 156](#)

Related Topics

- Design in construction: [Chapter 5](#)
- Software architecture: [Section 3.5](#)
- High-quality routines: [Chapter 7](#)
- The Pseudocode Programming Process: [Chapter 9](#)
- Refactoring: [Chapter 24](#)

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.



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

This chapter contains a distillation of advice in creating high-quality classes. If you're still warming up to object-oriented concepts, this chapter might be too advanced. Make sure you've read [Chapter 5, "Design in Construction."](#) Then start with [Section 6.1, "Class Foundations: Abstract Data Types \(ADTs\)"](#) and ease your way

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

6.1. Class Foundations: Abstract Data Types (ADTs)

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.

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.

Cross-Reference

Thinking about ADTs first and classes second is an example of programming into a language vs. programming in one. See [Section 4.3, "Your Location on the Technology Wave,"](#) and [Section 34.4, "Program into Your Language, Not in It."](#)

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!

Example of the Need for an ADT

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

Suppose you're writing a program to control text output to the screen using a variety of typefaces, point sizes, and font attributes (such as bold and italic). Part of the program manipulates the text's fonts. If you use an ADT, you'll have a group of font routines bundled with the data—the typeface names, point sizes, and font attributes—they operate on. The collection of font routines and data is an ADT.

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

```
currentFont.size = 16
```

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

```
currentFont.size = PointsToPixels( 12 )
```

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

```
currentFont.sizeInPixels = PointsToPixels( 12 )
```

But what you can't do is have both `currentFont.sizeInPixels` and `currentFont.sizeInPoints`, because, if both the data members are in play, `currentFont` won't have any way to know which of the two it should use. And if you change sizes in several places in the program, you'll have similar lines spread throughout your program.

If you need to set a font to bold, you might have code like this that uses a logical or and a hexadecimal constant `0x02`:

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 that the details remain hidden behind the abstraction.

Good Abstraction

As "[Form Consistent Abstractions](#)" in [Section 5.3](#) described, 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 GetName() const;  
  
    String GetAddress() const;  
  
    String GetWorkPhone() const;  
  
    String GetHomePhone() const;  
  
    TaxId GetTaxIdNumber() const;  
  
    JobClassification GetJobClassification() const;  
  
    ...  
  
private:  
  
    ...  
};
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

6.3. Design and Implementation Issues

Defining good class interfaces goes a long way toward creating a high-quality program. The internal class design and implementation are also important. This section discusses issues related to containment, inheritance, member functions and data, class coupling, constructors, and value-vs.-reference objects.

Containment ("has a" Relationships)



KEY POINT Containment is the simple idea that a class contains a primitive data element or object. A lot more is written about inheritance than about containment, but that's 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 One way of thinking of containment is as a "has a" relationship. For example, an employee "has a" name, "has a" phone number, "has a" tax ID, and so on. You can usually accomplish this by making the name, phone number, and tax ID member data of the Employee class.

Implement "has a" through private inheritance as a last resort In some instances you might find that you can't achieve containment through making one object a member of another. In that case, some experts suggest privately inheriting from the contained object (Meyers 1998, Sutter 2000). The main reason you would do that is to set up the containing class to access protected member functions or protected member data of the class that's contained. In practice, this approach creates an overly cozy relationship with the ancestor class and violates encapsulation. It tends to point to design errors that should be resolved some way other than through private inheritance.

Be critical of classes that contain more than about seven data members The number "7 \pm 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 \pm 2 if the data members are primitive data types like integers and strings, more toward the lower end of 7 \pm 2 if the data members are complex objects.

Inheritance ("is a" Relationships)

Inheritance is the idea that one class is a specialization of another class. 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. Inheritance helps avoid the need to repeat code and data in multiple locations by centralizing it within a base class.

When you decide to use inheritance, you have to make several decisions:

- 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?

The following subsections explain the ins and outs of making these decisions:

Implement "is a" through public inheritance When a programmer decides to create a new class by inheriting from an existing class, that programmer is saying that the new class "is a" more specialized version of the older class. The base class sets expectations about how the derived class will operate and imposes constraints on how the derived class can operate (Meyers 1998).

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

6.4. Reasons to Create a Class

If you believe everything you read, you might get the idea that the only reason to create a class is to model real-world objects. In practice, classes get created for many more reasons than that. Here's a list of good reasons to create a class.

Cross-Reference

Reasons for creating classes and routines overlap. See [Section 7.1](#).

Model real-world objects Modeling real-world objects might not be the only reason to create a class, but it's still a good reason! Create a class for each real-world object type that your program models. Put the data needed for the object into the class, and then build service routines that model the behavior of the object. See the discussion of ADTs in [Section 6.1](#) for examples.

Cross-Reference

For more on identifying real-world objects, see "[Find Real-World Objects](#)" in [Section 5.3](#).

Model abstract objects Another good reason to create a class is to model an abstract object—an object that isn't a concrete, real-world object but that provides an abstraction of other concrete objects. A good example is the classic Shape object. Circle and Square really exist, but Shape is an abstraction of other specific shapes.

On programming projects, the abstractions are not ready-made the way Shape is, so we have to work harder to come up with clean abstractions. The process of distilling abstract concepts from real-world entities is non-deterministic, and different designers will abstract out different generalities. If we didn't know about geometric shapes like circles, squares and triangles, for example, we might come up with more unusual shapes like squash shape, rutabaga shape, and Pontiac Aztek shape. Coming up with appropriate abstract objects is one of the major challenges in object-oriented design.



Reduce complexity **KEY POINT** 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.

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 better, simpler, or more reliable algorithm, it will be easier to replace the old algorithm if it has been isolated into a class. During development, it will be easier to try several designs and keep the one that works best.

Hide implementation details The desire to hide implementation details is a wonderful reason to create a class whether the details are as complicated as a convoluted database access or as mundane as whether a specific data member is stored as a number or a string.

Limit effects of changes Isolate areas that are likely to change so that the effects of changes are limited to the scope of a single class or a few classes. Design so that areas that are most likely to change are the easiest to change. Areas likely to change include hardware dependencies, input/output, complex data types, and business rules. The

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

6.5. Language-Specific Issues

Approaches to classes in different programming languages vary in interesting ways. Consider how you override a member routine to achieve polymorphism in a derived class. In Java, all routines are overridable by default and a routine must be declared final to prevent a derived class from overriding it. In C++, routines are not overridable by default. A routine must be declared virtual in the base class to be overridable. In Visual Basic, a routine must be declared overridable in the base class and the derived class should use the overrides keyword.

Here are some of the class-related areas that vary significantly depending on the language:

- Behavior of overridden constructors and destructors in an inheritance tree
- Behavior of constructors and destructors under exception-handling conditions
- Importance of default constructors (constructors with no arguments)
- Time at which a destructor or finalizer is called
- Wisdom of overriding the language's built-in operators, including assignment and equality
- How memory is handled as objects are created and destroyed or as they are declared and go out of scope

Detailed discussions of these issues are beyond the scope of this book, but the "[Additional Resources](#)" section points to good language-specific resources.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

6.6. Beyond Classes: Packages

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.

Cross-Reference

For more on the distinction between classes and packages, see "[Levels of Design](#)" in [Section 5.2](#).

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. 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 the following:

- 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

These workarounds 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."](#)

cc2e.com/0672

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 233.

CHECKLIST: Class Quality

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?

Abstraction

- Does the class have a central purpose?

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

Classes in General

cc2e.com/0679

Meyer, Bertrand. Object-Oriented Software Construction, 2d ed. New York, NY: Prentice Hall PTR, 1997. This book contains an in-depth discussion of abstract data types and explains how they form the basis for classes. [Chapters 14–16](#) discuss inheritance in depth. Meyer provides an argument in favor of multiple inheritance in [Chapter 15](#).

Riel, Arthur J. Object-Oriented Design Heuristics. Reading, MA: Addison-Wesley, 1996. This book contains numerous suggestions for improving program design, mostly at the class level. I avoided the book for several years because it appeared to be too big—talk about people in glass houses! However, the body of the book is only about 200 pages long. Riel's writing is accessible and enjoyable. The content is focused and practical.

C++

cc2e.com/0686

Meyers, Scott. Effective C++: 50 Specific Ways to Improve Your Programs and Designs, 2d ed. Reading, MA: Addison-Wesley, 1998.

Meyers, Scott, 1996, More Effective C++: 35 New Ways to Improve Your Programs and Designs. Reading, MA: Addison-Wesley, 1996. Both of Meyers' books are canonical references for C++ programmers. The books are entertaining and help to instill a language-lawyer's appreciation for the nuances of C++.

Java

cc2e.com/0693

Bloch, Joshua. Effective Java Programming Language Guide. Boston, MA: Addison-Wesley, 2001. Bloch's book provides much good Java-specific advice as well as introducing more general, good object-oriented practices.

Visual Basic

cc2e.com/0600

The following books are good references on classes in Visual Basic:

Foxall, James. Practical Standards for Microsoft Visual Basic .NET. Redmond, WA: Microsoft Press, 2003.

Cornell, Gary, and Jonathan Morrison. Programming VB .NET: A Guide for Experienced Programmers. Berkeley, CA: Apress, 2002.

Barwell, Fred, et al. Professional VB.NET, 2d ed. Wrox, 2002.

Key Points

- 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 Software's Primary Technical Imperative of managing complexity.
- Classes are your primary tool for managing complexity. Give their design as much attention as needed to accomplish that objective.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 7. High-Quality Routines

cc2e.com/0778

Contents

- [Valid Reasons to Create a Routine page 164](#)
- [Design at the Routine Level page 168](#)
- [Good Routine Names page 171](#)
- [How Long Can a Routine Be? page 173](#)
- [How to Use Routine Parameters page 174](#)
- [Special Considerations in the Use of Functions page 181](#)
- [Macro Routines and Inline Routines page 182](#)

Related Topics

- Steps in routine construction: [Section 9.3](#)
- Working classes: [Chapter 6](#)
- General design techniques: [Chapter 5](#)
- Software architecture: [Section 3.5](#)

[Chapter 6](#) described the 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 issues that affect the design of routines before wading into the nitty-gritty details, be sure to read [Chapter 5, "Design in Construction,"](#) first and come back to this chapter later. Some important attributes of high-quality routines are also discussed in [Chapter 8, "Defensive Programming."](#) 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 Microsoft 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.

What is a high-quality routine? That's a harder question. Perhaps the easiest answer is to show what a high-quality

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

7.1. Valid Reasons to Create a Routine

Here's a list of valid reasons to create a routine. The reasons overlap somewhat, and they're not intended to make an orthogonal set.



Reduce complexity **KEY POINT** The single most important reason to create a routine is to reduce a program's complexity. Create a routine 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 routine. But after it's written, you should be able to forget the details and use the routine without any knowledge of its internal workings. Other reasons to create routines—minimizing code size, improving maintainability, and improving correctness—are also good reasons, but without the abstractive power of routines, complex programs would be impossible to manage intellectually.

One indication that a routine needs to be broken out of another routine is deep nesting of an inner loop or a conditional. Reduce the containing routine's complexity by pulling the nested part out and putting it into its own routine.

Introduce an intermediate, understandable abstraction Putting a section of code into a well-named routine is one of the best ways to document its purpose. Instead of reading a series of statements like

```
if ( node <> NULL ) then  
  
    while ( node.next <> NULL ) do  
  
        node = node.next  
  
        leafName = node.name  
  
    end while  
  
else  
  
    leafName = ""  
  
end if
```

you can read a statement like this:

```
leafName = GetLeafName( node )
```

The new routine is so short that nearly all it needs for documentation is a good name. The name introduces a higher level of abstraction than the original eight lines of code, which makes the code more readable and easier to understand, and it reduces complexity within the routine that originally contained the code.

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 a base class, and then move the two specialized routines into subclasses. Alternatively, you could migrate 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.

Support subclassing You need less new code to override a short, well-factored routine than a long, poorly factored routine. You'll also reduce the chance of error in subclass implementations if you keep overrideable routines simple.

Hide sequences It's a good idea to hide the order in which events happen to be processed. For example, if the

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

7.2. Design at the Routine Level

The idea of cohesion was introduced in a paper by Wayne Stevens, Glenford Myers, and Larry Constantine (1974). Other more modern concepts, including abstraction and encapsulation, tend to yield more insight at the class level (and have, in fact, largely superceded cohesion at the class level), but cohesion is still alive and well as the workhorse design heuristic at the individual-routine level.

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.

Cross-Reference

For a discussion of cohesion in general, see "[Aim for Strong Cohesion](#)" in [Section 5.3](#).



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). Another study of a different 450 routines (which is just an unusual coincidence) found that routines with the highest coupling-to-cohesion ratios had 7 times as many errors as those with the lowest coupling-to-cohesion ratios and were 20 times as costly to fix (Selby and Basili 1991).

Discussions about cohesion typically refer to several levels of cohesion. Understanding the concepts is more important than remembering specific terms. Use the concepts as aids in thinking about how to make routines as cohesive as possible.

Functional cohesion is the strongest and best kind of cohesion, occurring when a routine performs one and only one operation. Examples of highly cohesive routines include `sin()`, `GetCustomerName()`, `EraseFile()`, `CalculateLoanPayment()`, and `AgeFromBirthdate()`. Of course, this evaluation of their cohesion assumes that the routines do what their names say they do—if they do anything else, they are less cohesive and poorly named.

Several other kinds of cohesion are normally considered to be less than ideal:

-

Sequential cohesion exists when a routine contains operations that must be performed in a specific order, that share data from step to step, and that don't make up a complete function when done together.

An example of sequential cohesion is a routine that, given a birth date, calculates an employee's age and time to retirement. If the routine calculates the age and then uses that result to calculate the employee's time to retirement, it has sequential cohesion. If the routine calculates the age and then calculates the time to retirement in a completely separate computation that happens to use the same birth-date data, it has only communicational cohesion.

How would you make the routine functionally cohesive? You'd create separate routines to compute an employee's age given a birth date and compute time to retirement given a birth date. The time-to-retirement routine could call the age routine. They'd both have functional cohesion. Other routines could call either routine or both routines.

-

Communicational cohesion occurs when operations in a routine make use of the same data and aren't related in any other way. If a routine prints a summary report and then reinitializes the summary data passed into it, the routine has communicational cohesion: the two operations are related only by the fact that they use the

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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:

Cross-Reference

For details on naming variables, see [Chapter 11, "The Power of Variable 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. `ComputeReportTotalsAndOpen-OutputFile()` 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, vague, or wishy-washy verbs Some verbs are elastic, stretched to cover just about any meaning. Routine names like `HandleCalculation()`, `PerformServices()`, `OutputUser()`, `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, users, 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.

KEY POINT

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.



Don't differentiate routine names solely by number One developer wrote all his code in one big function. Then he took every 15 lines and created functions named `Part1`, `Part2`, and so on. After that, he created one high-level function that called each part. This method of creating and naming routines is especially egregious (and rare, I hope). But programmers sometimes use numbers to differentiate routines with names like `OutputUser`, `OutputUser1`, and `OutputUser2`. The numerals at the ends of these names provide no indication of the different abstractions the routines represent, and the routines are thus poorly named.

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. On the other hand, routine names are often attached to object names, which essentially provides part of the name for free. Overall, the emphasis when creating a routine name should be to make the name as clear as possible, which means you should make its name as long or short as needed to make it understandable.

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.

Cross-Reference

For the distinction between procedures and functions, see [Section 7.6, "Special Considerations in the Use of Functions"](#) later in this chapter.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

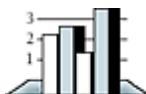
[NEXT ▶](#)

7.4. How Long Can a Routine Be?

On their way to America, the Pilgrims argued about the best maximum length for a routine. After arguing about it for the entire trip, they arrived at Plymouth Rock and started to draft the Mayflower Compact. They still hadn't settled the maximum-length question, and since they couldn't disembark until they'd signed the compact, they gave up and didn't include it. The result has been an interminable debate ever since about how long a routine can be.

The theoretical best maximum length is often described as one screen or one or two pages of program listing, approximately 50 to 150 lines. In this spirit, IBM once limited routines to 50 lines, and TRW limited them to two pages (McCabe 1976). Modern programs tend to have volumes of extremely short routines mixed in with a few longer routines. Long routines are far from extinct, however. Shortly before finishing this book, I visited two client sites within a month. Programmers at one site were wrestling with a routine that was about 4,000 lines of code long, and programmers at the other site were trying to tame a routine that was more than 12,000 lines long!

A mountain of research on routine length has accumulated over the years, some of which is applicable to modern programs, and some of which isn't:



HARD DATA A study by Basili and Perricone found that routine size was inversely correlated with errors: as the size of routines increased (up to 200 lines of code), the number of errors per line of code decreased (Basili and Perricone 1984).

Another study found that routine size was not correlated with errors, even though structural complexity and amount of data were correlated with errors (Shen et al. 1985).

A 1986 study found that small routines (32 lines of code or fewer) were not correlated with lower cost or fault rate (Card, Church, and Agresti 1986; Card and Glass 1990). The evidence suggested that larger routines (65 lines of code or more) were cheaper to develop per line of code.

An empirical study of 450 routines found that small routines (those with fewer than 143 source statements, including comments) had 23 percent more errors per line of code than larger routines but were 2.4 times less expensive to fix than larger routines (Selby and Basili 1991).

Another study found that code needed to be changed least when routines averaged 100 to 150 lines of code (Lind and Vairavan 1989).

A study at IBM found that the most error-prone routines were those that were larger than 500 lines of code. Beyond 500 lines, the error rate tended to be proportional to the size of the routine (Jones 1986a).

Where does all this leave the question of routine length in object-oriented programs? A large percentage of routines in object-oriented programs will be accessor routines, which will be very short. From time to time, a complex algorithm will lead to a longer routine, and in those circumstances, the routine should be allowed to grow organically up to 100–200 lines. (A line is a noncomment, nonblank line of source code.) Decades of evidence say that routines of such length are no more error prone than shorter routines. Let issues such as the routine's cohesion, depth of nesting, number of variables, number of decision points, number of comments needed to explain the routine, and other complexity-related considerations dictate the length of the routine rather than imposing a length restriction per se.

That said, if you want to write routines longer than about 200 lines, be careful. None of the studies that reported

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

7.5. How to Use Routine Parameters



Interfaces between routines are some of the most error-prone areas of a program. 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:

Put parameters in input-modify-output order 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:

Cross-Reference

For details on documenting routine parameters, see "[Commenting Routines](#)" in [Section 32.5](#). For details on formatting parameters, see [Section 31.7](#), "[Laying Out Routines](#)."

Ada Example of Parameters in Input-Modify-Output Order

```
procedure InvertMatrix(
    originalMatrix: in Matrix;           <-- 1
    resultMatrix: out Matrix
);
...
procedure ChangeSentenceCase(
    desiredCase: in StringCase;
    sentence: in out Sentence
);
...
procedure PrintPageNumber(
    pageNumber: in Integer;
    status: out StatusType
);
```

(1)Ada uses in and out keywords to make input and output parameters clear.

This ordering convention conflicts with the C-library convention of putting the modified parameter first. The

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

7.6. Special Considerations in the Use of Functions

Modern languages such as C++, Java, and Visual Basic support both functions and procedures. A function is a routine that returns a value; a procedure is a routine that does not. In C++, all routines are typically called "functions"; however, a function with a void return type is semantically a procedure. The distinction between functions and procedures is as much a semantic distinction as a syntactic one, and semantics should be your guide.

When to Use a Function and When to Use a Procedure

Purists argue that a function should return only one value, just as a mathematical function does. This means that a function would take only input parameters and return its only value through the function itself. The function would always be named for the value it returned, as `sin()`, `CustomerID()`, and `ScreenHeight()` are. A procedure, on the other hand, could take input, modify, and output parameters—as many of each as it wanted to.

A common programming practice is to have a function that operates as a procedure and returns a status value. Logically, it works as a procedure, but because it returns a value, it's officially a function. For example, you might have a routine called `FormatOutput()` used with a report object in statements like this one:

```
if ( report.FormatOutput( formattedReport ) = Success ) then ...
```

In this example, `report.FormatOutput()` operates as a procedure in that it has an output parameter, `formattedReport`, but it is technically a function because the routine itself returns a value. Is this a valid way to use a function? In defense of this approach, you could maintain that the function return value has nothing to do with the main purpose of the routine, formatting output, or with the routine name, `report.FormatOutput()`. In that sense it operates more as a procedure does even if it is technically a function. The use of the return value to indicate the success or failure of the procedure is not confusing if the technique is used consistently.

The alternative is to create a procedure that has a status variable as an explicit parameter, which promotes code like this fragment:

```
report.FormatOutput( formattedReport, outputStatus )  
if ( outputStatus = Success ) then ...
```

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:

```
outputStatus = report.FormatOutput( formattedReport )  
if ( outputStatus = Success ) then ...
```

 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.

KEY POINT

Setting the Function's Return Value

Using a function creates the risk that the function will return an incorrect return value. This usually happens when the function has several possible paths and one of the paths doesn't set a return value. To reduce this risk, do the following:

Check all possible return paths When creating a function, mentally execute each path to be sure that the function returns a value under all possible circumstances. It's good practice to initialize the return value at the beginning of the function to a default value—this provides a safety net in the event that the correct return value is not set.

Don't return references or pointers to local data As soon as the routine ends and the local data goes out of scope, the reference or pointer to the local data will be invalid. If an object needs to return information about its

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

7.7. Macro Routines and Inline Routines

Routines created with preprocessor macros call for a few unique considerations. The following rules and examples pertain to using the preprocessor in C++. If you're using a different language or preprocessor, adapt the rules to your situation.

Cross-Reference

Even if your language doesn't have a macro preprocessor, you can build your own. For details, see [Section 30.5, "Building Your Own Programming Tools."](#)

Fully parenthesize macro expressions Because macros and their arguments are expanded into code, be careful that they expand the way you want them to. One common problem lies in creating a macro like this one:

C++ Example of a Macro That Doesn't Expand Properly

```
#define Cube( a ) a*a*a
```

If you pass this macro 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:

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:

C++ Example of a Macro That Works

```
#define Cube( a ) ((a)*(a)*(a))
```

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:



C++ Example of a Nonworking Macro with Multiple Statements

```
#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 ] );
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- The most important reason for creating a routine is to improve the intellectual manageability of a program, and you can create a routine for many other good reasons. Saving space is a minor reason; improved readability, reliability, and modifiability are better reasons.
- Sometimes the operation that most benefits from being put into a routine of its own is a simple one.
- You can classify routines into various kinds of cohesion, but you can make most routines functionally cohesive, which is best.
- The name of a routine is an indication of its quality. If the name is bad and it's accurate, the routine might be poorly designed. If the name is bad and it's inaccurate, it's not telling you what the program does. Either way, a bad name means that the program needs to be changed.
- Functions should be used only when the primary purpose of the function is to return the specific value described by the function's name.
- Careful programmers use macro routines with care and only as a last resort.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 8. Defensive Programming

cc2e.com/0861

Contents

- [Protecting Your Program from Invalid Inputs page 188](#)
- [Assertions page 189](#)
- [Error-Handling Techniques page 194](#)
- [Exceptions page 198](#)
- [Barricade Your Program to Contain the Damage Caused by Errors page 203](#)
- [Debugging Aids page 205](#)
- [Determining How Much Defensive Programming to Leave in Production Code page 209](#)
- [Being Defensive About Defensive Programming page 210](#)

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](#)
- Design in Construction: [Chapter 5](#)
- Debugging: [Chapter 23](#)



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

KEY POINT 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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

8.1. Protecting Your Program from Invalid Inputs

In school you might have heard the expression, "Garbage in, garbage out." That expression is essentially software development's version of caveat emptor: let the user beware.



KEY POINT

For production software, garbage in, garbage out isn't good enough. A good program never puts out garbage, regardless of what it takes in. A good program uses "garbage in, nothing out," "garbage in, error message out," or "no garbage allowed in" instead. By today's standards, "garbage in, garbage out" is the mark of a sloppy, nonsecure program.

There are three general ways to handle garbage in:

Check the values of all data from external sources When getting data from a file, a user, the network, or some other external interface, check to be sure that the data falls within the allowable range. Make sure that numeric values are within tolerances and that strings are short enough to handle. If a string is intended to represent a restricted range of values (such as a financial transaction ID or something similar), be sure that the string is valid for its intended purpose; otherwise reject it. If you're working on a secure application, be especially leery of data that might attack your system: attempted buffer overflows, injected SQL commands, injected HTML or XML code, integer overflows, data passed to system calls, and so on.

Check the values of all routine input parameters Checking the values of routine input parameters is essentially the same as checking data that comes from an external source, except that the data comes from another routine instead of from an external interface. The discussion in [Section 8.5, "Barricade Your Program to Contain the Damage Caused by Errors,"](#) provides a practical way to determine which routines need to check their inputs.

Decide how to handle bad inputs Once you've detected an invalid parameter, what do you do with it? Depending on the situation, you might choose any of a dozen different approaches, which are described in detail in [Section 8.3, "Error-Handling Techniques,"](#) later in this chapter.

Defensive programming is useful as an adjunct to the other quality-improvement techniques described in this book. The best form of defensive coding is not inserting errors in the first place. Using iterative design, writing pseudocode before code, writing test cases before writing the 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.

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.

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

[\[View full size image\]](#)



[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

8.2. Assertions

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 customerinformation file will never have more than 50,000 records, 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 an error is in the program.



KEY POINT Assertions are especially useful in large, complicated programs and in high-reliability

programs. They enable programmers to more quickly flush out mismatched interface assumptions, errors that creep in when code is modified, and so on.

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:

Java Example of an Assertion

```
assert denominator != 0 : "denominator is unexpectedly equal to 0.;"
```

This assertion asserts that denominator is not equal to 0. The first argument, denominator != 0, is a boolean expression that evaluates to true or false. The second argument is a message to print if the first argument is false—that is, if the assertion is false.

Use assertions to document assumptions made in the code and to flush out unexpected conditions. Assertions can be used to check assumptions like these:

- That an input parameter's value falls within its expected range (or an output parameter's value does)
- That a file or stream is open (or closed) when a routine begins executing (or when it ends executing)
- That a file or stream is at the beginning (or end) when a routine begins executing (or when it ends executing)
- That a file or stream is open for read-only, write-only, or both read and write
- That the value of an input-only variable is not changed by a routine
- That a pointer is non-null
- That an array or other container passed into a routine can contain at least X number of data elements
- That a table has been initialized to contain real values
- That a container is empty (or full) when a routine begins executing (or when it finishes)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

8.3. Error-Handling Techniques

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 shut down—or you might want to use a combination of these responses.

Here are some more details on these options:

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 in a video game might use the default background or foreground color. A drawing routine that displays x-ray data for cancer patients, however, would not want to display a "neutral value." In that case, you'd be better off shutting down the program than displaying incorrect patient data.

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.

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. But if you're authorizing transactions at a cash machine, you probably wouldn't want to use the "same answer as last time"—that would be the previous user's bank account number!

Substitute the closest legal value In some cases, you might choose to return the closest legal value, as in the Velocity example earlier. 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.

Log a warning message to a file When bad data is detected, you might choose to log a warning message to a file and then continue on. This approach can be used in conjunction with other techniques like substituting the closest legal value or substituting the next piece of valid data. If you use a log, consider whether you can safely make it publicly available or whether you need to encrypt it or protect it some other way.

Return an error code You could decide that only certain parts of a system will handle errors. Other parts will not handle errors locally; they will simply report that an error has been detected and trust that some other routine higher up in the calling hierarchy will handle the error. The specific mechanism for notifying the rest of the system that an error has occurred could be any of the following:

- Set the value of a status variable
- Return status as the function's return value
- Throw an exception by using the language's built-in exception mechanism

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)



8.4. Exceptions

Exceptions are a specific means by which code can pass along errors or exceptional events to the code that called it. If code in one routine encounters an unexpected condition that it doesn't know how to handle, it throws an exception, essentially throwing up its hands and yelling, "I don't know what to do about this—I sure hope somebody else knows how to handle it!" Code that has no sense of the context of an error can return control to other parts of the system that might have a better ability to interpret the error and do something useful about it.

Exceptions can also be used to straighten out tangled logic within a single stretch of code, such as the "Rewrite with try-finally" example in [Section 17.3](#). The basic structure of an exception is that a routine uses throw to throw an exception object. Code in some other routine up the calling hierarchy will catch the exception within a try-catch block.

Popular languages vary in how they implement exceptions. [Table 8-1](#) summarizes the major differences in three of them:

Table 8-1. Popular-Language Support for Exceptions

Exception Attribute	C++	Java	Visual Basic
Try-catch support	yes	yes	yes
Try-catch-finally support	no	yes	yes
What can be thrown	Exception object or object derived from Exception class; object pointer; object reference; data type like string or int	Exception object or object derived from Exception class	Exception object or object derived from Exception class
Effect of uncaught exception	Invokes std::unexpected(), which by default invokes std::terminate(), which by default invokes abort()	Terminates thread of execution if exception is a "checked exception"; no effect if exception is a "runtime exception"	Terminates program
Exceptions thrown must be defined in class interface	No	Yes	No
Exceptions caught must be defined in class interface	No	Yes	No

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.

Programs that use exceptions as part of their normal processing suffer from all the readability and maintainability problems of classic spaghetti code.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

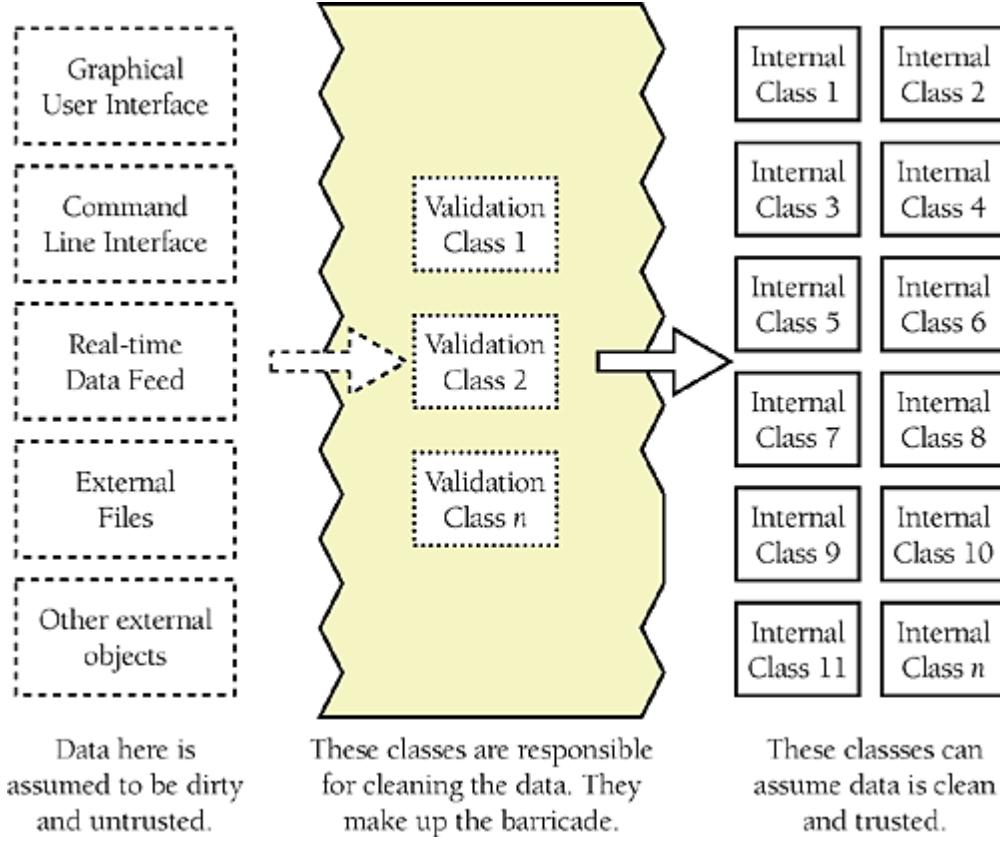
8.5. Barricade Your Program to Contain the Damage Caused by Errors

Barricades are a damage-containment strategy. The reason is similar to that for having isolated compartments in the hull of a ship. If the ship runs into an iceberg and pops open the hull, that compartment is shut off and the rest of the ship isn't affected. They are also similar to firewalls in a building. A building's firewalls prevent fire from spreading from one part of a building to another part. (Barricades used to be called "firewalls," but the term "firewall" now commonly refers to blocking hostile network traffic.)

One way to barricade for defensive programming purposes is to designate certain interfaces as boundaries to "safe" areas. Check data crossing the boundaries of a safe area for validity, and respond sensibly if the data isn't valid. [Figure 8-2](#) illustrates this concept.

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

[\[View full size image\]](#)



This same approach can be used at the class level. The class's public methods assume the data is unsafe, and they are responsible for checking the data and sanitizing it. Once the data has been accepted by the class's public methods, the class's private methods can assume the data is safe.

Another way of thinking about this approach is as an operating-room technique. Data is sterilized before it's allowed to enter the operating room. Anything that's in the operating room is assumed to be safe. The key design decision is deciding what to put in the operating room, what to keep out, and where to put the doors—which routines are considered to be inside the safety zone, which are outside, and which sanitize the data. The easiest way to do this is usually by sanitizing external data as it arrives, but data often needs to be sanitized at more than one level, so multiple levels of sterilization are sometimes required.

Convert input data to the proper type at input time Input typically arrives in the form of a string or number.

Sometimes the code will want to check for a file, URL, or other "Something". In this case, the code must convert

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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.

Don't Automatically Apply Production Constraints to the Development Version

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.

Further Reading

For more on using debug code to support defensive programming, see *Writing Solid Code* (Maguire 1993).

One program I worked on made extensive use of a quadruply linked list. The linked-list code was error prone, and the linked list tended to get corrupted. I added a menu option to check the integrity of the linked list.

In debug mode, Microsoft Word contains code in the idle loop that checks the integrity of the Document object every few seconds. This helps to detect data corruption quickly, and it makes for easier error diagnosis.



Be willing to trade speed and resource usage during development in exchange for built-in tools that can make development go more smoothly.

KEY POINT

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).

Cross-Reference

For more details on handling unanticipated cases, see "[Tips for Using case Statements](#)" in [Section 15.2](#).

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:

A dead program normally does a lot less damage than a crippled one.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

8.7. Determining How Much Defensive Programming to Leave in Production Code

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:

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 such 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.

Remove code that checks for trivial errors If an error has truly trivial consequences, remove code that checks for it. In the previous example, you might remove the code that checks the spreadsheet screen update. "Remove" doesn't mean physically remove the code. It means use version control, precompiler switches, or some other technique to compile the program without that particular code. If space isn't a problem, you could leave in the error-checking code but have it log messages to an error-log file unobtrusively.

Remove code that results in hard crashes As I mentioned, during development, when your program detects an error, you'd like the error to be as noticeable as possible so that you can fix it. Often, the best way to accomplish that goal is to have the program print a debugging message and crash when it detects an error. This is useful even for minor errors.

During production, your users need a chance to save their work before the program crashes and they are probably willing to tolerate a few anomalies in exchange for keeping the program going long enough for them to do that. Users don't appreciate anything that results in the loss of their work, regardless of how much it helps debugging and ultimately improves the quality of the program. If your program contains debugging code that could cause a loss of data, take it out of the production version.

Leave in code that helps the program crash gracefully If your program contains debugging code that detects potentially fatal errors, leave the code in that allows the program to crash gracefully. In the Mars Pathfinder, for example, engineers left some of the debug code in by design. An error occurred after the Pathfinder had landed. By using the debug aids that had been left in, engineers at JPL were able to diagnose the problem and upload revised code to the Pathfinder, and the Pathfinder completed its mission perfectly (March 1999).

Log errors for your technical support personnel Consider leaving debugging aids in the production code but changing their behavior so that it's appropriate for the production version. If you've loaded your code with assertions that halt the program during development, you might consider changing the assertion routine to log messages to a file during production rather than eliminating them altogether.

Make sure that the error messages you leave in are friendly If you leave internal error messages in the program, verify that they're in language that's friendly to the user. In one of my early programs, I got a call from a user who reported that she'd gotten a message that read "You've got a bad pointer allocation, Dog Breath!" Fortunately for me, she had a sense of humor. A common and effective approach is to notify the user of an "internal error" and list an e-mail address or phone number the user can use to report it.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

8.8. Being Defensive About Defensive Programming

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.

Too much of anything is bad, but too much whiskey is just enough.

—Mark Twain

cc2e.com/0868

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?
- 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?

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/0875

Take a look at the following defensive-programming resources:

Security

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, MA: Addison-Wesley, 1997. [Section 24.3](#) .7.2 describes several variations on the theme of implementing assertions in C++, including the relationship between assertions and preconditions and postconditions.

Meyer, Bertrand. *Object-Oriented Software Construction*, 2d ed. New York, NY: Prentice Hall PTR, 1997. This book contains the definitive discussion of preconditions and postconditions.

Exceptions

Meyer, Bertrand. *Object-Oriented Software Construction*, 2d ed. New York, NY: Prentice Hall PTR, 1997. [Chapter 12](#) contains a detailed discussion of exception handling.

Stroustrup, Bjarne. *The C++ Programming Language*, 3d ed. Reading, MA: Addison-Wesley, 1997. [Chapter 14](#) contains a detailed discussion of exception handling in C++. Section 14.11 contains an excellent summary of 21 tips for handling C++ exceptions.

Meyers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Reading, MA: Addison-Wesley, 1996. Items 9–15 describe numerous nuances of exception handling in C++.

Arnold, Ken, James Gosling, and David Holmes. *The Java Programming Language*, 3d ed. Boston, MA: Addison-Wesley, 2000. [Chapter 8](#) contains a discussion of exception handling in Java.

Bloch, Joshua. *Effective Java Programming Language Guide*. Boston, MA: Addison-Wesley, 2001. Items 39–47 describe nuances of exception handling in Java.

Foxall, James. *Practical Standards for Microsoft Visual Basic .NET*. Redmond, WA: Microsoft Press, 2003. [Chapter 10](#) describes exception handling in Visual Basic.

Key Points

- Production code should handle errors in a more sophisticated way than "garbage in, garbage out."
- Defensive-programming techniques make errors easier to find, easier to fix, and less damaging to production code.
- Assertions can help detect errors early, especially in large systems, high-reliability systems, and fast-changing code bases.
- The decision about how to handle bad inputs is a key error-handling decision and a key high-level design decision.
- Exceptions provide a means of handling errors that operates in a different dimension from the normal flow of the code. They are a valuable addition to the programmer's intellectual toolbox when used with care, and they should be weighed against other error-processing techniques.
- Constraints that apply to the production system do not necessarily apply to the development version. You can use that to your advantage, adding code to the development version that helps to flush out errors quickly.

Chapter 9. The Pseudocode Programming Process

cc2e.com/0936

Contents

- [Summary of Steps in Building Classes and Routines page 216](#)
- [Pseudocode for Pros page 218](#)
- [Constructing Routines by Using the PPP page 220](#)
- [Alternatives to the PPP page 232](#)

Related Topics

- Creating high-quality classes: [Chapter 6](#)
- Characteristics of high-quality routines: [Chapter 7](#)
- Design in Construction: [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, the steps 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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

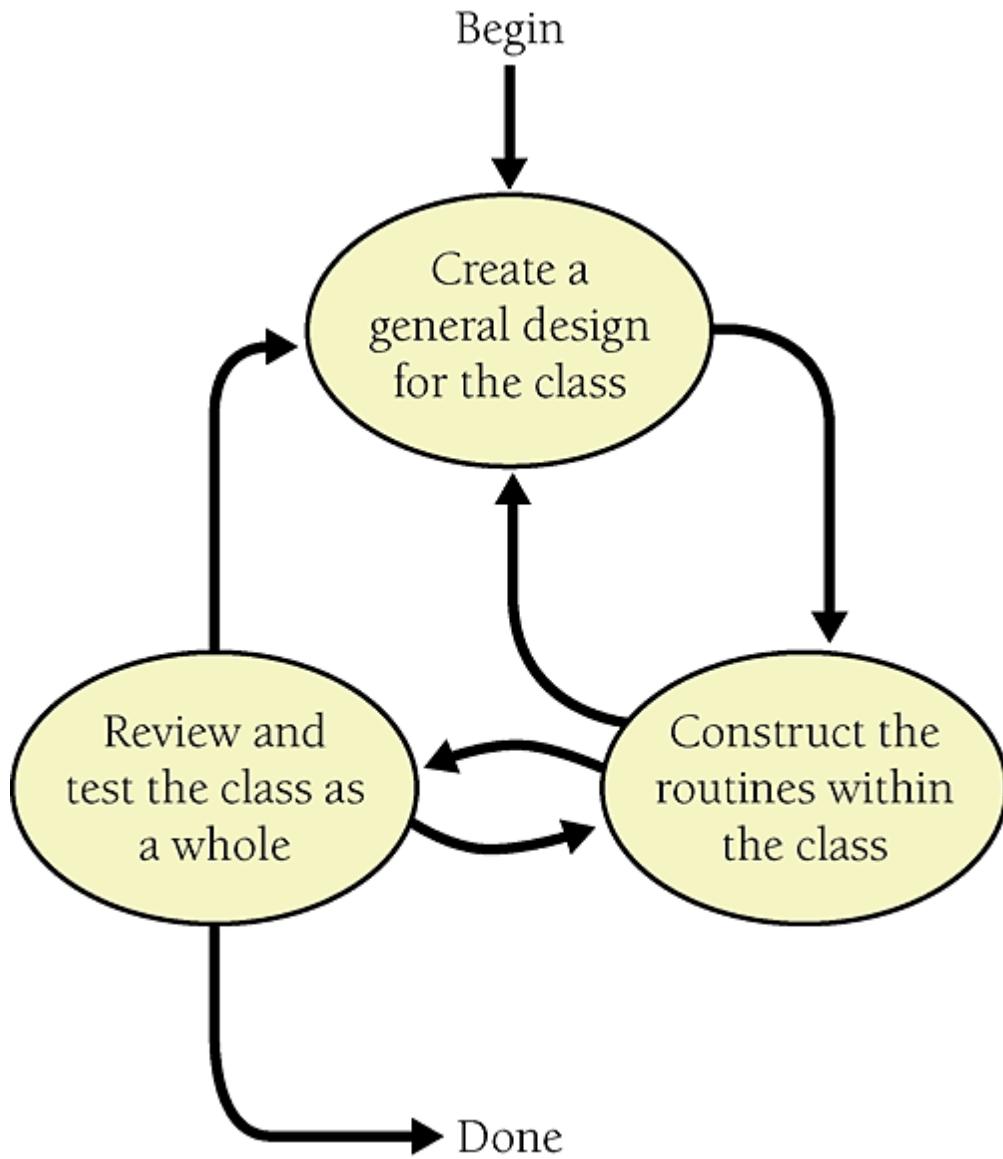
[NEXT ▶](#)

9.1. Summary of Steps in Building Classes and Routines

Class construction can be approached from numerous directions, but usually it's an iterative process of creating a general design for the class, enumerating specific routines within the class, constructing specific routines, and checking class construction as a whole. As [Figure 9-1](#) suggests, class creation can be a messy process for all the reasons that design is a messy process (reasons that are described in [Section 5.1, "Design Challenges"](#)).

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

[\[View full size image\]](#)



Steps in Creating a Class

The key steps in constructing a class are:

Create a general design for the class Class design includes numerous specific issues. Define the class's specific responsibilities, define what "secrets" the class will hide, and define exactly what abstraction the class interface will capture. Determine whether the class will be derived from another class and whether other classes will be allowed to derive from it. Identify the class's key public methods, and identify and design any nontrivial data members used by the class. Iterate through these topics as many times as needed to create a straightforward design for the routine. These considerations and many others are discussed in more detail in [Chapter 6, "Working Classes."](#)

Construct each routine within the class Once you've identified the class's major routines in the first step, you must

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

9.2. Pseudocode for Pros

The term "pseudocode" refers to an informal, English-like notation for describing how an algorithm, a routine, a class, or a program will work. The Pseudocode Programming Process defines a specific approach to using pseudocode to streamline the creation of code within routines.

Because pseudocode resembles English, it's natural to assume that any English-like description that collects your thoughts will have roughly the same effect as any other. In practice, you'll find that some styles of pseudocode are more useful than others. Here are guidelines for using pseudocode effectively:

- Use English-like statements that precisely describe specific operations.

- Avoid syntactic elements from the target programming language. Pseudocode allows you to design at a slightly higher level than the code itself. When you use programming-language constructs, you sink to a lower level, eliminating the main benefit of design at a higher level, and you saddle yourself with unnecessary syntactic restrictions.

- Write pseudocode at the level of intent. Describe the meaning of the approach rather than how the approach will be implemented in the target language.

Cross-Reference

For details on commenting at the level of intent, see "[Kinds of Comments](#)" in [Section 32.4](#).

- Write pseudocode at a low enough level that generating code from it will be nearly automatic. If the pseudocode is at too high a level, it can gloss over problematic details in the code. Refine the pseudocode in more and more detail until it seems as if it would be easier to simply write the code.

Once the pseudocode is written, you build the code around it and the pseudocode turns into programming-language comments. This eliminates most commenting effort. If the pseudocode follows the guidelines, the comments will be complete and meaningful.

Here's an example of a design in pseudocode that violates virtually all the principles just described:



Example of Bad Pseudocode

```
increment resource number by 1

allocate a dlg struct using malloc

if malloc() returns NULL then return 1

invoke OSrsrc_init to initialize a resource for the operating
system

*hRsrcPtr = resource number

return 0
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

9.3. Constructing Routines by Using the PPP

This section describes the activities involved in constructing a routine, namely these:

- Design the routine.
- Code the routine.
- Check the code.
- Clean up loose ends.
- 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()`:

Cross-Reference

For details on other aspects of design, see [Chapters 5 through 8](#).

`ReportErrorMessage()` 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.

Cross-Reference

For details on checking prerequisites, see [Chapter 3, "Measure Twice, Cut Once: Upstream Prerequisites,"](#) and [Chapter 4, "Key Construction Decisions."](#)

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

9.4. Alternatives to the PPP

For my money, the PPP is the best method for creating classes and routines. Here are some different approaches recommended by other experts. You can use these approaches as alternatives or as supplements to the PPP.

Test-first development Test-first is a popular development style in which test cases are written prior to writing any code. This approach is described in more detail in "[Test First or Test Last?](#)" in [Section 22.2](#). A good book on test-first programming is Kent Beck's *Test-Driven Development: By Example* (Beck 2003).

Refactoring Refactoring is a development approach in which you improve code through a series of semantic preserving transformations. Programmers use patterns of bad code or "smells" to identify sections of code that need to be improved. [Chapter 24, "Refactoring,"](#) describes this approach in detail, and a good book on the topic is Martin Fowler's *Refactoring: Improving the Design of Existing Code* (Fowler 1999).

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 and verify 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 found 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.

[cc2e.com/0943](#)

Cross-Reference

The point of this list is to check whether you followed a good set of steps to create a routine. For a checklist that focuses on the quality of the routine itself, see the "[High-Quality Routines](#)" checklist in [Chapter 7](#), page 185.

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 mainly in terms of meeting resource and speed budgets?

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- Constructing classes and constructing routines tends to be an iterative process. Insights gained while constructing specific routines tend to ripple back through the class's design.
- Writing good pseudocode calls for using understandable English, avoiding features specific to a single programming language, and writing at the level of intent (describing what the design does rather than how it will do it).
- The Pseudocode Programming Process is a useful tool for detailed design and makes coding easy. Pseudocode translates directly into comments, ensuring that the comments are accurate and useful.
- Don't settle for the first design you think of. Iterate through multiple approaches in pseudocode and pick the best approach before you begin writing code.
- Check your work at each step, and encourage others to check it too. That way, you'll catch mistakes at the least expensive level, when you've invested the least amount of effort.

Part III: Variables

[In this part:](#)

[Chapter 10. General Issues in Using Variables](#)

[Chapter 11. The Power of Variable Names](#)

[Chapter 12. Fundamental Data Types](#)

[Chapter 13. Unusual Data Types](#)

In this part:

[Chapter 10: General Issues in Using Variables](#)

[Chapter 11: The Power of Variable Names](#)

[Chapter 12: Fundamental Data Types](#)

[Chapter 13: Unusual Data Types](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 10. General Issues in Using Variables

cc2e.com/1085

Contents

- [Data Literacy page 238](#)
- [Making Variable Declarations Easy page 239](#)
- [Guidelines for Initializing Variables page 240](#)
- [Scope page 244](#)
- [Persistence page 251](#)
- [Binding Time page 252](#)
- [Relationship Between Data Types and Control Structures page 254](#)
- [Using Each Variable for Exactly One Purpose page 255](#)

Related Topics

- Naming variables: [Chapter 11](#)
- Fundamental data types: [Chapter 12](#)
- Unusual data types: [Chapter 13](#)
- Formatting data declarations: "[Laying Out Data Declarations](#)" in [Section 31.5](#)
- Documenting variables: "[Commenting Data Declarations](#)" in [Section 32.5](#)

It's normal and desirable for construction to fill in small gaps in the requirements and architecture. It would be inefficient to draw blueprints to such a microscopic level that every detail was completely specified. This chapter describes a nuts-and-bolts construction issue: the ins and outs of using variables.

The information in this chapter should be particularly valuable to you if you're an experienced programmer. It's easy to start using hazardous practices before you're fully aware of your alternatives and then to continue to use them out of habit even after you've learned ways to avoid them. An experienced programmer might find the discussions on binding time in [Section 10.6](#) and on using each variable for one purpose in [Section 10.8](#) particularly interesting. If you're not

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

10.1. Data Literacy



KEY POINT

The first step in creating effective data is knowing which kind of data to create. A good repertoire of data types is a key part of a programmer's toolbox. A tutorial in data types is beyond the scope of this book, but the "Data Literacy Test" will help you determine how much more you might need to learn about them.

The Data Literacy Test

Put a 1 next to each term that looks familiar. If you think you know what a term means but aren't sure, give yourself a 0.5. Add the points when you're done, and interpret your score according to the scoring table below.

abstract data type

literal

array

local variable

bitmap

lookup table

boolean variable

member data

B-tree

pointer

character variable

private

container class

retroactive synapse

double precision

referential integrity

elongated stream

stack

enumerated type

string

floating point

structured variable

heap

tree

index

typedef

integer

union

linked list

value chain

named constant

variant

Total Score

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 might 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.

Cross-Reference

For details on layout of variable declarations, see "[Laying Out Data Declarations](#)" in [Section 31.5](#). For details on documenting them, see "[Commenting Data Declarations](#)" in [Section 32.5](#).

Implicit Declarations

Some languages have implicit variable declarations. For example, if you use a variable in Microsoft 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.



KEY POINT 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 it virtually eliminates the synonymous-variables problem. Languages that require you to declare data explicitly are, in essence, requiring 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 Option Explicit and No so that you don't use two variables when you mean to use one.

Cross-Reference

For details on the standardization of abbreviations, see "[General Abbreviation Guidelines](#)" in [Section 11.6](#).

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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

10.3. Guidelines for Initializing Variables



KEY POINT

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.

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 several reasons:

- 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.
- Cross-Reference
For a testing approach based on data initialization and use patterns, see "[Data-Flow Testing](#)" in [Section 22.3](#).
- 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.

Following are guidelines for avoiding initialization problems:

Initialize each variable as it's declared Initializing variables as they're declared is an inexpensive form of defensive programming. It's a good insurance policy against initialization errors. The example below ensures that `studentGrades` will be reinitialized each time you call the routine that contains it.

C++ Example of Initialization at Declaration Time

```
float studentGrades[ MAX_STUDENTS ] = { 0.0 };
```

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 following one, in which declarations are grouped together and then initializations are grouped together—all far from the first actual use of the variables.

Cross-Reference

Checking input parameters is a form of defensive programming. For details on defensive programming, see [Chapter 8](#), "[Defensive Programming](#)".



Visual Basic Example of Bad

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

10.4. Scope

"[Scope](#)" is a way of thinking about a variable's celebrity status: how famous is it? Scope, or visibility, refers to the extent to which your variables are known and can be referenced throughout a program. A variable with limited or small scope is known in only a small area of a program—a loop index used in only one small loop, for instance. A variable with large scope is known in many places in a program—a table of employee information that's used throughout a program, for instance.

Different languages handle scope in different ways. In some primitive languages, all variables are global. You therefore don't have any control over the scope of a variable, and that can create a lot of problems. In C++ and similar languages, a variable can be visible to a block (a section of code enclosed in curly brackets), a routine, a class (and possibly its derived classes), or the whole program. In Java and C#, a variable can also be visible to a package or namespace (a collection of classes).

The following sections provide guidelines that apply to scope.

Localize References to Variables

The code between references to a variable is a "window of vulnerability." In the window, new code might be added, inadvertently altering the variable, or someone reading the code might forget the value the variable is supposed to contain. It's always a good idea to localize references to variables by keeping them close together.

The idea of localizing references to a variable is pretty self-evident, but it's an idea that lends itself to formal measurement. One method of measuring how close together the references to a variable are is to compute the "span" of a variable. Here's an example:

Java Example of Variable Span

```
a = 0;  
  
b = 0;  
  
c = 0;  
  
a = b + c;
```

In this case, two lines come between the first reference to a and the second, so a has a span of two. One line comes between the two references to b, so b has a span of one, and c has a span of zero. Here's another example:

Java Example of Spans of One and Zero

```
a = 0;  
  
b = 0;  
  
c = 0;  
  
b = a + 1;  
  
b = b / c;
```

In this case, there is one line between the first reference to b and the second, for a span of one. There are no lines between the second reference to b and the third, for a span of zero.

Further Reading

For more information on variable span, see Software Engineering Metrics and Models (Conte, Dunsmore, and Shen 1986).

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

10.5. Persistence

"[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.

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.

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.
- Cross-Reference
- Debug code is easy to include in access routines and is discussed more in "[Advantages of Access Routines](#)" in [Section 13.3](#).
- Set variables to "unreasonable values" when you're through with them. For example, you could set a pointer to null after you delete it.
- 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!

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

10.6. Binding Time

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?

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:

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 sync with 0xFFs used elsewhere in the code that must be the same value as this one.

Here's an example of binding at a slightly later time, when the code is compiled:

Java Example of a Variable That's Bound at Compile Time

```
private static final int COLOR_BLUE = 0xFF;  
  
private static final int TITLE_BAR_COLOR = COLOR_BLUE;  
  
...  
  
titleBar.color = TITLE_BAR_COLOR;
```

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.

Here's an example of binding later, at run time:

Java Example of a Variable That's Bound at Run Time

```
titleBar.color = ReadTitleBarColor();
```

ReadTitleBarColor() is a routine that reads a value while a program is executing, perhaps from the Microsoft Windows registry file or a Java properties file.

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.

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 represents successively later binding times.

To summarize, following are the times a variable can be bound to a value in this example. (The details could vary somewhat in other cases.)

-

Coding time (use of magic numbers)

-

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

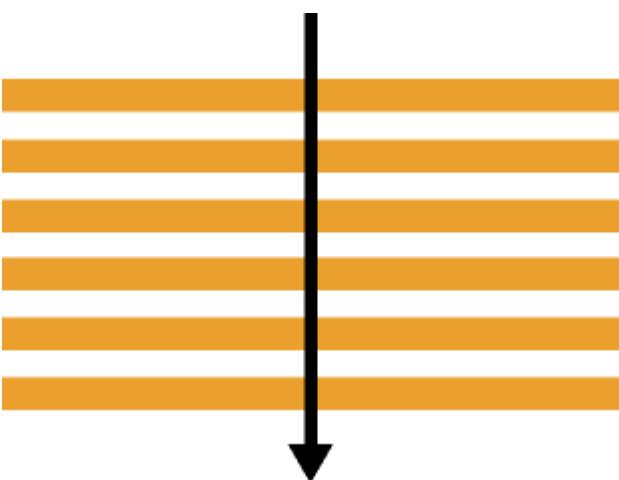
10.7. Relationship Between Data Types and Control Structures

Data types and control structures relate to each other in well-defined ways that were originally described by the British computer scientist Michael Jackson (Jackson 1975). This section sketches the regular relationship between data and control flow.

Jackson draws connections between three types of data and corresponding control structures:

Sequential data translates to sequential statements in a program Sequences consist of clusters of data used together in a certain order, as suggested by [Figure 10-2](#). 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.

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

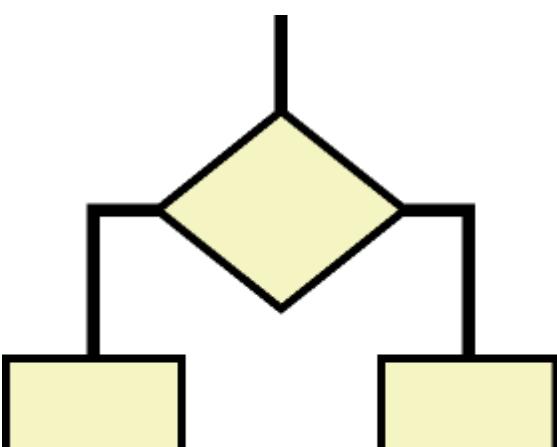


Cross-Reference

For details on conditionals, see [Chapter 15, "Using Conditionals."](#)

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 used at any particular time, but only one, as shown in [Figure 10-3](#). The corresponding program statements must do the actual selection, and they consist 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.

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



[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)



10.8. Using Each Variable for Exactly One Purpose

KEY POINT

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 temps 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:

Cross-Reference

Routine parameters should also be used for one purpose only. For details on using routine parameters, see [Section 7.5, "How to Use Routine Parameters."](#)

C++ Example of Using Two Variables for Two Purposes---Good Practice

```
// Compute roots of a quadratic equation.
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- Data initialization is prone to errors, so use the initialization techniques described in this chapter to avoid the problems caused by unexpected initial values.
- Minimize the scope of each variable. Keep references to a variable close together. Keep it local to a routine or class. Avoid global data.
- Keep statements that work with the same variables as close together as possible.
- Early binding tends to limit flexibility but minimize complexity. Late binding tends to increase flexibility but at the price of increased complexity.
- Use each variable for one and only one purpose.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 11. The Power of Variable Names

cc2e.com/1184

Contents

- [Considerations in Choosing Good Names page 259](#)
- [Naming Specific Types of Data page 264](#)
- [The Power of Naming Conventions page 270](#)
- [Informal Naming Conventions page 272](#)
- [Standardized Prefixes page 279](#)
- [Creating Short Names That Are Readable page 282](#)
- [Kinds of Names to Avoid page 285](#)

Related Topics

- Routine names: [Section 7.3](#)
- Class names: [Section 6.2](#)
- General issues in using variables: [Chapter 10](#)
- Formatting data declarations: "[Laying Out Data Declarations](#)" in [Section 31.5](#)
- Documenting variables: "[Commenting Data Declarations](#)" in [Section 32.5](#)

As important as the topic of good names is to effective programming, I have never read a discussion that covered more than a handful of the dozens of considerations that go into creating good names. Many programming texts devote a few paragraphs to choosing abbreviations, spout a few platitudes, and expect you to fend for yourself. I intend to be guilty of the opposite: to inundate you with more information about good names than you will ever be able to use!

This chapter's guidelines apply primarily to naming variables—objects and primitive data. But they also apply to naming classes, packages, files, and other programming entities. For details on naming routines, see [Section 7.3, "Good Routine Names."](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)



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:



Java Example of Poor Variable Names

```
x = x - xx;  
  
xxx = fido + SalesTax( fido );  
  
x = x + LateFee( x1, x ) + xxxx;  
  
x = x + Interest( x1, x );
```

What's happening in this piece of code? What do x1, xx, and xxx mean? What does fido 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 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



KEY POINT

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 description of the entity, it won't be confused with something else. And it's easy to remember because the name is similar to the concept.

For a variable that represents the number of people on the U.S. Olympic team, you would create the name `numberOfPeopleOnTheUsOlympicTeam`. A variable that represents the number of seats in a stadium would be

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

11.2. Naming Specific Types of Data

In addition to the general considerations in naming data, special considerations come up in the naming of specific kinds of data. This section describes considerations specifically for loop variables, status variables, temporary variables, boolean variables, enumerated types, and named constants.

Naming Loop Indexes

Cross-Reference

For details on loops, see [Chapter 16, "Controlling Loops."](#)

Guidelines for naming variables in loops have arisen because loops are such a common feature of computer programming. The names i, j, and k are customary:

Java Example of a Simple Loop Variable Name

```
for ( i = firstItem; i < lastItem; i++ ) {  
    data[ i ] = 0;  
}
```

If a variable is to be used outside the loop, it should be given a name more meaningful than i, j, or k. For example, if you are reading records from a file and need to remember how many records you've read, a name like recordCount would be appropriate:

Java Example of a Good Descriptive Loop Variable Name

```
recordCount = 0;  
  
while ( moreScores() ) {  
  
    score[ recordCount ] = GetNextScore();  
  
    recordCount++;  
  
}  
  
  
// lines using recordCount  
  
...
```

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.

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.

Java Example of Good Loop Names in a Nested Loop

```
for ( teamIndex = 0; teamIndex < teamCount; teamIndex++ ) {  
  
    for ( eventIndex = 0; eventIndex < eventCount[ teamIndex ]; eventIndex++ ) {  
  
        score[ teamIndex ][ eventIndex ] = 0;  
  
    }  
}
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 support objects, you can supplement it with a naming convention. Names like address, phone, and name don't indicate that the variables are related. But suppose you decide that all employee-data variables should begin with an Employee prefix. employeeAddress, employeePhone, and employeeName leave no doubt that the variables are related. Programming conventions can make up for the weakness of the language you're using.



KEY POINT The key is that any convention at all is often better than no convention. The convention may be arbitrary. The power of naming conventions doesn't come 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.

When You Should Have a Naming Convention

There are no hard-and-fast rules for when you should establish a naming convention, but here are a few cases in which conventions are worthwhile:

- When multiple programmers are working on a project

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 The convention this book uses 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. Several standard options exist, 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;
```

Each of these options has strengths and weaknesses. 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.

The Option 1 approach can't be applied consistently in mixed-language environments if any of the languages are case-insensitive. In Microsoft 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 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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

11.5. Standardized Prefixes

Further Reading

For further details on the Hungarian naming convention, see "The Hungarian Revolution" (Simonyi and Heller 1991).

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 type (UDT) abbreviation and the semantic prefix.

User-Defined Type Abbreviations

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. [Table 11-6](#) offers 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
<code>ch</code>	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)
<code>doc</code>	Document
<code>pa</code>	Paragraph
<code>scr</code>	Screen region
<code>sel</code>	Selection
<code>wn</code>	Window

When you use UDTs, you also define programming-language data types that use the same abbreviations as the UDTs. Thus, if you had the UDTs in [Table 11-6](#), you'd see data declarations like these:

```
CH      chCursorPosition;
```

```
SCR     scrUserWorkspace;
```

```
DOC     docActive
```

```
PA      firstPaActiveDocument;
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

11.6. Creating Short Names That Are Readable



KEY POINT

The desire to use short variable names is in some ways a remnant of an earlier age of computing. Older languages like assembler, generic Basic, and Fortran limited variable names to 2–8 characters and forced programmers to create short names. Early computing was more closely linked to mathematics and its 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 can create names of virtually any length; you have almost no reason to shorten meaningful names.

If circumstances do require you to create short names, note that some methods of shortening names are better than others. You can create good short variable names by eliminating needless words, using short synonyms, and using any of several abbreviation strategies. It's a good idea to be familiar with multiple techniques for abbreviating because no single technique works well in all cases.

General Abbreviation Guidelines

Here are several guidelines for creating abbreviations. Some of them contradict others, so don't try to use them all at the same time.

- Use standard abbreviations (the ones in common use, which are listed in a dictionary).
- Remove all nonleading vowels. (computer becomes cmpr, and screen becomes scrn, apple becomes appl, and integer becomes intgr.)
- Remove articles: and, or, the, and so on.
- Use the first letter or first few letters of each word.
- Truncate consistently after the first, second, or third (whichever is appropriate) letter of each word.
- Keep the first and last letters of each word.
- Use every significant word in the name, up to a maximum of three words.
- Remove useless suffixes—ing, ed, and so on.
- Keep the most noticeable sound in each syllable.
- Be sure not to change the meaning of the variable.
- Iterate through these techniques until you abbreviate each variable name to between 8 to 20 characters or the number of characters to which your language limits variable names.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

11.7. Kinds of Names to Avoid

Here are some guidelines regarding variable names to avoid:

Avoid misleading names or abbreviations Make sure that a name is unambiguous. For example, FALSE is usually the opposite of TRUE and would be a bad abbreviation for "Fig and Almond Season."

Avoid names with similar meanings If you can switch the names of two variables without hurting the program, you need to rename both variables. For example, input and inputValue, recordNum and numRecords, and fileNumber and fileIndex are so semantically similar that if you use them in the same piece of code you'll easily confuse them and install some subtle, hard-to-find errors.

Avoid variables with different meanings but similar names If you have two variables with similar names and different meanings, try to rename one of them or change your abbreviations. Avoid names like clientRecs and clientReps. They're only one letter different from each other, and the letter is hard to notice. Have at least two-letter differences between names, or put the differences at the beginning or at the end. clientRecords and clientReports are better than the original names.

Cross-Reference

The technical term for differences like this between similar variable names is "psychological distance." For details, see "[How "Psychological Distance" Can Help](#)" in [Section 23.4](#).

Avoid names that sound similar, such as wrap and rap Homonyms get in the way when you try to discuss your code with others. One of my pet peeves about Extreme Programming (Beck 2000) is its overly clever use of the terms Goal Donor and Gold Owner, which are virtually indistinguishable when spoken. You end up having conversations like this:

I was just speaking with the Goal Donor—

Did you say "Gold Owner" or "Goal Donor"?

I said "Goal Donor."

What?

GOAL - - - DONOR!

OK, Goal Donor. You don't have to yell, Goll' Darn it.

Did you say "Gold Donut?"

Remember that the telephone test applies to similar sounding names just as it does to oddly abbreviated names.

Avoid numerals in names If the numerals in a name are really significant, use an array instead of separate variables. If an array is inappropriate, numerals are even more inappropriate. For example, avoid file1 and file2, or totall and total2. You can almost always think of a better way to differentiate between two variables than by tacking a 1 or a 2

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- Good variable names are a key element of program readability. Specific kinds of variables such as loop indexes and status variables require specific considerations.
- Names should be as specific as possible. Names that are vague enough or general enough to be used for more than one purpose are usually bad names.
- Naming conventions distinguish among local, class, and global data. They distinguish among type names, named constants, enumerated types, and variables.
- Regardless of the kind of project you're working on, you should adopt a variable naming convention. The kind of convention you adopt depends on the size of your program and the number of people working on it.
- Abbreviations are rarely needed with modern programming languages. If you do use abbreviations, keep track of abbreviations in a project dictionary or use the standardized prefixes approach.
- Code is read far more times than it is written. Be sure that the names you choose favor read-time convenience over write-time convenience.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 12. Fundamental Data Types

cc2e.com/1278

Contents

- [Numbers in General page 292](#)
- [Integers page 293](#)
- [Floating-Point Numbers page 295](#)
- [Characters and Strings page 297](#)
- [Boolean Variables page 301](#)
- [Enumerated Types page 303](#)
- [Named Constants page 307](#)
- [Arrays page 310](#)
- [Creating Your Own Types \(Type Aliasing\) page 311](#)

Related Topics

- Naming data: [Chapter 11](#)
- Unusual data types: [Chapter 13](#)
- General issues in using variables: [Chapter 10](#)
- Formatting data declarations: "[Laying Out Data Declarations](#)" in [Section 31.5](#)
- Documenting variables: "[Commenting Data Declarations](#)" in [Section 32.5](#)
- Creating classes: [Chapter 6](#)

The fundamental data types are the basic building blocks for all other data types. This chapter contains tips for using numbers (in general), integers, floating-point numbers, characters and strings, boolean variables, enumerated types,

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

12.1. Numbers in General

Here are several guidelines for making your use of numbers less error-prone:

Cross-Reference

For more details on using named constants instead of magic numbers, see [Section 12.7, "Named Constants,"](#) later in this chapter.

Avoid "magic numbers" Magic numbers are literal numbers, such as 100 or 47524, that appear in the middle of a program without explanation. If you program in a language that supports named constants, use them instead. If you can't use named constants, use global variables when it's feasible to do so.

Avoiding magic numbers yields three advantages:

- Changes can be made more reliably. If you use named constants, you won't over-look one of the 100s 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 100s and change them to 200s. If you use 100+1 or 100-1, you'll also have to find all the 101s and 99s and change them to 201s and 199s. 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.

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.

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

```
y = x + (float) i
```

and in Microsoft Visual Basic you could say

```
y = x + CSng( i )
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

12.2. Integers

Bear these considerations in mind when using integers:

Check for integer division When you're using integers, $7/10$ does not equal 0.7 . It usually equals 0 , or minus infinity, or the nearest integer, or—you get the picture. What it equals varies from language to language. 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 integer division $(7/10)$ equals 0 . The easiest way to remedy this problem is to reorder the expression so that the divisions are done last: $(10*7) / 10$.

Check for integer overflow When doing integer multiplication or addition, you need to be aware of the largest possible integer. The largest possible unsigned integer is often $2^{32}-1$ and is sometimes $2^{16}-1$, or $65,535$. The problem comes up when you multiply two numbers that produce a 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 answer you'll get is probably 9464 because of integer overflow ($75,000 - 65,536 = 9464$). [Table 12-1](#) shows the ranges of common integer types.

Table 12-1. Ranges for Different Types of Integers

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

The easiest way to prevent integer overflow is to think through each of the terms in your arithmetic expression and try to imagine the largest value each can assume. For example, if in the integer expression $m = j * k$, the largest expected value for j is 200 and the largest expected value for k is 25 , the largest value you can expect for m is $200 * 25 = 5,000$. This is OK on a 32-bit machine since the largest integer is $2,147,483,647$. On the other hand, if the largest expected value for j is $200,000$ and the largest expected value for k is $100,000$, the largest value you can expect for m is $200,000 * 100,000 = 20,000,000,000$. This is not OK since $20,000,000,000$ is larger than $2,147,483,647$. In this case, you would have to use 64-bit integers or floating-point numbers to accommodate the largest expected value of m .

Also consider future extensions to the program. If m will never be bigger than $5,000$, that's great. But if you expect m to grow steadily for several years, take that into account.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

12.3. Floating-Point Numbers



KEY POINT

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 Microsoft 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.

Following are a few specific guidelines for using floating-point numbers:

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 .

Cross-Reference

For algorithms books that describe ways to solve these problems, see "[Additional Resources on Data Types](#)" in [Section 10.1](#).

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 following example shows two variables, nominal and sum, that should be equal but aren't.

1 is equal to 2 for sufficiently large values of 1.

—Anonymous

Java Example of a Bad Comparison of Floating-Point Numbers

```
double nominal = 1.0;           <-- 1
double sum = 0.0;

for ( int i = 0; i < 10; i++ ) {
    sum += 0.1;               <-- 2
}

if ( nominal == sum ) {        <-- 3
    System.out.println( "Numbers are the same." );
}
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

12.4. Characters and Strings

This section provides some tips for using strings. The first applies to strings in all languages.

Cross-Reference

Issues for using magic characters and strings are similar to those for magic numbers discussed in [Section 12.1, "Numbers in General."](#)

Avoid magic characters and strings Magic characters are literal characters (such as 'A') and magic strings are literal strings (such as "Gigamatic Accounting Program") 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 exist:

- 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 next example, the meaning of 0x1B isn't clear. The use of the ESCAPE constant makes the meaning more obvious.

C++ Examples of Comparisons Using Strings

```
if ( input_char == 0x1B ) ...           <-- 1  
if ( input_char == ESCAPE ) ...         <-- 2
```

(1)Bad!

(2)Better!

Watch for off-by-one errors Because substrings can be indexed much as arrays are, watch for off-by-one errors that read or write past the end of a string.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)



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 next fragment, 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 )  
) {  
    ...  
}
```

Cross-Reference

For details on using comments to document your program, see [Chapter 32, "Self-Documenting Code."](#)

Cross-Reference

For an example of using a boolean function to document your program, see "[Making Complicated Expressions Simple](#)" in [Section 19.1](#).

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 previous example, 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

Complex Test

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

12.6. Enumerated Types

An enumerated type is a type of data that allows each member of a class of objects to be described in English. Enumerated types are available in C++ and Visual Basic and are generally used when you know all the possible values of a variable and want to express them in words. Here are some examples of enumerated types in Visual Basic:

Visual Basic Examples of Enumerated Types

```
Public Enum Color
```

```
    Color_Red
```

```
    Color_Green
```

```
    Color_Blue
```

```
End Enum
```

```
Public Enum Country
```

```
    Country_China
```

```
    Country_England
```

```
    Country_France
```

```
    Country_Germany
```

```
    Country_India
```

```
    Country_Japan
```

```
    Country_Usa
```

```
End Enum
```

```
Public Enum Output
```

```
    Output_Screen
```

```
    Output_Printer
```

```
    Output_File
```

```
End Enum
```

Enumerated types are a powerful alternative to shopworn schemes in which you explicitly say, "1 stands for red, 2 stands for green, 3 stands for blue...." This ability suggests several guidelines for using enumerated types:

Use enumerated types for readability

```
Instead of writing statements like
```

```
if chosenColor = 1
```

you can write more readable expressions like

```
if chosenColor = Color_Red
```

Anytime you see a numeric literal, ask whether it makes sense to replace it with an enumerated type.

Enumerated types are especially useful for defining routine parameters. Who knows what the parameters to this function call are?

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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—MAXIMUM_EMPLOYEES rather than 1000, for instance.

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.

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 following example, you use LOCAL_NUMBER_LENGTH to describe the length of employee phone numbers rather than the literal 7.

Good Visual Basic Example of Using a Named Constant in a Data Declaration

```
Const AREA_CODE_LENGTH = 3

Const LOCAL_NUMBER_LENGTH = 7          <-- 1

...

Type PHONE_NUMBER

    areaCode( AREA_CODE_LENGTH ) As String
    localNumber( LOCAL_NUMBER_LENGTH ) As String      <-- 2

End Type

...

' make sure all characters in phone number are digits

For iDigit = 1 To LOCAL_NUMBER_LENGTH          <-- 3
    If ( phoneNumber.localNumber( iDigit ) < "0" ) Or _
        ( "9" < phoneNumber.localNumber( iDigit ) ) Then
        ' do some error processing
    ...

```

(1)LOCAL_NUMBER_LENGTH is declared as a constant here.

(2)It's used here.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

12.8. Arrays

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.



Make sure that all array indexes are within the bounds of the array **KEY POINT** 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.

Consider using containers instead of arrays, or 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 gotos in a program: such accesses tend to be undisciplined, error prone, and hard to prove correct. They suggest using sets, stacks, and queues, whose elements are accessed sequentially, rather than using arrays.



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.

Consider using container classes that you can 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.

Cross-Reference

Issues in using arrays and loops are similar and related. For details on loops, see [Chapter 16, "Controlling Loops."](#)

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]`. Switching loop indexes is called "index cross-talk." Check for this problem. Better yet, use more meaningful index names than `i` and `j` to make it harder to commit cross-talk mistakes in the first place.

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:

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

12.9. Creating Your Own Types (Type Aliasing)



KEY POINT

Programmer-defined data 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, or test new classes. If you're using C, C++, or another language that allows user-defined types, take advantage of them!

Cross-Reference

In many cases, it's better to create a class than to create a simple data type. For details, see [Chapter 6, "Working Classes."](#)

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  
  
    ...  
  
}  
  
...  
  
Routine2( ... ) {  
  
    Coordinate x;    // x coordinate in meters  
  
    Coordinate y;    // y coordinate in meters  
  
    Coordinate z;    // z coordinate in meters  
  
    ...  
  
}
```

In this code, the variables `latitude`, `longitude`, `elevation`, `x`, `y`, and `z` are all declared to be of type `Coordinate`.

Now suppose that the program changes and you find that you need to use double-precision variables for coordinates after all. Because you defined a type specifically for coordinate data, all you have to change is the type definition. And

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- Working with specific data types means remembering many individual rules for each type. Use this chapter's 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.

Chapter 13. Unusual Data Types

cc2e.com/1378

Contents

- [Structures page 319](#)
- [Pointers page 323](#)
- [Global Data page 335](#)

Related Topics

- Fundamental data types: [Chapter 12](#)
- Defensive programming: [Chapter 8](#)
- Unusual control structures: [Chapter 17](#)
- Complexity in software development: [Section 5.2](#)

Some languages support exotic kinds of data in addition to the data types discussed in [Chapter 12](#), "Fundamental Data Types." [Section 13.1](#) describes when you might still use structures rather than classes in some circumstances. [Section 13.2](#) describes the ins and outs of using pointers. If you've ever encountered problems associated with using global data, [Section 13.3](#) explains how to avoid such difficulties. If you think the data types described in this chapter are not the types you normally read about in modern object-oriented programming books, you're right. That's why the chapter is called "Unusual Data Types."

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

13.1. Structures

The term "structure" refers to data that's built up from other types. Because arrays are a special case, they are treated separately in [Chapter 12](#). This section deals with user-created structured data—structs in C and C++ and Structures in Microsoft Visual Basic. In Java and C++, classes also sometimes perform as structures (when the class consists entirely of public data members with no public routines).

You'll generally want to create classes rather than structures so that you can take advantage of the privacy and functionality offered by classes in addition to the public data supported by structures. But sometimes directly manipulating blocks of data can be useful, so here are some reasons for using structures:

Use structures to clarify data relationships Structures bundle groups of related items together. Sometimes the hardest part of figuring out a program is figuring out which data goes with which other data. It's like going to a small town and asking who's related to whom. You come to find out that everybody's kind of related to everybody else, but not really, and you never get a good answer.

If the data has been carefully structured, figuring out what goes with what is much easier. Here's an example of data that hasn't been structured:

Visual Basic Example of Misleading, Unstructured Variables

```
name = inputName  
  
address = inputAddress  
  
phone = inputPhone  
  
title = inputTitle  
  
department = inputDepartment  
  
bonus = inputBonus
```

Because this data is unstructured, it looks as if all the assignment statements belong together. Actually, name, address, and phone are variables associated with individual employees, and title, department, and bonus are variables associated with a supervisor. The code fragment provides no hint that there are two kinds of data at work. In the code fragment below, the use of structures makes the relationships clearer:

Visual Basic Example of More Informative, Structured Variables

```
employee.name = inputName  
  
employee.address = inputAddress  
  
employee.phone = inputPhone  
  
  
supervisor.title = inputTitle  
supervisor.department = inputDepartment  
supervisor.bonus = inputBonus
```

In the code that uses structured variables, it's clear that some of the data is associated with an employee, other data with a supervisor.

Use structures to simplify operations on blocks of data You can combine related elements into a structure and perform operations on the structure. It's easier to operate on the structure than to perform the same operation on each of the elements. It's also more reliable, and it takes fewer lines of code.

Suppose you have a group of data items that belong together—for instance, data about an employee in a personnel

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

13.2. Pointers



Pointer usage is one of the most error-prone areas of modern programming, to such an extent that modern languages like Java, C#, and Visual Basic don't provide a pointer data type.

KEY POINT 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 problems, 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, a good understanding of pointers will help your understanding of how your programming language works. A liberal dose of defensive programming practices will help even further.

Paradigm for Understanding Pointers

Conceptually, every pointer consists of two parts: a location in memory and a knowledge of how to interpret the contents of that location.

Location in Memory

The location in memory is an address, often expressed in hexadecimal notation. An address on a 32-bit processor would be a 32-bit value, such as 0x0001EA40. The pointer itself contains only this address. To use the data the pointer points to, you have to go to that address and interpret the contents of memory at that location. If you were to look at the memory in that location, it would be just a collection of bits. It has to be interpreted to be meaningful.

Knowledge of How to Interpret the Contents

The knowledge of how to interpret the contents of a location in memory is provided by the base type of the pointer. If a pointer points to an integer, what that really means is that the compiler interprets the memory location given by the pointer as an integer. Of course, you can have an integer pointer, a string pointer, and a floating-point pointer all pointing at the same memory location. But only one of the pointers interprets the contents at that location correctly.

In thinking about pointers, it's helpful to remember that memory doesn't have any inherent interpretation associated with it. It is only through use of a specific type of pointer that the bits in a particular location are interpreted as meaningful data.

[Figure 13-1](#) shows several views of the same location in memory, interpreted in several different ways.

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

[\[View full size image\]](#)

0A	61	62	63	64	65	66	67	68	69	6A
----	----	----	----	----	----	----	----	----	----	----

Viewed as: Raw memory contents used for further examples (in hex)

Interpreted as: No interpretation possible without associated pointer variable

0A	61	62	63	64	65	66	67	68	69	6A
----	----	----	----	----	----	----	----	----	----	----

Viewed as: String[10] (in Visual Basic format with length byte first)

Interpreted as: abcdefghij

0A	61	62	63	64	65	66	67	68	69	6A
----	----	----	----	----	----	----	----	----	----	----

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 class. But accessibility anywhere within a single class does not by itself mean that a variable is global.

Cross-Reference

For details on the differences between global data and class data, see "[Class data mistaken for global data](#)" in [Section 5.3](#).

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 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.

KEY POINT

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 this example, theAnswer is a global variable:

Visual Basic Example of a Side-Effect Problem

```
theAnswer = GetTheAnswer()           <-- 1  
otherAnswer = GetOtherAnswer()       <-- 2  
averageAnswer = (theAnswer + otherAnswer) / 2    <-- 3
```

(1)theAnswer is a global variable.

(2)GetOtherAnswer() changes theAnswer.

(3)averageAnswer is wrong.

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/1385

Following are more resources that cover unusual data types:

Maguire, Steve. Writing Solid Code. Redmond, WA: Microsoft Press, 1993. [Chapter 3](#) contains an excellent discussion of the hazards of pointer use and numerous specific tips for avoiding problems with pointers.

Meyers, Scott. Effective C++, 2d ed. Reading, MA: Addison-Wesley, 1998; Meyers, Scott, More Effective C++. Reading, MA: Addison-Wesley, 1996. As the titles suggest, these books contain numerous specific tips for improving C++ programs, including guidelines for using pointers safely and effectively. More Effective C++ in particular contains an excellent discussion of C++'s memory management issues.

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 of 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?

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- Structures can help make programs less complicated, easier to understand, and easier to maintain.
- Whenever you consider using a structure, consider whether a class would work better.
- Pointers are error-prone. Protect yourself by using access routines or classes and defensive-programming practices.
- Avoid global variables, not just because they're dangerous, but because you can replace them with something better.
- If you can't avoid global variables, work with them through access routines. Access routines give you everything that global variables give you, and more.

Part IV: Statements

[In this part:](#)

[Chapter 14. Organizing Straight-Line Code](#)

[Chapter 15. Using Conditionals](#)

[Chapter 16. Controlling Loops](#)

[Chapter 17. Unusual Control Structures](#)

[Chapter 18. Table-Driven Methods](#)

[Chapter 19. General Control Issues](#)

In this part:

[Chapter 14: Organizing Straight-Line Code](#)

[Chapter 15: Using Conditionals](#)

[Chapter 16: Controlling Loops](#)

[Chapter 17: Unusual Control Structures](#)

[Chapter 18: Table-Driven Methods](#)

[Chapter 19: General Control Issues](#)

Chapter 14. Organizing Straight-Line Code

cc2e.com/1465

Contents

- [Statements That Must Be in a Specific Order page 347](#)
- [Statements Whose Order Doesn't Matter page 351](#)

Related Topics

- General control topics: [Chapter 19](#)
- Code with conditionals: [Chapter 15](#)
- Code with loops: [Chapter 16](#)
- Scope of variables and objects: [Section 10.4, "Scope"](#)

This chapter turns from a data-centered view of programming to a statement-centered view. It introduces the simplest kind of control flow: putting statements and blocks of statements in sequential order.

Although organizing straight-line code is a relatively simple task, some organizational subtleties influence code quality, correctness, readability, and maintainability.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

14.1. Statements That Must Be in a Specific Order

The easiest sequential statements to order are those in which the order counts. Here's an example:

Java Example of Statements in Which Order Counts

```
data = ReadData();  
  
results = CalculateResultsFromData( data );  
  
PrintResults( results );
```

Unless something mysterious is happening with this code fragment, the statement must be executed in the order shown. The data must be read before the results can be calculated, and the results must be calculated before they can be printed.

The underlying concept in this example is that of dependencies. The third statement depends on the second, the second on the first. In this example, the fact that one statement depends on another is obvious from the routine names. In the following code fragment, the dependencies are less obvious:

Java Example of Statements in Which Order Counts, but Not Obviously

```
revenue.ComputeMonthly();  
  
revenue.ComputeQuarterly();  
  
revenue.ComputeAnnual();
```

In this case, the quarterly revenue calculation assumes that the monthly revenues have already been calculated. A familiarity with accounting—or even common sense—might tell you that quarterly revenues have to be calculated before annual revenues. There is a dependency, but it's not obvious merely from reading the code. And here, the dependencies aren't obvious—they're literally hidden:

Visual Basic Example of Statements in Which Order Dependencies Are Hidden

```
ComputeMarketingExpense  
  
ComputeSalesExpense  
  
ComputeTravelExpense  
  
ComputePersonnelExpense  
  
DisplayExpenseSummary
```

Suppose that `ComputeMarketingExpense()` initializes the class member variables that all the other routines put their data into. In such a case, it needs to be called before the other routines. How could you know that from reading this code? Because the routine calls don't have any parameters, you might be able to guess that each of these routines accesses class data. But you can't know for sure from reading this code.



When statements have dependencies that require you to put them in a certain order, take steps to make the dependencies clear. Here are some simple guidelines for ordering

KEY POINT statements:

Organize code so that dependencies are obvious In the Microsoft Visual Basic example just presented, `ComputeMarketingExpense()` shouldn't initialize the class member variables. The routine names suggest that `ComputeMarketingExpense()` is similar to `ComputeSalesExpense()`, `ComputeTravelExpense()`, and the other routines except that it works with marketing data rather than with sales data or other data. Having `ComputeMarketingExpense()` initialize the member variable is an arbitrary practice you should avoid. Why should initialization be done in that routine instead of one of the other two? Unless you can think of a good reason, you should write another routine, `InitializeExpenseData()`, to initialize the member variable. The routine's name is a clear

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

14.2. Statements Whose Order Doesn't Matter

You might encounter cases in which it seems as if the order of a few statements or a few blocks of code doesn't matter at all. One statement doesn't depend on, or logically follow, another statement. But ordering affects readability, performance, and maintainability, and in the absence of execution-order dependencies, you can use secondary criteria to determine the order of statements or blocks of code. The guiding principle is the Principle of Proximity: Keep related actions together.

Making Code Read from Top to Bottom

As a general principle, make the program read from top to bottom rather than jumping around. Experts agree that top-to-bottom order contributes most to readability. Simply making the control flow from top to bottom at run time isn't enough. If someone who is reading your code has to search the whole program to find needed information, you should reorganize the code. Here's an example:

C++ Example of Bad Code That Jumps Around

```
MarketingData marketingData;

SalesData salesData;

TravelData travelData;

travelData.ComputeQuarterly();

salesData.ComputeQuarterly();

marketingData.ComputeQuarterly();

salesData.ComputeAnnual();

marketingData.ComputeAnnual();

travelData.ComputeAnnual();

salesData.Print();

travelData.Print();

marketingData.Print();
```

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

```
MarketingData marketingData;

marketingData.ComputeQuarterly();

marketingData.ComputeAnnual();

marketingData.Print();
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- The strongest principle for organizing straight-line code is ordering dependencies.
- Dependencies should be made obvious through the use of good routine names, parameter lists, comments, and—if the code is critical enough—housekeeping variables.
- If code doesn't have order dependencies, keep related statements as close together as possible.

Chapter 15. Using Conditionals

cc2e.com/1538

Contents

- [if Statements page 355](#)
- [case Statements page 361](#)

Related Topics

- Taming deep nesting: [Section 19.4](#)
- General control issues: [Chapter 19](#)
- Code with loops: [Chapter 16](#)
- Straight-line code: [Chapter 14](#)
- Relationship between data types and control structures: [Section 10.7](#)

A conditional is a statement that controls the execution of other statements; execution of the other statements is "conditioned" on statements such as if, else, case, and switch. Although it makes sense logically to refer to loop controls such as while and for as conditionals too, by convention they've been treated separately. [Chapter 16](#), "[Controlling Loops](#)," will examine while and for statements.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

15.1. if Statements

Depending on the language you're using, you might be able to use any of several kinds of if statements. The simplest is the plain if or if-then statement. The if-then-else is a little more complex, and chains of if-then-else-if are the most complex.

Plain if-then Statements

Follow these guidelines when writing if statements:



Write the nominal path through the code first; then write the unusual cases KEY POINT Write your code so that the normal path through the code is clear. Make sure that the rare cases don't obscure the normal path of execution. This is important for both readability and performance.

Make sure that you branch correctly on equality Using `>` instead of `>=` or `<` instead of `<=` is analogous to making an off-by-one error in accessing an array or computing a loop index. In a loop, think through the endpoints to avoid an off-by-one error. In a conditional statement, think through the equals case to avoid one.

Put the normal case after the if rather than after the else Put the case you normally expect to process first. This is in line with the general principle of putting code that results from a decision as close as possible to the decision. Here's a code example that does a lot of error processing, haphazardly checking for errors along the way:

Cross-Reference

For other ways to handle error-processing code, see "[Summary of Techniques for Reducing Deep Nesting](#)" in [Section 19.4](#).

Visual Basic Example of Code That Processes a Lot of Errors Haphazardly

```
OpenFile( inputFile, status )

If ( status = Status_Error ) Then
    errorType = FileOpenError          <-- 1

Else
    ReadFile( inputFile, fileData, status )      <-- 2

    If ( status = Status_Success ) Then
        SummarizeFileData( fileData, summaryData, status )      <-- 3

        If ( status = Status_Error ) Then
            errorType = ErrorType_DataSummaryError           <-- 4

        Else
            PrintSummary( summaryData )                  <-- 5

            SaveSummaryData( summaryData, status )

            If ( status = Status_Error ) Then
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 a time. Visual Basic supports case for ordinal types and has powerful shorthand notations for expressing ranges and combinations of values. Many scripting languages don't support case statements at all.

The following sections present guidelines for using case statements effectively:

Choosing the Most Effective Ordering of Cases

You can choose from among a variety of ways to organize the cases in a case statement. If you have a small case statement with three options and three corresponding lines of code, the order you use doesn't matter much. If you have a long case statement—for example, a case statement that handles dozens of events in an event-driven program—order is significant. Following are some ordering possibilities:

Order cases alphabetically or numerically If cases are equally important, putting them in A-B-C order improves readability. That way a specific case is easy to pick out of the group.

Put the normal case first If you have one normal case and several exceptions, put the normal case first. Indicate with comments that it's the normal case and that the others are unusual.

Order cases by frequency Put the most frequently executed cases first and the least frequently executed last. This approach has two advantages. First, human readers can find the most common cases easily. Readers scanning the list for a specific case are likely to be interested in one of the most common cases, and putting the common ones at the top of the code makes the search quicker.

Tips for Using case Statements

Here are several tips for using case statements:

Keep the actions of each case simple Keep the code associated with each case short. Short code following each case helps make the structure of the case statement clear. If the actions performed for a case are complicated, write a routine and call the routine from the case rather than putting the code into the case itself.

Cross-Reference

For other tips on simplifying code, see [Chapter 24, "Refactoring."](#)

Don't make up phony variables 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. For example, don't do this:



Java Example of Creating a Phony Variable---Bad Practice

```
action = userCommand[ 0 ];  
  
switch ( action ) {  
  
    case 'c':
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- For simple if-else statements, 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.
- For if-then-else chains and case statements, choose an order that maximizes read-ability.
- To trap errors, use the default clause in a case statement or the last else in a chain of if-then-else statements.
- All control constructs are not created equal. Choose the control construct that's most appropriate for each section of code.

Chapter 16. Controlling Loops

cc2e.com/1609

Contents

- [Selecting the Kind of Loop page 367](#)
- [Controlling the Loop page 373](#)
- [Creating Loops Easily—From the Inside Out page 385](#)
- [Correspondence Between Loops and Arrays page 387](#)

Related Topics

- Taming deep nesting: [Section 19.4](#)
- General control issues: [Chapter 19](#)
- Code with conditionals: [Chapter 15](#)
- Straight-line code: [Chapter 14](#)
- Relationship between control structures and data types: [Section 10.7](#)

"Loop" is an informal term that refers to any kind of iterative control structure—any structure that causes a program to repeatedly execute a block of code. Common loop types are for, while, and do-while in C++ and Java, and For-Next, While-Wend, and Do-Loop-While in Microsoft Visual Basic. Using loops is one of the most complex aspects of programming; knowing how and when to use each kind of loop is a decisive factor in constructing high-quality software.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

16.1. Selecting the Kind of Loop

In most languages, you'll use a few kinds of loops:

- The counted loop is performed a specific number of times, perhaps one time for each employee.
- The continuously evaluated loop doesn't know ahead of time how many times it will be executed and tests whether it has finished on each iteration. For example, it runs while money remains, until the user selects quit, or until it encounters an error.
- The endless loop executes forever once it has started. It's the kind you find in embedded systems such as pacemakers, microwave ovens, and cruise controls.
- The iterator loop performs its action once for each element in a container class.

The kinds of loops are differentiated first by flexibility—whether the loop executes a specified number of times or whether it tests for completion on each iteration.

The kinds of loops are also differentiated by the location of the test for completion. You can put the test at the beginning, the middle, or the end of the loop. This characteristic tells you whether the loop executes at least once. If the loop is tested at the beginning, its body isn't necessarily executed. If the loop is tested at the end, its body is executed at least once. If the loop is tested in the middle, the part of the loop that precedes the test is executed at least once, but the part of the loop that follows the test isn't necessarily executed at all.

Flexibility and the location of the test determine the kind of loop to choose as a control structure. [Table 16-1](#) shows the kinds of loops in several languages and describes each loop's flexibility and test location.

Table 16-1. The Kinds of Loops

Language	Kind of Loop	Flexibility	Test Location
Visual Basic	For-Next	rigid	beginning
	While-Wend	flexible	beginning
	Do-Loop-While	flexible	beginning or end
	For-Each	rigid	beginning
C, C++, C#, Java	for	flexible	beginning
	while	flexible	beginning
	do-while	flexible	end
	foreach ^[*]	rigid	beginning

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

16.2. Controlling the Loop

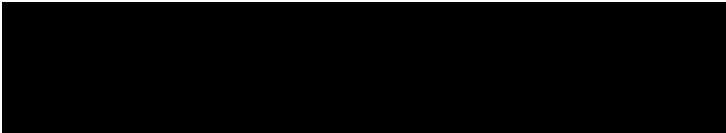
What can go wrong with a loop? Any answer would have to include incorrect or omitted loop initialization, omitted initialization of accumulators or other variables related to the loop, improper nesting, incorrect termination of the loop, forgetting to increment a loop variable or incrementing the variable incorrectly, and indexing an array element from a loop index incorrectly.



KEY POINT 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 ) {
```



```
}
```

Cross-Reference

If you use the while (true)-break technique described earlier, the exit condition is inside the black box. Even if you use only one exit condition, you lose the benefit of treating the loop as a black box.

What are the conditions under which this loop terminates? Clearly, all you know is that either inputFile.EndOfFile() becomes true or MoreDataAvailable becomes false.

Entering the Loop

Use these guidelines when entering a loop:

Enter the loop from one location only A variety of loop-control structures allows you to test at the beginning, middle, or end of a loop. These structures are rich enough to allow you to enter the loop from the top every time. You don't need to enter at multiple locations.

Put initialization code directly before the loop The Principle of Proximity advocates putting related statements together. If related statements are strewn across a routine, it's easy to overlook them during modification and to make the modifications incorrectly. If related statements are kept together, it's easier to avoid errors during modification.

Keep loop-initialization statements with the loop they're related to. If you don't, you're more likely to cause errors when you generalize the loop into a bigger loop and forget to modify the initialization code. The same kind of error can occur when you move or copy the loop code into a different routine without moving or copying its initialization code. Putting initializations away from the loop—in the data-declaration section or in a housekeeping section at the top of the routine that contains the loop—invites initialization troubles.

Cross-Reference

For more on limiting the scope of loop variables, see "[Limit the scope of loop-index variables to the loop itself](#)" later

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

16.3. Creating Loops Easily—From the Inside Out

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. 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.

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.

Cross-Reference

Coding a loop from the inside out is similar to the process described in [Chapter 9, "The Pseudocode Programming Process."](#)

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.

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.

Step 2: Creating a Loop from the Inside Out (Pseudocode Example)

```
rate = table[ ]           <-- 1  
  
totalRate = totalRate + rate
```

(1)table doesn't have any indexes yet.

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 is a variable that holds the total of the rates.

Next, put in indexes for the table array:

Step 3: Creating a Loop from the Inside Out (Pseudocode Example)

```
rate = table[ census.Age ][ census.Gender ]  
  
totalRate = totalRate + rate
```

The array is accessed by age and sex, so census.Age and census.Gender are used to index the array. The example assumes that census is a structure that holds information about people in the group to be rated.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

16.4. Correspondence Between Loops and Arrays

Cross-Reference

For further discussion of the correspondence between loops and arrays, see [Section 10.7, "Relationship Between Data Types and Control Structures."](#)

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, these Java for loop indexes 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 just shown. Here's an APL code fragment that performs the same operation:

APL Example of an Array Multiplication

```
product <- a × b
```

The APL is simpler and less error-prone. It uses only three 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.

cc2e.com/1616

Checklist: Loops

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?
-

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 through the loop carefully to verify that it operates normally under each case and terminates under all possible conditions.

Chapter 17. Unusual Control Structures

cc2e.com/1778

Contents

- [Multiple Returns from a Routine page 391](#)
- [Recursion page 393](#)
- [goto page 398](#)
- [Perspective on Unusual Control Structures page 408](#)

Related Topics

- General control issues: [Chapter 19](#)
- Straight-line code: [Chapter 14](#)
- Code with conditionals: [Chapter 15](#)
- Code with loops: [Chapter 16](#)
- Exception handling: [Section 8.4](#)

Several control constructs exist in a hazy twilight zone somewhere between being leading-edge and being discredited and disproved—often in both places at the same time! These constructs aren't available in all languages but can be useful when used with care in those languages that do offer them.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

17.1. Multiple Returns from a Routine

Most languages support some means of exiting from a routine partway through the routine. The return and exit statements are control constructs that enable a program to exit from a routine at will. They cause the routine to terminate through the normal exit channel, returning control to the calling routine. The word return is used here as a generic term for return in C++ and Java, Exit Sub and Exit Function in Microsoft Visual Basic, and similar constructs. Here are guidelines for using the return statement:



Use a return when it enhances readability **KEY POINT** 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 require any further cleanup once it detects an error, not returning immediately means that you have to write more code.

The following is a good example of a case in which returning from multiple places in a routine makes sense:

C++ Example of a Good Multiple Return from a Routine

```
Comparison Compare( int value1, int value2 ) {           <-- 1
    if ( value1 < value2 ) {
        return Comparison_LessThan;
    }
    else if ( value1 > value2 ) {
        return Comparison_GreaterThan;
    }
    return Comparison_Equal;
}
```

(1) This routine returns a Comparison enumerated type.

Other examples are less clear-cut, as the next subsection illustrates.

Use guard clauses (early returns or exits) to simplify complex error processing Code that has to check for numerous error conditions before performing its nominal actions can result in deeply indented code and can obscure the nominal case, as shown here:

Visual Basic Code That Obscures the Nominal Case

```
If file.validName() Then
    If file.Open() Then
        If encryptionKey.valid() Then
            If file.Decrypt( encryptionKey ) Then
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

17.2. Recursion

In recursion, a routine solves a small part of a problem itself, divides the problem into smaller pieces, and then calls itself to solve each of the smaller pieces. Recursion is usually called into play when a small part of the problem is easy to solve and a large part is easy to decompose into smaller pieces.



KEY POINT

Recursion isn't useful often, but when used judiciously it produces elegant solutions, as in this example in which a sorting algorithm makes excellent use of recursion:

Java Example of a Sorting Algorithm That Uses Recursion

```
void QuickSort( int firstIndex, int lastIndex, String [] names ) {  
    if ( lastIndex > firstIndex ) {  
        int midPoint = Partition( firstIndex, lastIndex, names );  
        QuickSort( firstIndex, midPoint-1, names );           <-- 1  
        QuickSort( midPoint+1, lastIndex, names )           <-- 1  
    }  
}
```

(1)Here are the recursive calls.

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—such as (`lastIndex <= firstIndex`)—it stops calling itself.

For a small group of problems, recursion can produce simple, elegant solutions. For a slightly larger group of problems, it can produce simple, elegant, hard-to-understand solutions. For most problems, it produces massively complicated solutions—in those cases, simple iteration is usually more understandable. Use recursion selectively.

Example of Recursion

Suppose you have a data type that represents a maze. A maze is basically a grid, and at each point on the grid you might be able to turn left, turn right, move up, or move down. You'll often be able to move in more than one direction.

How do you write a program to find its way through the maze, as shown in [Figure 17-1](#)? If you use recursion, the answer is fairly straightforward. You start at the beginning and then try all possible paths until you find your way out of the maze. The first time you visit a point, you try to move left. If you can't move left, you try to go up or down, and if you can't go up or down, you try to go right. You don't have to worry about getting lost because you drop a few bread crumbs on each spot as you visit it, and you don't visit the same spot twice.

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

17.3. goto

cc2e.com/1785

You might think the debate related to gotos is extinct, but a quick trip through modern source-code repositories like SourceForge.net shows that the goto is still alive and well and living deep in your company's server. Moreover, modern equivalents of the goto debate still crop up in various guises, including debates about multiple returns, multiple loop exits, named loop exits, error processing, and exception handling.

The Argument Against gotos

The general argument against gotos is that code without gotos is higher-quality code. The famous letter that sparked the original controversy was Edsger Dijkstra's "Go To Statement Considered Harmful" in the March 1968 Communications of the ACM. Dijkstra observed that the quality of code was inversely proportional to the number of gotos the programmer used. In subsequent work, Dijkstra has argued that code that doesn't contain gotos can more easily be proven correct.

Code containing gotos is hard to format. Indentation should be used to show logical structure, and gotos have an effect on logical structure. Using indentation to show the logical structure of a goto and its target, however, is difficult or impossible.

Use of gotos defeats compiler optimizations. Some optimizations depend on a program's flow of control residing within a few statements. An unconditional goto makes the flow harder to analyze and reduces the ability of the compiler to optimize the code. Thus, even if introducing a goto produces an efficiency at the source-language level, it may well reduce overall efficiency by thwarting compiler optimizations.

Proponents of gotos sometimes argue that they make code faster or smaller. But code containing gotos is rarely the fastest or smallest possible. Donald Knuth's marvelous, classic article "Structured Programming with go to Statements" gives several examples of cases in which using gotos makes for slower and larger code (Knuth 1974).

In practice, the use of gotos leads to the violation of the principle that code should flow strictly from top to bottom. Even if gotos aren't confusing when used carefully, once gotos are introduced, they spread through the code like termites through a rotting house. If any gotos are allowed, the bad creep in with the good, so it's better not to allow any of them.

Overall, experience in the two decades that followed the publication of Dijkstra's letter showed the folly of producing goto-laden code. In a survey of the literature, Ben Shneiderman concluded that the evidence supports Dijkstra's view that we're better off without the goto (1980), and many modern languages, including Java, don't even have gotos.

The Argument for gotos

The argument for the goto is characterized by an advocacy of its careful use in specific circumstances rather than its indiscriminate use. Most arguments against gotos speak against indiscriminate use. The goto controversy erupted when Fortran was the most popular language. Fortran had no presentable loop structures, and in the absence of good advice on programming loops with gotos, programmers wrote a lot of spaghetti code. Such code was undoubtedly correlated with the production of low-quality programs, but it has little to do with the careful use of a goto to make up for a gap in a modern language's capabilities.

A well-placed goto can eliminate the need for duplicate code. Duplicate code leads to problems if the two sets of code are modified differently. Duplicate code increases the size of source and executable files. The bad effects of the goto are outweighed in such a case by the risks of duplicate code.

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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

17.4. Perspective on Unusual Control Structures

At one time or another, someone thought that each of the following control structures was a good idea:

- Unrestricted use of gotos
- Ability to compute a goto target dynamically and jump to the computed location
- Ability to use goto to jump from the middle of one routine into the middle of another routine
- Ability to call a routine with a line number or label that allowed execution to begin somewhere in the middle of the routine
- Ability to have the program generate code on the fly and then execute the code it just wrote

At one time, each of these ideas was regarded as acceptable or even desirable, even though now they all look hopelessly quaint, outdated, or dangerous. The field of software development has advanced largely through restricting what programmers can do with their code. Consequently, I view unconventional control structures with strong skepticism. I suspect that the majority of constructs in this chapter will eventually find their way onto the programmer's scrap heap along with computed goto labels, variable routine entry points, self-modifying code, and other structures that favored flexibility and convenience over structure and the ability to manage complexity.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/1792

The following resources also address unusual control structures:

Returns

Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999. In the description of the refactoring called "Replace Nested Conditional with Guard Clauses," Fowler suggests using multiple return statements from a routine to reduce nesting in a set of if statements. Fowler argues that multiple returns are an appropriate means of achieving greater clarity, and that no harm arises from having multiple returns from a routine.

gotos

cc2e.com/1799

These articles contain the whole goto debate. It erupts from time to time in most work-places, textbooks, and magazines, but you won't hear anything that wasn't fully explored 20 years ago.

Dijkstra, Edsger. "Go To Statement Considered Harmful." *Communications of the ACM* 11, no. 3 (March 1968): 147–48, also available from <http://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, NJ: 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 goto-less 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 goto-less alternatives.

The response that Rubin's letter generated was more interesting than the letter itself. For five months, *Communications of the ACM* (CACM) 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.
- *Communications of the ACM* 30, no. 6 (June 1987): 475–78.
-

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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.

Chapter 18. Table-Driven Methods

cc2e.com/1865

Contents

- [General Considerations in Using Table-Driven Methods page 411](#)
- [Direct Access Tables page 413](#)
- [Indexed Access Tables page 425](#)
- [Stair-Step Access Tables page 426](#)
- [Other Examples of Table Lookups page 429](#)

Related Topics

- Information hiding: "[Hide Secrets \(Information Hiding\)](#)" in [Section 5.3](#)
- Class design: [Chapter 6](#)
- Using decision tables to replace complicated logic: in [Section 19.1](#)
- Substitute table lookups for complicated expressions: in [Section 26.1](#)

A table-driven method is a scheme that allows you to look up information in a table rather than using logic statements (if and case) to figure it out. Virtually anything you can select with logic statements, you can select with tables instead. In simple cases, logic statements are easier and more direct. As the logic chain becomes more complex, tables become increasingly attractive.

If you're already familiar with table-driven methods, this chapter might be just a review. In that case, you might examine "[Flexible-Message-Format Example](#)" in [Section 18.2](#) for a good example of how an object-oriented design isn't necessarily better than any other kind of design just because it's object-oriented, and then you might move on to the discussion of general control issues in [Chapter 19](#).

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

18.1. General Considerations in Using Table-Driven Methods



Used in appropriate circumstances, table-driven code is simpler than complicated logic,

easier to modify, and more efficient. Suppose you wanted to classify characters into letters,

punctuation marks, and digits; you might use a complicated chain of logic like this one:

Java Example of Using Complicated Logic to Classify a Character

```
if ( ( ( 'a' <= inputChar ) && ( inputChar <= 'z' ) ) ||  
    ( ( 'A' <= inputChar ) && ( inputChar <= 'Z' ) ) ) {  
  
    charType = CharacterType.Letter;  
  
}  
  
else if ( ( inputChar == ' ' ) || ( inputChar == ',' ) ||  
          ( inputChar == '.' ) || ( inputChar == '!' ) || ( inputChar == '(' ) ||  
          ( inputChar == ')' ) || ( inputChar == ':' ) || ( inputChar == ';' ) ||  
          ( inputChar == '?' ) || ( inputChar == '-' ) ) {  
  
    charType = CharacterType.Punctuation;  
  
}  
  
else if ( ( '0' <= inputChar ) && ( inputChar <= '9' ) ) {  
  
    charType = CharacterType.Digit;  
  
}
```

If you used a lookup table instead, you'd store the type of each character in an array that's accessed by character code. The complicated code fragment just shown would be replaced by this:

Java Example of Using a Lookup Table to Classify a Character

```
charType = charTypeTable[ inputChar ];
```

This fragment assumes that the charTypeTable array has been set up earlier. You put your program's knowledge into its data rather than into its logic—in the table instead of in the if tests.

Two Issues in Using Table-Driven Methods



When you use table-driven methods, you have to address two issues. First you have to address the question of how to look up entries in the table. You can use some data to access

KEY POINT a table directly. If you need to classify data by month, for example, keying into a month table is straightforward. You can use an array with indexes 1 through 12.

Other data is too awkward to be used to look up a table entry directly. If you need to classify data by Social Security Number, for example, you can't use the Social Security Number to key into the table directly unless you can afford to store 999-99-9999 entries in your table. You're forced to use a more complicated approach. Here's a list of ways to look up an entry in a table:

-

Direct access

-

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

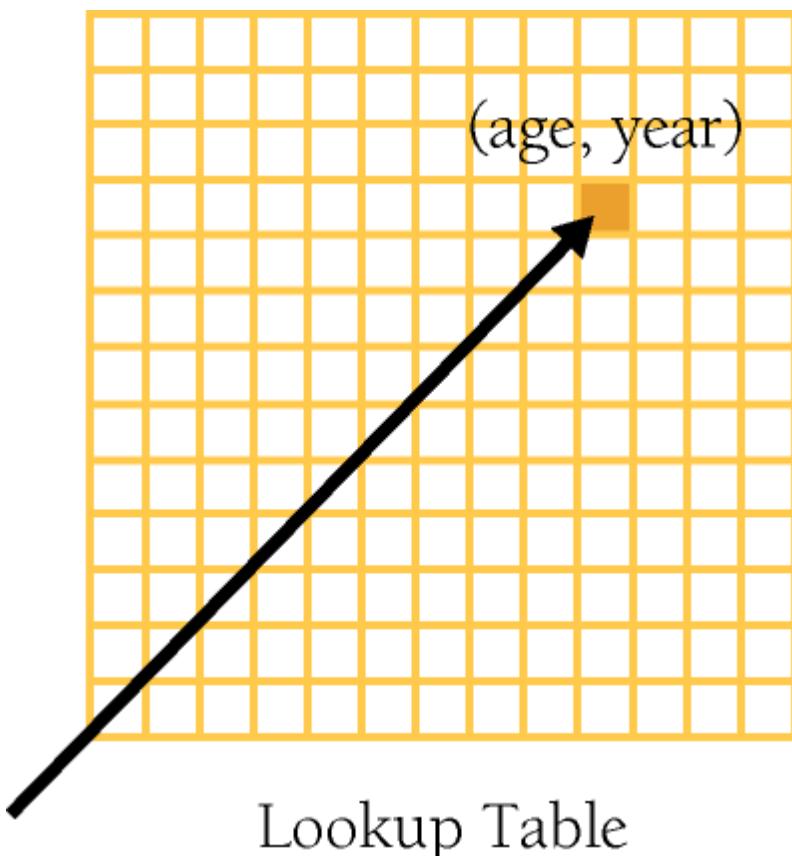
[**< Free Open Study >**](#)

[NEXT ▶](#)

18.2. Direct Access Tables

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.

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



Lookup Table

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:

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
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

18.3. Indexed Access Tables

Sometimes a simple mathematical transformation isn't powerful enough to make the jump from data like Age to a table key. Some such cases are suited to the use of an indexed access scheme.

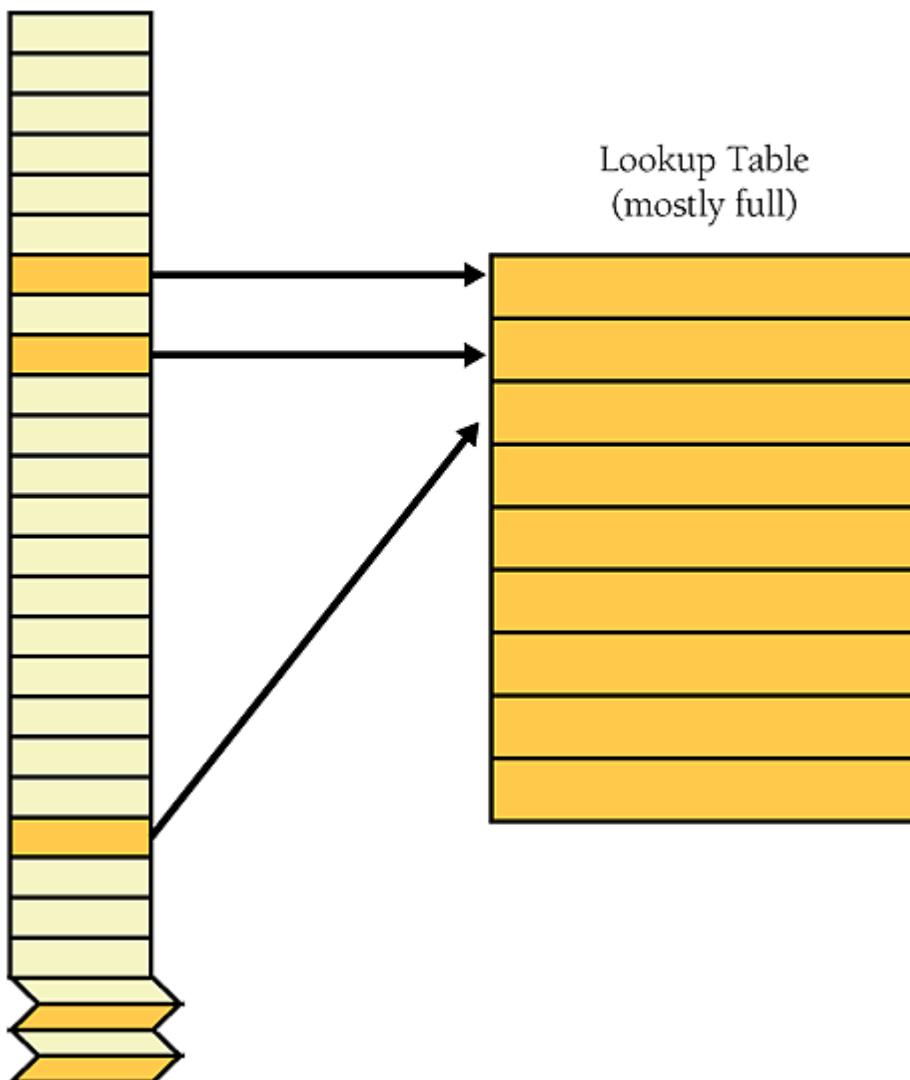
When you use indexes, you use the primary data to look up a key in an index table and then you use the value from the index table to look up the main data you're interested in.

Suppose you run a warehouse and have an inventory of about 100 items. Suppose further that each item has a four-digit part number that ranges from 0000 through 9999. In this case, if you want to use the part number to key directly into a table that describes some aspect of each item, you set up an index array with 10,000 entries (from 0 through 9999). The array is empty except for the 100 entries that correspond to part numbers of the 100 items in your warehouse. As [Figure 18-4](#) shows, those entries point to an item-description table that has far fewer than 10,000 entries.

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

[\[View full size image\]](#)

Array of Indexes into
Lookup Table
(mostly empty)



Indexed access schemes offer two main advantages. First, if each of the entries in the main lookup table is large, it

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

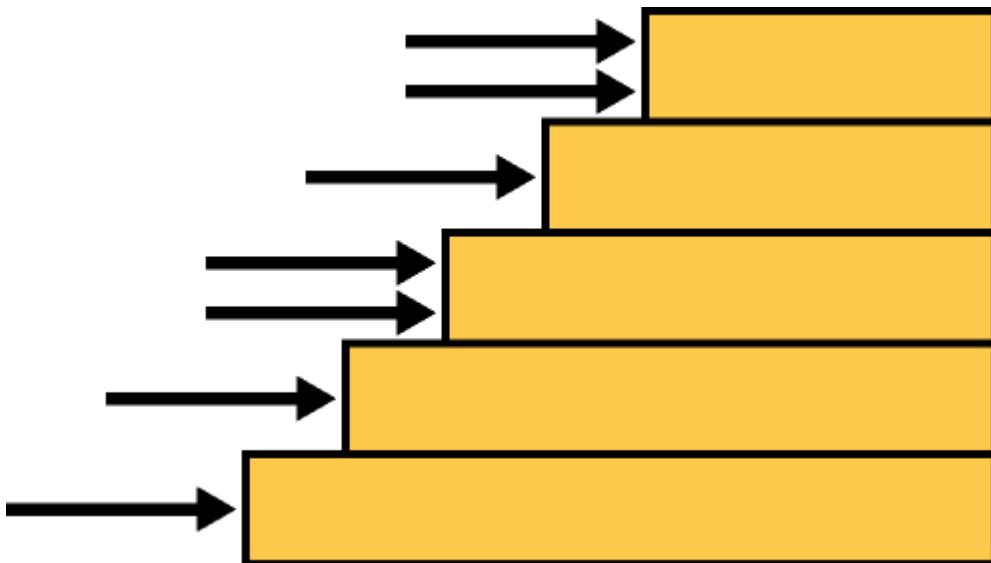
[NEXT ▶](#)

18.4. Stair-Step Access Tables

Yet another kind of table access is the stair-step method. This access method isn't as direct as an index structure, but it doesn't waste as much data space.

The general idea of stair-step structures, illustrated in [Figure 18-5](#), is that entries in a table are valid for ranges of data rather than for distinct data points.

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



For example, if you're writing a grading program, the "B" entry range might be from 75 percent to 90 percent. Here's a range of grades you might have to program someday:

$\geq 90.0\%$ A

$< 90.0\%$ B

$< 75.0\%$ C

$< 65.0\%$ D

$< 50.0\%$ F

This is an ugly range for a table lookup because you can't use a simple data-transformation function to key into the letters A through F. An index scheme would be awkward because the numbers are floating point. You might consider converting the floating-point numbers to integers, and in this case that would be a valid design option, but for the sake of illustration, this example will stick with floating point.

To use the stair-step method, you put the upper end of each range into a table and then write a loop to check a score against the upper end of each range. When you find the point at which the score first exceeds the top of a range, you know what the grade is. With the stair-step technique, you have to be careful to handle the endpoints of the ranges properly. Here's the code in Visual Basic that assigns grades to a group of students based on this example:

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

18.5. Other Examples of Table Lookups

A few other examples of table lookups appear in other sections of the book. They're used in the course of discussing other techniques, and the contexts don't emphasize the table lookups per se. Here's where you'll find them:

- Looking up rates in an insurance table: [Section 16.3, "Creating Loops Easily—From the Inside Out"](#)
- Using decision tables to replace complicated logic: "Use decision tables to replace complicated conditions" in [Section 19.1](#).
- Cost of memory paging during a table lookup: [Section 25.3, "Kinds of Fat and Molasses"](#)
- Combinations of boolean values (A or B or C): "[Substitute Table Lookups for Complicated Expressions](#)" in [Section 26.1](#)
- Precomputing values in a loan repayment table: [Section 26.4, "Expressions."](#)

cc2e.com/1872

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?

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 by 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.

Chapter 19. General Control Issues

cc2e.com/1978

Contents

- [Boolean Expressions page 431](#)
- [Compound Statements \(Blocks\) page 443](#)
- [Null Statements page 444](#)
- [Taming Dangerously Deep Nesting page 445](#)
- [A Programming Foundation: Structured Programming page 454](#)
- [Control Structures and Complexity page 456](#)

Related Topics

- Straight-line code: [Chapter 14](#)
- Code with conditionals: [Chapter 15](#)
- Code with loops: [Chapter 16](#)
- Unusual control structures: [Chapter 17](#)
- Complexity in software development: "[Software's Primary Technical Imperative: Managing Complexity](#)" in [Section 5.2](#)

No discussion of control would be complete unless it went into several general issues that crop up when you think about control constructs. Most of the information in this chapter is detailed and pragmatic. If you're reading for the theory of control structures rather than for the gritty details, concentrate on the historical perspective on structured programming in [Section 19.5](#) and on the relationships between control structures in [Section 19.6](#).

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

19.1. Boolean Expressions

Except for the simplest control structure, the one that calls for the execution of statements in sequence, all control structures depend on the evaluation of boolean expressions.

Using true and false for Boolean Tests

Use the identifiers true and false in boolean expressions rather than using values 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:



Visual Basic Examples of Using Ambiguous Flags for Boolean Values

```
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.

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 previous code example is rewritten here using Microsoft Visual Basic's built-in True and False:

Good, but Not Great Visual Basic Examples of Using True and False for Tests Instead of Numeric Values

```
Dim printerError As Boolean
```

```
Dim reportSelected As ReportType
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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. Guidelines for using compound statements effectively follow:

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.

Cross-Reference

Many programmer-oriented text editors have commands that match braces, brackets, and parentheses. For details, see "[Editing](#)" in [Section 30.2](#).

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 the 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 statement after an if test is sometimes appealing aesthetically, but under maintenance such statements tend to become more complicated blocks, and single statements are errorprone when that happens.

Use blocks to clarify your intentions regardless of whether the code inside the block is 1 line or 20.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

19.3. Null Statements

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() )  
;  
;
```

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:

Cross-Reference

The best way to handle null statements is probably to avoid them. For details, see "Avoid empty loops" in [Section 16.2](#).

C++ Examples of a Null Statement That's Emphasized

```
while ( recordArray.Read( index++ ) ) != recordArray.EmptyRecord() ) {}           <-- 1  
  
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) {           <-- 2  
;  
}  
}
```

(1)This is one way to show the null statement.

(2)This is another way to show it.

Create a preprocessor DoNothing() 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++ by 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 DoNothing()

```
#define DoNothing()  
...  
  
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) {  
    DoNothing();
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

19.4. Taming Dangerously Deep Nesting



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, "Design in Construction,"](#) describes as Software's Primary Technical Imperative: Managing Complexity. That is reason enough to avoid deep nesting.



KEY POINT

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 refactor code into simpler routines. The following subsections present several ways to reduce the nesting depth:

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. This code example has nesting that's deep enough to warrant restructuring:



CODING HORROR

C++ Example of Bad, Deeply Nested Code

```
if ( inputStatus == InputStatus_Success ) {  
    // lots of code  
  
    ...  
  
    if ( printerRoutine != NULL ) {  
  
        // lots of code  
  
        ...  
  
        if ( SetupPage() ) {  
  
            // lots of code  
  
            ...  
  
            if ( AllocMem( &printData ) ) {  
  
                // lots of code  
  
                ...  
            }  
        }  
    }  
}
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 code that provides a less meaningful, less readable picture of how a program executes in the machine. Less readability means less understanding and, ultimately, lower program quality.

The central concepts of structured programming are still useful today and apply to considerations in using break, continue, throw, catch, return, and other topics.

The Three Components of Structured Programming

The next few sections describe the three constructs that constitute the core of structured programming.

Sequence

A sequence is a set of statements executed in order. Typical sequential statements include assignments and calls to routines. Here are two examples:

Cross-Reference

For details on using sequences, see [Chapter 14, "Organizing Straight-Line Code."](#)

Java Examples of Sequential Code

```
// a sequence of assignment statements

a = "1";
b = "2";
c = "3";

// a sequence of calls to routines

System.out.println( a );
System.out.println( b );
System.out.println( c );
```

Selection

A selection is a control structure that causes statements to be executed selectively. The if-then-else statement is a common example. Either the if-then clause or the else clause is executed, but not both. One of the clauses is "else if" form.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

19.6. Control Structures and Complexity

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.

Make things as simple as possible—but no simpler.

—Albert Einstein



KEY POINT 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.

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.

Cross-Reference

For more on complexity, see "[Software's Primary Technical Imperative: Managing Complexity](#)" in [Section 5.2](#).



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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- Making boolean expressions simple and readable contributes substantially to the quality of your code.
- Deep nesting makes a routine hard to understand. Fortunately, you can avoid it relatively easily.
- Structured programming is a simple idea that is still relevant: you can build any program out of a combination of sequences, selections, and iterations.
- Minimizing complexity is a key to writing high-quality code.

Part V: Code Improvements

[In this part:](#)

[Chapter 20. The Software-Quality Landscape](#)

[Chapter 21. Collaborative Construction](#)

[Chapter 22. Developer Testing](#)

[Chapter 23. Debugging](#)

[Chapter 24. Refactoring](#)

[Chapter 25. Code-Tuning Strategies](#)

[Chapter 26. Code-Tuning Techniques](#)

In this part:

[Chapter 20](#): The Software-Quality Landscape

[Chapter 21](#): Collaborative Construction

[Chapter 22](#): Developer Testing

[Chapter 23](#): Debugging

[Chapter 24](#): Refactoring

[Chapter 25](#): Code-Tuning Strategies

[Chapter 26](#): Code-Tuning Techniques

Chapter 20. The Software-Quality Landscape

cc2e.com/2036

Contents

- [Characteristics of Software Quality page 463](#)
- [Techniques for Improving Software Quality page 466](#)
- [Relative Effectiveness of Quality Techniques page 469](#)
- [When to Do Quality Assurance page 473](#)
- [The General Principle of Software Quality page 474](#)

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 from a construction point of view. The entire 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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 the following:

-
- Correctness The degree to which a system is free from faults in its specification, design, and implementation.
-
- Usability The ease with which users can learn and use a system.
-
- Efficiency Minimal use of system resources, including memory and execution time.
-
- Reliability The ability of a system to perform its required functions under stated conditions whenever required—having a long mean time between failures.
-
- Integrity The degree to which a system prevents unauthorized or improper access to its programs and its data. The idea of integrity includes restricting unauthorized user accesses as well as ensuring that data is accessed properly—that is, that tables with parallel data are modified in parallel, that date fields contain only valid dates, and so on.
-
- Adaptability The extent to which a system can be used, without modification, in applications or environments other than those for which it was specifically designed.
-
- Accuracy The degree to which a system, as built, is free from error, especially with respect to quantitative outputs. Accuracy differs from correctness; it is a determination of how well a system does the job it's built for rather than whether it was built correctly.
-
- Robustness The degree to which a system continues to function in the presence of invalid inputs or stressful environmental conditions.

Some of these characteristics overlap, but all have different shades of meaning that are applicable more in some cases, less in others.

External characteristics of quality are the only kind of software characteristics that users care about. Users care about whether the software is easy to use, not about whether it's easy for you to modify. They care about whether the software works correctly, not about whether the code is readable or well structured.

Programmers care about the internal characteristics of the software as well as the external ones. This book is code-centered, so it focuses on the internal quality characteristics, including

-
- Maintainability The ease with which you can modify a software system to change or add capabilities, improve performance, or correct defects.
-
- Flexibility The extent to which you can modify a system for uses or environments other than those for which it was specifically designed.
-

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 you also need to focus on the software-development process. Some of the elements of a software-quality program are described in the following subsections:

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 previous section. Without explicit goals, programmers might work to maximize characteristics different from the ones you expect them to maximize. The power of setting explicit goals is discussed in more detail later in this section.

Explicit quality-assurance activity One common problem in assuring quality is that quality is perceived as a secondary goal. Indeed, in some organizations, quick and dirty programming is the rule rather than the exception. Programmers like Global Gary, who litter their code with defects and "complete" their programs quickly, are rewarded more than programmers like High-Quality Henry, who write excellent programs and make sure that they are usable before releasing them. In such organizations, it shouldn't be surprising that programmers don't make quality their first priority. The organization must show programmers that quality is a priority. Making the quality-assurance activity explicit makes the priority clear, and programmers will respond accordingly.

Testing strategy Execution testing can provide a detailed assessment of a product's reliability. Part of quality assurance is developing a test strategy in conjunction with the product requirements, architecture, and design. Developers on many projects rely on testing as the primary method of both quality assessment and quality improvement. The rest of this chapter demonstrates in more detail that this is too heavy a burden for testing to bear by itself.

Cross-Reference

For details on testing, see [Chapter 22, "Developer Testing."](#)

Software-engineering guidelines Guidelines should control the technical character of the software as it's developed. Such guidelines apply to all software development activities, including problem definition, requirements development, architecture, construction, and system testing. The guidelines in this book are, in one sense, a set of software-engineering guidelines for construction.

Cross-Reference

For a discussion of one class of software-engineering guidelines appropriate for construction, see [Section 4.2, "Programming Conventions."](#)

Informal technical reviews Many software developers review their work before turning it over for formal review. Informal reviews include desk-checking the design or the code or walking through the code with a few peers.

Formal technical reviews One part of managing a software-engineering process is catching problems at the "lowest-value" stage—that is, at the time at which the least investment has been made and at which problems cost the least to correct. To achieve such a goal, developers use "quality gates," periodic tests or reviews that determine whether the quality of the product at one stage is sufficient to support moving on to the next. Quality gates are usually used to transition between requirements development and architecture, architecture and construction, and construction and system testing. The "gate" can be an inspection, a peer review, a customer review, or an audit.

Cross-Reference

Reviews and inspections are discussed in [Chapter 21, "Collaborative Construction."](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.

—Gerald Weinberg

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 detect out of the total defects that exist at that point in the project. [Table 20-2](#) shows the percentages of defects detected by several common defect-detection techniques.

Table 20-2. 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%
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%

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

20.4. When to Do Quality Assurance

As [Chapter 3 \("Measure Twice, Cut Once: Upstream Prerequisites"\)](#) noted, the earlier an error is inserted into software, the more entangled 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. Or a requirements error can result in architecture, code, and test cases that are thrown away. Just as it's a good idea to work out the defects in the blue-prints 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.

Cross-Reference

Quality assurance of upstream activities—requirements and architecture, for instance—is outside the scope of this book. The "[Additional Resources](#)" section at the end of the chapter describes books you can turn to for more information about them.

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.

**KEY POINT**

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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

20.5. The General Principle of Software Quality



KEY POINT

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.

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 arises 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 noncoding 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?

Part of the reason for these seemingly low productivity figures is that industry average numbers like these factor nonprogrammer time into the lines-of-code-per-day figure. Tester time, project manager time, and administrative support time are all included. Noncoding activities, such as 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.

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](#).

The single biggest activity on most projects is debugging and correcting code that doesn't work properly. Debugging and associated refactoring and other rework consume about 50 percent of the time on a traditional, naive software-development cycle. (See [Section 3.1, "Importance of Prerequisites,"](#) for more details.) Reducing debugging by preventing errors improves productivity. Therefore, the most obvious method of shortening a development schedule is to improve the quality of the product and decrease the amount of time spent debugging and reworking the software.



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 did not increase overall development cost (Card 1987).

A study at IBM produced similar findings:

Software projects with the lowest levels of defects had the shortest development schedules and the highest development productivity.... software defect removal is actually the most expensive and time-consuming form of work for software (Jones 2000).



The same effect holds true at the small end of the 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.

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/2050

It's not hard to list books in this section because virtually any book on effective software methodologies describes techniques that result in improved quality and productivity. The difficulty is finding books that deal with software quality per se. Here are two:

Ginac, Frank P. Customer Oriented Software Quality Assurance. Englewood Cliffs, NJ: Prentice Hall, 1998. This is a very short book that describes quality attributes, quality metrics, QA programs, and the role of testing in quality, as well as well-known quality improvement programs, including the Software Engineering Institute's CMM and ISO 9000.

Lewis, William E. Software Testing and Continuous Quality Improvement, 2d ed. Auerbach Publishing, 2000. This book provides a comprehensive discussion of a quality life cycle, as well as extensive discussion of testing techniques. It also provides numerous forms and checklists.

Relevant Standards

cc2e.com/2057

IEEE Std 730-2002, IEEE Standard for Software Quality Assurance Plans.

IEEE Std 1061-1998, IEEE Standard for a Software Quality Metrics Methodology.

IEEE Std 1028-1997, Standard for Software Reviews.

IEEE Std 1008-1987 (R1993), Standard for Software Unit Testing.

IEEE Std 829-1998, Standard for Software Test Documentation.

Key Points

- Quality is free, in the end, but it requires a reallocation of resources so that defects are prevented cheaply instead of fixed expensively.
- Not all quality-assurance goals are simultaneously achievable. Explicitly decide which goals you want to achieve, and communicate the goals to other people on your team.
- No single defect-detection technique is completely effective by itself. Testing by itself is not optimally effective at removing errors. Successful quality-assurance programs use several different techniques to detect different kinds of errors.
- You can apply effective techniques during construction and many equally powerful techniques before construction. The earlier you find a defect, the less intertwined it will become with the rest of your code and the less damage it will cause.
- Quality assurance in the software arena is process-oriented. Software development doesn't have a repetitive phase that affects the final product like manufacturing does, so the quality of the result is controlled by the process used to develop the software.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 21. Collaborative Construction

cc2e.com/2185

Contents

- [Overview of Collaborative Development Practices page 480](#)
- [Pair Programming page 483](#)
- [Formal Inspections page 485](#)
- [Other Kinds of Collaborative Development Practices page 492](#)

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.

Table 21-1. Comparison of Collaborative Construction Techniques

Property	Pair Programming	Formal Inspection	Informal Review (Walk-Throughs)
Defined participant roles	Yes	Yes	No

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

21.1. Overview of Collaborative Development Practices

["Collaborative construction"](#) refers to pair programming, formal inspections, informal technical reviews, and document reading, as well as other techniques in which developers share responsibility for creating code and other work products. At my company, the term "collaborative construction" was coined by Matt Peloquin in about 2000. The term appears to have been coined independently by others in the same time frame.



All collaborative construction techniques, despite their differences, are based on the ideas that developers are blind to some of the trouble spots in their work, that other people don't have the same blind spots, and that it's beneficial for developers to have someone else look at their work. Studies at the Software Engineering Institute have found that developers insert an average of 1 to 3 defects per hour into their designs and 5 to 8 defects per hour into code (Humphrey 1997), so attacking these blind spots is a key to effective construction.

Collaborative Construction Complements Other Quality-Assurance Techniques

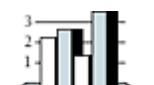


The primary purpose of collaborative construction is to improve software quality. As noted in [Chapter 20, "The Software-Quality Landscape,"](#) software testing has limited effectiveness when used alone—the average defect-detection rate is only about 30 percent for unit testing, 35 percent for integration testing, and 35 percent for low-volume beta testing. In contrast, the average effectivenesses of design and code inspections are 55 and 60 percent (Jones 1996). The secondary benefit of collaborative construction is that it decreases development time, which in turn lowers development costs.



Early reports on pair programming suggest that it can achieve a code-quality level similar to formal inspections (Shull et al 2002). The cost of full-up pair programming is probably higher than the cost of solo development—on the order of 10–25 percent higher—but the reduction in development time appears to be on the order of 45 percent, which in some cases may be a decisive advantage over solo development (Boehm and Turner 2004), although not over inspections which have produced similar results.

Technical reviews have been studied much longer than pair programming, and their results, as described in case studies and elsewhere, have been impressive:

- 

HARD DATA IBM found that each hour of inspection prevented about 100 hours of related work (testing and defect correction) (Holland 1999).
 - Raytheon reduced its cost of defect correction (rework) from about 40 percent of total project cost to about 20 percent through an initiative that focused on inspections (Haley 1996).
 - Hewlett-Packard reported that its inspection program saved an estimated \$21.5 million per year (Grady and Van Slack 1994).
 - Imperial Chemical Industries found that the cost of maintaining a portfolio of about 400 programs was only about 10 percent as high as the cost of maintaining a similar set of programs that had not been inspected (Gillb

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

21.2. Pair Programming

When pair programming, one programmer types in code at the keyboard and the other programmer watches for mistakes and thinks strategically about whether the code is being written correctly and whether the right code is being written. Pair programming was originally popularized by Extreme Programming (Beck 2000), but it is now being used more widely (Williams and Kessler 2002).

Keys to Success with Pair Programming

The basic concept of pair programming is simple, but its use nonetheless benefits from a few guidelines:

Support pair programming with coding standards Pair programming will not be effective if the two people in the pair spend their time arguing about coding style. Try to standardize what [Chapter 5, "Design in Construction,"](#) refers to as the "accidental attributes" of programming so that the programmers can focus on the "essential" task at hand.

Don't let pair programming turn into watching The person without the keyboard should be an active participant in the programming. That person is analyzing the code, thinking ahead to what will be coded next, evaluating the design, and planning how to test the code.

Don't force pair programming of the easy stuff One group that used pair programming for the most complicated code found it more expedient to do detailed design at the whiteboard for 15 minutes and then to program solo (Manzo 2002). Most organizations that have tried pair programming eventually settle into using pairs for part of their work but not all of it (Boehm and Turner 2004).

Rotate pairs and work assignments regularly In pair programming, as with other collaborative development practices, benefit arises from different programmers learning different parts of the system. Rotate pair assignments regularly to encourage cross-pollination—some experts recommend changing pairs as often as daily (Reifer 2002).

Encourage pairs to match each other's pace One partner going too fast limits the benefit of having the other partner. The faster partner needs to slow down, or the pair should be broken up and reconfigured with different partners.

Make sure both partners can see the monitor Even seemingly mundane issues like being able to see the monitor and using fonts that are too small can cause problems.

Don't force people who don't like each other to pair Sometimes personality conflicts prevent people from pairing effectively. It's pointless to force people who don't get along to pair, so be sensitive to personality matches (Beck 2000, Reifer 2002).

Avoid pairing all newbies Pair programming works best when at least one of the partners has paired before (Larman 2004).

Assign a team leader If your whole team wants to do 100 percent of its programming in pairs, you'll still need to assign one person to coordinate work assignments, be held accountable for results, and act as the point of contact for people outside the project.

Benefits of Pair Programming

Pair programming produces numerous benefits:

- It holds up better under stress than solo development. Pairs encourage each other to keep code quality high even when there's pressure to write quick and dirty code.
-

It is a good idea. The ability to detect and fix mistakes quickly is critical to the success of the project.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

21.3. Formal Inspections

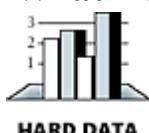
Further Reading

If you want to read the original article on inspections, see "Design and Code Inspections to Reduce Errors in Program Development" (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:

- Checklists focus the reviewers' attention on areas that have been problems in the past.
- The inspection focuses on defect detection, not correction.
- Reviewers prepare for the inspection meeting beforehand and arrive with a list of the problems they've discovered.
- Distinct roles are assigned to all participants.
- The moderator of the inspection isn't the author of the work product under inspection.
- The moderator has received specific training in moderating inspections.
- The inspection meeting is held only if all participants have adequately prepared.
- Data is collected at each inspection and is fed into future inspections to improve them.
- General management doesn't attend the inspection meeting unless you're inspecting a project plan or other management materials. Technical leaders might attend.

What Results Can You Expect from Inspections?



Individual inspections typically catch about 60 percent of defects, which is higher than other techniques except prototyping and high-volume beta testing. These results have been confirmed numerous times at various organizations, including Harris BCSD, National Software Quality Experiment, Software Engineering Institute, Hewlett Packard, and so on (Shull et al 2002).

The combination of design and code inspections usually removes 70–85 percent or more of the defects in a product (Jones 1996). Inspections identify error-prone classes early, and Capers Jones reports that they result in 20–30 percent fewer defects per 1000 lines of code than less formal review practices. Designers and coders learn to improve their work through participating in inspections, and inspections increase productivity by about 20 percent

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

21.4. Other Kinds of Collaborative Development Practices

Other kinds of collaboration haven't accumulated the body of empirical support that inspections or pair programming have, so they're covered in less depth here. The collaborations covered in this section includes walk-throughs, code reading, and dog-and-pony shows.

Walk-Throughs

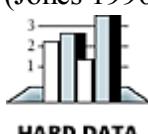
A walk-through is a popular kind of review. The term is loosely defined, and at least some of its popularity can be attributed to the fact that people can call virtually any kind of review a "walk-through."

Because the term is so loosely defined, it's hard to say exactly what a walk-through is. Certainly, a walk-through involves two or more people discussing a design or code. It might be as informal as an impromptu bull session around a whiteboard; it might be as formal as a scheduled meeting with an overhead presentation prepared by the art department and a formal summary sent to management. In one sense, "where two or three are gathered together," there is a walk-through. Proponents of walk-throughs like the looseness of such a definition, so I'll just point out a few things that all walk-throughs have in common and leave the rest of the details to you:

- The walk-through is usually hosted and moderated by the author of the design or code under review.
-
- The walk-through focuses on technical issues—it's a working meeting.
-
- All participants prepare for the walk-through by reading the design or code and looking for errors.
-
- The walk-through 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.
-
- A walk-through usually lasts 30 to 60 minutes.
-
- The emphasis is on error detection, not correction.
-
- Management doesn't attend.
-
- The walk-through concept is flexible and can be adapted to the specific needs of the organization using it.

What Results Can You Expect from a Walk-Through?

Used intelligently and with discipline, a walk-through can produce results similar to those of an inspection—that is, it can typically find between 20 and 40 percent of the errors in a program (Myers 1979, Boehm 1987b, Yourdon 1989b, Jones 1996). But in general, walk-throughs have been found to be significantly less effective than inspections (Jones 1996).



Used unintelligently, walk-throughs are more trouble than they're worth. The low end of their effectiveness, 20 percent, isn't worth much, and at least one organization (Boeing Computer Services) found peer reviews of code to be "extremely expensive." Boeing found it was difficult to motivate project personnel to apply walk-through techniques consistently, and

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Comparison of Collaborative Construction Techniques

What are the differences among the various kinds of collaborative construction? [Table 21-1](#) provides a summary of each technique's major characteristics.

Pair programming doesn't have decades of data supporting its effectiveness like formal inspection does, but the initial data suggests it's on roughly equal footing with inspections, and anecdotal reports have also been positive.

If pair programming and formal inspections produce similar results for quality, cost, and schedule, the choice between them becomes a matter of personal style rather than one 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 be allowed to choose which way they would like to do most of their work. You should also use different techniques with a project, as appropriate.

Additional Resources

cc2e.com/2106

Here are more resources concerning collaborative construction:

Pair Programming

Williams, Laurie and Robert Kessler. *Pair Programming Illuminated*. Boston, MA: Addison Wesley, 2002. This book explains the detailed ins and outs of pair programming, including how to handle various personality matches (for example, expert and inexpert, introvert and extrovert) and other implementation issues.

Beck, Kent. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison Wesley, 2000. This book touches on pair programming briefly and shows how it can be used in conjunction with other mutually supportive techniques, including coding standards, frequent integration, and regression testing.

Reifer, Donald. "How to Get the Most Out of Extreme Programming/Agile Methods," *Proceedings, XP/Agile Universe 2002*. New York, NY: Springer; pp. 185–196. This paper summarizes industrial experience with Extreme Programming and agile methods and presents keys to success for pair programming.

Inspections

Wieggers, Karl. *Peer Reviews in Software: A Practical Guide*. Boston, MA: Addison Wesley, 2002. This well-written book describes the ins and outs of various kinds of reviews, including formal inspections and other, less formal practices. It's well researched, has a practical focus, and is easy to read.

Gilb, Tom and Dorothy Graham. *Software Inspection*. Wokingham, England: Addison-Wesley, 1993. This contains a thorough discussion of inspections circa the early 1990s. It has a practical focus and includes case studies that describe experiences several organizations have had in setting up inspection programs.

Fagan, Michael E. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal* 15, no. 3 (1976): 182–211.

Fagan, Michael E. "Advances in Software Inspections." *IEEE Transactions on Software Engineering*, SE-12, no. 7 (July 1986): 744–51. These two articles were written by the developer of inspections. They contain the meat of what you need to know to run an inspection, including all the standard inspection forms.

Relevant Standards

IEEE Std 1028-1997, Standard for Software Reviews

IEEE Std 730-2002, Standard for Software Quality Assurance Plans

Key Points

- Collaborative development practices tend to find a higher percentage of defects than testing and to find them more efficiently.
- Collaborative development practices tend to find different kinds of errors than testing does, implying that you need to use both reviews and testing to ensure the quality of your software.
- Formal inspections use checklists, preparation, well-defined roles, and continual process improvement to maximize error-detection efficiency. They tend to find more defects than walk-throughs.
- Pair programming typically costs about the same as inspections and produces similar quality code. Pair programming is especially valuable when schedule reduction is desired. Some developers prefer working in pairs to working solo.
- Formal inspections can be used on work products such as requirements, designs, and test cases, as well as on code.
- Walk-throughs and code reading are alternatives to inspections. Code reading offers more flexibility in using each person's time effectively.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 22. Developer Testing

cc2e.com/2261

Contents

- [Role of Developer Testing in Software Quality page 500](#)
- [Recommended Approach to Developer Testing page 503](#)
- [Bag of Testing Tricks page 505](#)
- [Typical Errors page 517](#)
- [Test-Support Tools page 523](#)
- [Improving Your Testing page 528](#)
- [Keeping Test Records page 529](#)

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 percent of the errors present each. The combination of testing steps often finds less than 60 percent of the errors present (Jones 1998).

Cross-Reference

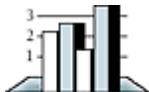
For details on reviews, see [Chapter 21, "Collaborative Construction."](#)

Programs do not acquire bugs as people acquire germs, by hanging around other buggy programs. Programmers must insert them.

—Harlan Mills

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.



HARD DATA 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 carefully enough. The errors were visible, but the programmers didn't notice them (Myers 1978).

You must hope to find errors in your code. Such a hope might seem like an unnatural act, but you should hope that it's you who finds the errors and not someone else.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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.
- Use a checklist of the kinds of errors you've made on the project to date or have made on previous projects.

Design 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. Plan to test and find defects as early as possible because it's cheaper to fix defects early.

Test First or Test Last?

Developers sometimes wonder whether it's better to write test cases after the code has been written or beforehand (Beck 2003). The defect-cost increase graph—see [Figure 3-1](#) on page 30—suggests that writing test cases first will minimize the amount of time between when a defect is inserted into the code and when the defect is detected and removed. This turns out to be one of many reasons to write test cases first:

- Writing test cases before writing the code doesn't take any more effort than writing test cases after the code; it simply resequences the test-case-writing activity.
- When you write test cases first, you detect defects earlier and you can correct them more easily.
- Writing test cases first forces you to think at least a little bit about the requirements and design before writing code, which tends to produce better code.
- Writing test cases first exposes requirements problems sooner, before the code is written, because it's hard to write a test case for a poor requirement.
- If you save your test cases, which you should do, you can still test last, in addition to testing first.

All in all, I think test-first programming is one of the most beneficial software practices to emerge during the past

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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	2620 (20 characters, each with 26 possible choices)
Address	2620 (20 characters, each with 26 possible choices)
Phone Number	1010 (10 digits, each with 10 possible choices)
Total Possibilities	$= 2620 * 2620 * 1010 \approx 1066$

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 percent of the way done today. Obviously, if you added a 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 1066 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.

Cross-Reference

One way of telling whether you've covered all the code is to use a coverage monitor. For details, see "[Coverage Monitors](#)" in [Section 22.5, "Test-Support Tools,"](#) later in this chapter.

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 in the following sections.

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—an if or a while, for example—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 a rigorous analysis of the logic of the program. If you can't analyze the logic of the program, then you can't know if you've covered all the bases.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

22.4. Typical Errors

This section is dedicated to the proposition that you can test best when you know as much as possible about your enemy: errors.

Which Classes Contain the Most Errors?

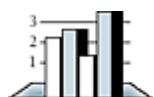


It's natural to assume that defects are distributed evenly throughout your source code. If you have an average of 10 defects per 1000 lines of code, you might assume that you'll have one defect in a class that contains 100 lines of code. This is a natural assumption, but it's wrong.

KEY POINT Capers Jones reported that a focused quality-improvement program at IBM identified 31 of 425 classes in the IMS system as error-prone. The 31 classes were repaired or completely redeveloped, and, in less than a year, customer-reported defects against IMS were reduced ten to one. Total maintenance costs were reduced by about 45 percent. Customer satisfaction improved from "unacceptable" to "good" (Jones 2000).

Most errors tend to be concentrated in a few highly defective routines. Here is the general relationship between errors and code:

-

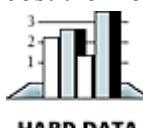


HARD DATA Eighty percent of the errors are found in 20 percent of a project's classes or routines (Endres 1975, Gremillion 1984, Boehm 1987b, Shull et al 2002).

-

Fifty percent of the errors are found in 5 percent of a project's classes (Jones 2000).

These relationships might not seem so important until you recognize a few corollaries. First, 20% of a project's routines contribute 80% of the cost of development (Boehm 1987b). That doesn't necessarily mean that the 20% that cost the most are the same as the 20% with the most defects, but it's pretty suggestive.



HARD DATA 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 an analysis 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 percent 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: improving quality improves the development schedule and reduces development costs.

Cross-Reference

Another class of routines that tend to contain a lot of errors is the class of overly complex routines. For details on identifying and simplifying routines, see "[General Guidelines for Reducing Complexity](#)" in [Section 19.6](#).

Fourth, the implication of avoiding troublesome routines for maintenance is equally clear. Maintenance activities should

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

22.5. Test-Support Tools

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.

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.

Further Reading

For several good examples of scaffolding, see Jon Bentley's essay "A Small Matter of Programming" in *Programming Pearls*, 2d ed. (2000).

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. In these cases, the scaffolding 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." This scaffolding 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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 sections 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.

Cross-Reference

Part of planning to test is formalizing your plans in writing. To find further information on test documentation, refer to the "[Additional Resources](#)" section at the end of [Chapter 32](#).

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 backward, or "regressed," is called "regression testing."

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



KEY POINT

The only practical way to manage regression testing is to automate it. People become numb 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).

Benefits of test automation include the following:

- An automated test has a lower chance of being wrong than a manual test.
- Once you automate a test, it's readily available for the rest of the project with little incremental effort on your part.
- If tests are automated, they can be run frequently to see whether any code checkins have broken the code. Test automation is part of the foundation of test-intensive practices, such as the daily build and smoke test and Extreme Programming.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

22.7. Keeping Test Records

Aside from making the testing process repeatable, you need to measure the project so that you can tell for sure whether changes improve or degrade it. Here are a few kinds of data you can collect to measure your project:

- Administrative description of the defect (the date reported, the person who reported it, a title or description, the build number, the date fixed)
- Full description of the problem
- Steps to take to repeat the problem
- Suggested workaround for the problem
- Related defects
- Severity of the problem—for example, fatal, bothersome, or cosmetic
- Origin of the defect: requirements, design, coding, or testing
- Subclassification of a coding defect: off-by-one, bad assignment, bad array index, bad routine call, and so on
- Classes and routines changed by the fix
- Number of lines of code affected by the defect
- Hours to find the defect
- Hours to fix the defect

Once you collect the data, you can crunch a few numbers to determine whether your project is getting sicker or healthier:

- Number of defects in each class, sorted from worst class to best, possibly normalized by class size
- Number of defects in each routine, sorted from worst routine to best, possibly normalized by routine size
- Average number of testing hours per defect found

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/2203

Federal truth-in-advising statutes compel me to disclose that several other books cover testing in more depth than this chapter does. Books that are devoted to testing discuss system and black-box testing, which haven't been discussed in this chapter. They also go into more depth on developer topics. They discuss formal approaches such as cause-effect graphing and the ins and outs of establishing an independent test organization.

Testing

Kaner, Cem, Jack Falk, and Hung Q. Nguyen. *Testing Computer Software*, 2d ed. New York, NY: John Wiley & Sons, 1999. This is probably the best current book on software testing. It is most applicable to testing applications that will be distributed to a widespread customer base, such as high-volume websites and shrink-wrap applications, but it is also generally useful.

Kaner, Cem, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing*. New York, NY: John Wiley & Sons, 2002. This book is a good supplement to *Testing Computer Software*, 2d ed. It's organized into 11 chapters that enumerate 250 lessons learned by the authors.

Tamre, Louise. *Introducing Software Testing*. Boston, MA: Addison-Wesley, 2002. This is an accessible testing book targeted at developers who need to understand testing. Belying the title, the book goes into some depth on testing details that are useful even to experienced testers.

Whittaker, James A. *How to Break Software: A Practical Guide to Testing*. Boston, MA: Addison-Wesley, 2002. This book lists 23 attacks testers can use to make software fail and presents examples for each attack using popular software packages. You can use this book as a primary source of information about testing or, because its approach is distinctive, you can use it to supplement other testing books.

Whittaker, James A. "What Is Software Testing? And Why Is It So Hard?" *IEEE Software*, January 2000, pp. 70–79. This article is a good introduction to software testing issues and explains some of the challenges associated with effectively testing software.

Myers, Glenford J. *The Art of Software Testing*. New York, NY: John Wiley, 1979. This is the classic book on software testing and is still in print (though quite expensive). The contents of the book are straightforward: A Self-Assessment Test; The Psychology and Economics of Program Testing; Program Inspections, Walkthroughs, and Reviews; Test-Case Design; Class Testing; Higher-Order Testing; Debugging; Test Tools and Other Techniques. It's short (177 pages) and readable. The quiz at the beginning gets you started thinking like a tester and demonstrates how many ways a piece of code can be broken.

Test Scaffolding

Bentley, Jon. "A Small Matter of Programming" in *Programming Pearls*, 2d ed. Boston, MA: Addison-Wesley, 2000. This essay includes several good examples of test scaffolding.

Mackinnon, Tim, Steve Freeman, and Philip Craig. "Endo-Testing: Unit Testing with Mock Objects," *eXtreme Programming and Flexible Processes Software Engineering - XP2000* Conference, 2000. This is the original paper to discuss the use of mock objects to support developer testing.

Thomas, Dave and Andy Hunt. "Mock Objects," *IEEE Software*, May/June 2002. This is a highly readable introduction to using mock objects to support developer testing.

cc2e.com/2217

<http://www.junit.org>. This site provides support for developers using JUnit. Similar resources are provided at coco-junit.sourceforge.net and xunit.sourceforge.net.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- 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.
- 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.
- 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 these practices detect different kinds of errors.
- You can generate many test cases deterministically by using basis testing, dataflow analysis, boundary analysis, classes of bad data, and classes of good data. You can generate additional test cases with error guessing.
- Errors tend to cluster in a few error-prone classes and routines. Find that errorprone code, redesign it, and rewrite it.
- 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, testdata errors are more aggravating than programming errors. Avoid them by developing your tests as carefully as your code.
- Automated testing is useful in general and is essential for regression testing.
- 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.

Chapter 23. Debugging

cc2e.com/2361

Contents

- [Overview of Debugging Issues page 535](#)
- [Finding a Defect page 540](#)
- [Fixing a Defect page 550](#)
- [Psychological Considerations in Debugging page 554](#)
- [Debugging Tools—Obvious and Not-So-Obvious page 556](#)

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 is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

—Brian W. Kernighan

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'll 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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

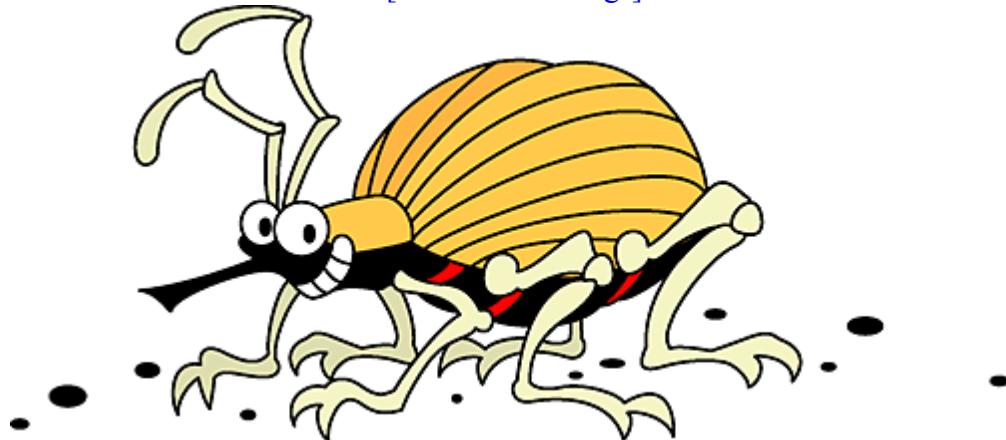
[NEXT ▶](#)

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 "bug" dates back at least to Thomas Edison, who is quoted as using it as early as 1878 (Tenner 1997).

The word "bug" is a cute word and conjures up images like this one:

[\[View full size image\]](#)



The reality of software defects, however, is that bugs aren't organisms that sneak into your code when you forget to spray it with pesticide. They are errors. A bug in software means that a programmer made a mistake. The result of the mistake isn't like the cute picture shown above. It's more likely a note like this one:



[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

23.2. Finding a Defect

Debugging consists of finding the defect and fixing it. Finding the defect—and understanding it—is usually 90 percent of the work.

Fortunately, you don't have to make a pact with Satan to find an approach to debugging that's better than random guessing. Debugging by thinking about the problem is much more effective and interesting than debugging with an eye of a newt and the dust of a frog's ear.

Suppose you were asked to solve a murder mystery. Which would be more interesting: going door to door throughout the county, checking every person's alibi for the night of October 17, or finding a few clues and deducing the murderer's identity? Most people would rather deduce the person's identity, and most programmers find the intellectual approach to debugging more satisfying. Even better, the effective programmers who debug in one-twentieth the time used by the ineffective programmers aren't randomly guessing about how to fix the program. They're using the scientific method—that is, the process of discovery and demonstration necessary for scientific investigation.

The Scientific Method of Debugging

Here are the steps you go through when you use the classic scientific method:

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.



The scientific method has many parallels in debugging. Here's an effective approach for finding a defect:

KEY POINT

1. Stabilize the error.
2. Locate the source of the error (the "fault").
 - a. Gather the data that produces the defect.
 - b. Analyze the data that has been gathered, and form a hypothesis about the defect.
 - c. Determine how to prove or disprove the hypothesis, either by testing the program or by examining the code.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

23.3. Fixing a Defect

The hard part of debugging 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 KEY POINT "[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 one of several causes, you don't yet have enough evidence to work on the one cause; rule out the others first.

Relax A programmer was ready for a ski trip. His product was ready to ship, he was already late, and he had only one more defect to correct. He changed the source file and checked it into version control. He didn't recompile the program and didn't verify that the change was correct.

Never debug standing up.

—Gerald Weinberg

In fact, the change was not correct, and his manager was outraged. How could he change code in a product that was ready to ship without checking it? What could be worse? Isn't this the pinnacle of professional recklessness?

If this isn't the height of recklessness, it's close and it's common. Hurrying to solve a problem is one of the most time-ineffective things you can do. It leads to rushed judgments, incomplete defect diagnosis, and incomplete corrections. Wishful thinking can lead you to see solutions where there are none. The pressure—often self-imposed—encourages haphazard trial-and-error solutions and the assumption that a solution works without verification that it does.

In striking contrast, during the final days of Microsoft Windows 2000 development, a developer needed to fix a defect that was the last remaining defect before a Release Candidate could be created. The developer changed the code, checked his fix, and tested his fix on his local build. But he didn't check the fix into version control at that point. Instead, he went to play basketball. He said, "I'm feeling too stressed right now to be sure that I've considered everything I should consider. I'm going to clear my mind for an hour, and then I'll come back and check in the code—once I've convinced myself that the fix is really correct."

Relax long enough to make sure your solution is right. Don't be tempted to take shortcuts. It might take more time, but it'll probably take less. If nothing else, you'll fix the problem correctly and your manager won't call you back from your ski trip.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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've seen that it has one. You have to think precisely—forming 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.

Further Reading

For an excellent discussion of psychological issues in debugging, as well as many other areas of software development, see *The Psychology of Computer Programming* (Weinberg 1998).

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?



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;
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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.

Cross-Reference

The line between testing and debugging tools is fuzzy. See [Section 22.5](#) for more on testing tools and [Chapter 30](#) for more on software-development tools.

Source-Code Comparators

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 what changed.

Compiler Warning Messages



One of the simplest and most effective debugging tools is your own compiler.

KEY POINT

Set your compiler's warning level to the highest, pickiest level possible, and fix the errors it reports 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.

Initiate projectwide standards for compile-time settings Set a standard that requires everyone on your team to compile code using the same compiler settings. Otherwise, when you try to integrate code compiled by different people with different settings, you'll get a flood of error messages and an integration nightmare. This is easy to enforce if you use a project-standard make file or build script.

Extended Syntax and Logic Checking

You can use additional tools to check your code more thoroughly than your compiler does. For example, for C programmers, the lint utility painstakingly checks for use of uninitialized variables (writing = when you mean ==) and similarly subtle problems.

Execution Profilers

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/2375

The following resources also address debugging:

Agans, David J. Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems. Amacom, 2003. This book provides general debugging principles that can be applied in any language or environment.

Myers, Glenford J. The Art of Software Testing. New York, NY: John Wiley & Sons, 1979. [Chapter 7](#) of this classic book is devoted to debugging.

Allen, Eric. Bug Patterns In Java. Berkeley, CA: Apress, 2002. This book lays out an approach to debugging Java programs that is conceptually very similar to what is described in this chapter, including "[The Scientific Method of Debugging](#)," distinguishing between debugging and testing, and identifying common bug patterns.

The following two books are similar in that their titles suggest they are applicable only to Microsoft Windows and .NET programs, but they both contain discussions of debugging in general, use of assertions, and coding practices that help to avoid bugs in the first place:

Robbins, John. Debugging Applications for Microsoft .NET and Microsoft Windows. Redmond, WA: Microsoft Press, 2003.

McKay, Everett N. and Mike Woodring. Debugging Windows Programs: Strategies, Tools, and Techniques for Visual C++ Programmers. Boston, MA: Addison-Wesley, 2000.

Key Points

- Debugging is a make-or-break aspect of software development. The best approach is to use other techniques described in this book to avoid defects in the first place. It's still worth your time to improve your debugging skills, however, because the difference between good and poor debugging performance is at least 10 to 1.
- A systematic approach to finding and fixing errors is critical to success. Focus your debugging so that each test moves you a step forward. Use the Scientific Method of Debugging.
- Understand the root problem before you fix the program. Random guesses about the sources of errors and random corrections will leave the program in worse condition than when you started.
- Set your compiler warning to the pickiest level possible, and fix the errors it reports. It's hard to fix subtle errors if you ignore the obvious ones.
- Debugging tools are powerful aids to software development. Find them and use them, and remember to use your brain at the same time.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 24. Refactoring

cc2e.com/2436

Contents

- [Kinds of Software Evolution page 564](#)
- [Introduction to Refactoring page 565](#)
- [Specific Refactorings page 571](#)
- [Refactoring Safely page 579](#)
- [Refactoring Strategies page 582](#)

Related Topics

- Tips for fixing defects: [Section 23.3](#)
- Code-tuning approach: [Section 25.6](#)
- Design in construction: [Chapter 5](#)
- Working 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.

All successful software gets changed.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

24.1. Kinds of Software Evolution

Software evolution is like biological evolution in that some mutations are beneficial and many mutations are not. Good software evolution produces code whose development mimics the ascent from monkeys to Neanderthals to our current exalted state as software developers. Evolutionary forces sometimes beat on a program the other way, however, knocking the program into a deevolutionary spiral.



KEY POINT The key distinction between kinds of software evolution is whether the program's quality improves or degrades under modification. If you fix errors with logical duct tape and superstition, quality degrades. If you treat modifications as opportunities to tighten up the original design of the program, quality improves. If you see that program quality is degrading, that's like that silent canary in a mine shaft I've mentioned before. It's a warning that the program is evolving in the wrong direction.

A second distinction in the kinds of software evolution is the one between changes made during construction and those made during maintenance. These two kinds of evolution differ in several ways. Construction changes are usually made by the original developers, usually before the program has been completely forgotten. The system isn't yet on line, so the pressure to finish changes is only schedule pressure—it's not 500 angry users wondering why their system is down. For the same reason, changes during construction can be more freewheeling—the system is in a more dynamic state, and the penalty for making mistakes is low. These circumstances imply a style of software evolution that's different from what you'd find during software maintenance.

Philosophy of Software Evolution

A common weakness in programmers' approaches to software evolution is that it goes on as an unselfconscious 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.

There is no code so big, twisted, or complex that maintenance can't make it worse.

—Gerald Weinberg

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 further change in mind.



KEY POINT 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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

24.2. Introduction to Refactoring

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).

Reasons to Refactor

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:

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—whenever you make changes in one place, you have to make parallel changes in another place. It also violates what Andrew Hunt and Dave Thomas refer to as the "DRY principle": Don't Repeat Yourself(2000). I think David Parnas says it best: "Copy and paste is a design error" (McConnell 1998b).

A routine is too long In object-oriented programming, routines longer than a screen are rarely needed and usually represent the attempt to force-fit a structured programming foot into an object-oriented shoe.

One of my clients was assigned the task of breaking up a legacy system's longest routine, which was more than 12,000 lines long. With effort, he was able to reduce the size of the largest routine to only about 4,000 lines.

One way to improve a system is to increase its modularity—increase the number of well-defined, well-named routines that do one thing and do it well. When changes lead you to revisit a section of code, take the opportunity to check the modularity of the routines in that section. If a routine would be cleaner if part of it were made into a separate routine, create a separate routine.

A loop is too long or too deeply nested Loop innards tend to be good candidates for being converted into routines, which helps to better factor the code and to reduce the loop's complexity.

A class has poor cohesion If you find a class that takes ownership for a hodgepodge of unrelated responsibilities, that class should be broken up into multiple classes, each of which has responsibility for a cohesive set of responsibilities.

A class interface does not provide a consistent level of abstraction Even classes that begin life with a cohesive interface can lose their original consistency. Class interfaces tend to morph over time as a result of modifications that are made in the heat of the moment and that favor expediency to interface integrity. Eventually the class interface becomes a Frankensteinian maintenance monster that does little to improve the intellectual manageability of the program.

A parameter list has too many parameters Well-factored programs tend to have many small, well-defined routines that don't need large parameter lists. A long parameter list is a warning that the abstraction of the routine interface has not been well thought out.

Changes within a class tend to be compartmentalized Sometimes a class has two or more distinct responsibilities. When that happens you find yourself changing either one part of the class or another part of the class—but few changes affect both parts of the class. That's a sign that the class should be cleaved into multiple classes along the lines of the separate responsibilities.

Changes require parallel modifications to multiple classes I saw one project that had a checklist of about 15 classes that had to be modified whenever a new kind of output was added. When you find yourself routinely making changes to the same set of classes, that suggests the code in those classes could be rearranged so that changes affect

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

24.3. Specific Refactorings

In this section, I present a catalog of refactorings, many of which I describe by summarizing the more detailed descriptions presented in Refactoring (Fowler 1999). I have not, however, attempted to make this catalog exhaustive. In a sense, every case 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 space, I've focused on the refactorings I personally have found most useful.

Data-Level Refactorings

Here are refactorings that improve the use of variables and other kinds of data.

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.

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.

Move an expression inline Replace an intermediate variable that was assigned the result of an expression with the expression itself.

Replace an expression with a routine Replace an expression with a routine (usually so that the expression isn't duplicated in the code).

Introduce an intermediate variable Assign an expression to an intermediate variable whose name summarizes the purpose of the expression.

Convert a multiuse 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.

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.

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 can apply to simple numeric types like Money and Temperature. It can also apply to enumerated types like Color, Shape, Country, or OutputType.

Convert a set of type codes to a class or an enumeration In older programs, it's common to see associations like

```
const int SCREEN = 0;  
  
const int PRINTER = 1;  
  
const int FILE = 2;
```

Rather than defining standalone constants, create a class so that you can receive the benefits of stricter type checking and set yourself up to provide richer semantics for OutputType if you ever need to. Creating an enumeration is sometimes a good alternative to creating a class.

Convert a set of type codes to a class with subclasses If the different elements associated with different types might have different behavior, consider creating a base class for the type with subclasses for each type code. For the OutputType base class, you might create subclasses like Screen, Printer, and File.

Change an array to an object If you're using an array in which different elements are different types, create an object that has a field for each former element of the array.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

24.4. Refactoring Safely

Refactoring is a powerful technique for improving code quality. Like all powerful tools, refactoring can cause more harm than good if misused. A few simple guidelines can prevent refactoring missteps.

Opening up a working system is more like opening up a human brain and replacing a nerve than opening up a sink and replacing a washer. Would maintenance be easier if it was called "Software Brain Surgery?"

—Gerald Weinberg

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, recompiling and retesting after a refactoring before doing the next one.

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 find the code suddenly going sideways while you're 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.

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.

Retest Reviews of changed code should be complemented by retests. Of course, this is dependent on having a good set of test cases in the first place. Regression testing and other test topics are described in more detail in [Chapter 22, "Developer Testing."](#)

Add test cases In addition to retesting with your old tests, add new unit tests to exercise the new code. Remove any test cases that have been made obsolete by the refactorings.

Review the changes If reviews are important the first time through, they are even more important during subsequent modifications. Ed Yourdon reports that programmers typically have more than a 50 percent chance of making an error on their first attempt to make a change (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, as shown in [Figure 24-1](#). 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.

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

[View full size image]

100%

I

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

24.5. 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. Consider the following guidelines when deciding which refactorings are most important:

Refactor when you add a routine When you add a routine, check whether related routines are well organized. If not, refactor them.

Refactor when you add a class Adding a class often brings issues with existing code to the fore. Use this time as an opportunity to refactor other classes that are closely related to the class you're adding.

Refactor when you fix a defect Use the understanding you gain from fixing a bug to improve other code that might be prone to similar defects.

Target error-prone modules Some modules are more error-prone and brittle than others. Is there a section of code that you and everyone else on your team is afraid of? That's probably an error-prone module. Although most people's natural tendency is to avoid these challenging sections of code, targeting these sections for refactoring can be one of the more effective strategies (Jones 2000).

Cross-Reference

For more on error-prone code, see "[Which Classes Contain the Most Errors?](#)" in [Section 22.4](#).

Target high-complexity modules Another approach is to focus on modules that have the highest complexity ratings. (See "[How to Measure Complexity](#)" in [Section 19.6](#) for details on these metrics.) One classic study found that program quality improved dramatically when maintenance programmers focused their improvement efforts on the modules that had the highest complexity (Henry and Kafura 1984).

In a maintenance environment, improve the parts you touch Code that is never modified doesn't need to be refactored. But when you do touch a section of code, be sure you leave it better than you found it.

Define an interface between clean code and ugly code, and then move code across the interface The "real world" is often messier than you'd like. The messiness might come from complicated business rules, hardware interfaces, or software interfaces. A common problem with geriatric systems is poorly written production code that must remain operational at all times.

An effective strategy for rejuvenating geriatric production systems is to designate some code as being in the messy real world, some code as being in an idealized new world, and some code as being the interface between the two. [Figure 24-2](#) illustrates this idea.

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

[\[View full size image\]](#)



[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/2464

The process of refactoring has a lot in common with the process of fixing defects. For more on fixing defects, see [Section 23.3, "Fixing a Defect."](#) The risks associated with refactoring are similar to the risks associated with code tuning. For more on managing code-tuning risks, see [Section 25.6, "Summary of the Approach to Code Tuning."](#)

Fowler, Martin. Refactoring: Improving the Design of Existing Code. Reading, MA: Addison Wesley, 1999. This is the definitive guide to refactoring. It contains detailed discussions of many of the specific refactorings summarized in this chapter, as well as a handful of other refactorings not summarized here. Fowler provides numerous code samples to illustrate how each refactoring is performed step by step.

Key Points

- Program changes are a fact of life both during initial development and after initial release.
- Software can either improve or degrade as it's changed. The Cardinal Rule of Software Evolution is that internal quality should improve with code evolution.
- One key to success in refactoring is learning to pay attention to the numerous warning signs or smells that indicate a need to refactor.
- Another key to refactoring success is learning numerous specific refactorings.
- A final key to success is having a strategy for refactoring safely. Some refactoring approaches are better than others.
- Refactoring during development is the best chance you'll get to improve your program, to make all the changes you'll wish you'd made the first time. Take advantage of these opportunities during development!

Chapter 25. Code-Tuning Strategies

cc2e.com/2578

Contents

- [Performance Overview page 588](#)
- [Introduction to Code Tuning page 591](#)
- [Kinds of Fat and Molasses page 597](#)
- [Measurement page 603](#)
- [Iteration page 605](#)
- [Summary of the Approach to Code Tuning page 606](#)

Related Topics

- Code-tuning techniques: [Chapter 26](#)
- 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 [Chapter 26, "Code-Tuning Techniques."](#) 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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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—and 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.

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity.

—W. A. Wulf

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 six pictures at a time. Uploading 50 pictures is a tedious process that requires dozens of mouse clicks and lots of navigation through the six-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 **KEY POINT** characteristics to make your code faster. Your work on speed might hurt overall performance rather than help it.

Performance and Code Tuning

Once you've chosen efficiency as a priority, whether its emphasis is on speed or on size, you should consider several options before choosing to improve either speed or size at the code level. Think about efficiency from each of these viewpoints:

- Program requirements
- Program design
- Class and routine design
-

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

25.2. Introduction to Code Tuning

What is code tuning's appeal? 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. And 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 in 20 microseconds, tweak a few lines, and reduce the execution speed to 2 microseconds.

It's also appealing because mastering the art of writing efficient code is a rite of passage to becoming a serious programmer. In tennis, you don't get any game points for the way you pick up a tennis ball, but you still need to learn the right way to do it. You can't just lean over and pick it up with your hand. If you're good, you whack it with the head of your racket until it bounces waist high and then you catch it. Whacking it more than three times, even not bouncing it the first time, is a serious failing. Despite its seeming unimportance, the way you pick up the ball carries a certain cachet within tennis culture. Similarly, no one but you and other programmers usually cares how tight your code is. Nonetheless, within the programming culture, writing microefficient code proves you're cool.

The problem with code tuning is that efficient code isn't necessarily "better" code. That's the subject of the next few sections.

The Pareto Principle

The Pareto Principle, also known as the 80/20 rule, states that you can get 80 percent of the result with 20 percent of the effort. The principle applies to a lot of areas other than programming, but it definitely applies to program optimization.



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 Fortran Programs," Donald Knuth found that less than four percent of a program usually accounts for more than 50 percent of its run time (1971).

Knuth used a line-count profiler to discover this surprising relationship, and the implications for optimization are clear. You should measure the code to find the hot spots and then put your resources into optimizing the few percent that are used the most. Knuth profiled his line-count program and found that it was spending half its execution time in two loops. He changed a few lines of code and doubled the speed of the profiler in less than an hour.

Jon Bentley describes a case in which a 1000-line program spent 80 percent of its time in a five-line square-root routine. By tripling the speed of the square-root routine, he doubled the speed of the program (1988). The Pareto Principle is also the source of the advice to write most of the code in an interpreted language like Python and then rewrite the hot spots in a faster compiled language like C.

Bentley also reports the case of a team that discovered half an operating system's time being spent in a small loop. They rewrote the loop in microcode and made the loop 10 times faster, but it didn't change the system's performance—they had rewritten the system's idle loop!

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 might prevent completion. Complete it first, and then perfect it. The part that needs to be perfect is usually small.

Old Wives' Tales

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

25.3. Kinds of Fat and Molasses

In code tuning you find the parts of a program that are as slow as molasses in winter and as big as Godzilla and change them so that they are as fast as greased lightning and so skinny they can hide in the cracks between the other bytes in RAM. You always have to profile the program to know with any confidence which parts are slow and fat, but some operations have a long history of laziness and obesity, and you can start by investigating them.

Common Sources of Inefficiency

Here are several common sources of inefficiency:

Input/output operations One of the most significant sources of inefficiency is unnecessary input/output (I/O). If you have a choice of working with a file in memory vs. on disk, in a database, or across a network, use an in-memory data structure unless space is critical.

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: 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.

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

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

Note: 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.

If the test had used a slower medium for external access—for example, a hard disk across a network connection—the difference would have been even greater. The performance looks like this 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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

25.4. Measurement

Because small parts of a program usually consume a disproportionate share of the run time, measure your code to find the hot spots. 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 counterintuitive. 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.



KEY POINT Experience doesn't help much with optimization either. A person's experience might have come from an old machine, language, or compiler—when any of those things changes, all bets are off. You can never be sure about the effect of an optimization until you measure the effect.

Many years ago now I wrote a program that summed the elements in a matrix. The original code looked like this:

C++ Example of Straightforward Code to Sum the Elements in a Matrix

```
sum = 0;

for ( row = 0; row < rowCount; row++ ) {
    for ( column = 0; column < columnCount; column++ ) {
        sum = sum + matrix[ row ][ column ];
    }
}
```

This code was straightforward, but performance of the matrix-summation routine was critical, and I knew that all the array accesses and loop tests had to be expensive. I knew from computer-science classes that every time the code accessed a two-dimensional array, it performed expensive multiplications and additions. For a 100-by-100 matrix, that totaled 10,000 multiplications and additions, plus the loop overhead. By converting to pointer notation, I reasoned, I could increment a pointer and replace 10,000 expensive multiplications with 10,000 relatively cheap increment operations. I carefully converted the code to pointer notation and got this:

C++ Example of an Attempt to Tune Code to Sum the Elements in a Matrix

```
sum = 0;

elementPointer = matrix;
lastElementPointer = matrix[ rowCount - 1 ][ columnCount - 1 ] + 1;

while ( elementPointer < lastElementPointer ) {
    sum = sum + *elementPointer++;
}
```

Further Reading

Jon Bentley reported a similar experience in which converting to pointers hurt performance by about 10 percent. The same conversion had—in another setting—improved performance more than 50 percent. See "Software Exploratorium: Writing Efficient 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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 (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.

Even though most individual optimizations were small, cumulatively they were significant. To judge from the percentage improvements, no three or even four optimizations would have met my performance goal. But the final combination was effective. The moral of the story is that if you dig deep enough, you can make some surprising gains.

The code tuning I did in this case is the most aggressive code tuning I've ever done. At the same time, the final code is the most unreadable, unmaintainable code I've ever written. The initial algorithm is complicated. The code resulting from the high-level language transformation was barely readable. The translation to assembler produced a single 500-line routine that I'm afraid to look at. In general, this relationship between code tuning and code quality holds true. Here's a table that shows a history of the optimizations:

Optimization	Benchmark Time	Improvement
Implement initially—straightforward	21:40	—
Convert from bit fields to arrays	7:30	65%
Unroll innermost for 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 logic in loops	0:45	53%

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

25.6. Summary of the Approach to Code Tuning

You should take the following steps as you consider whether code tuning can help you improve the performance of a program:

1.

Develop the software by using well-designed code that's easy to understand and modify.

2.

If performance is poor,

a.

Save a working version of the code so that you can get back to the "last known good state."

b.

Measure the system to find hot spots.

c.

Determine whether the weak performance comes from inadequate design, data types, or algorithms and whether code tuning is appropriate. If code tuning isn't appropriate, go back to step 1.

d.

Tune the bottleneck identified in step (c).

e.

Measure each improvement one at a time.

f.

If an improvement doesn't improve the code, revert to the code saved in step (a). (Typically, more than half the attempted tunings will produce only a negligible improvement in performance or degrade performance.)

3.

Repeat from step 2.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

[cc2e.com/2585](#)

This section contains resources related to performance improvement in general. For additional resources that discuss specific code-tuning techniques, see the "[Additional Resources](#)" section at the end of [Chapter 26](#).

Performance

Smith, Connie U. and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Boston, MA: Addison-Wesley, 2002. This book covers software performance engineering, an approach for building performance into software systems at all stages of development. It makes extensive use of examples and case studies for several kinds of programs. It includes specific recommendations for Web applications and pays special attention to scalability.

[cc2e.com/2592](#)

Newcomer, Joseph M. "Optimization: Your Worst Enemy." May 2000, <http://www.flounder.com/optimization.htm>. Newcomer is an experienced systems programmer who describes the various pitfalls of ineffective optimization strategies in graphic detail.

Algorithms and Data Types

[cc2e.com/2599](#)

Knuth, Donald. *The Art of Computer Programming*, vol. 1, *Fundamental Algorithms*, 3d ed. Reading, MA: Addison-Wesley, 1997.

Knuth, Donald. *The Art of Computer Programming*, vol. 2, *Seminumerical Algorithms*, 3d ed. Reading, MA: Addison-Wesley, 1997.

Knuth, Donald. *The Art of Computer Programming*, vol. 3, *Sorting and Searching*, 2d ed. Reading, MA: Addison-Wesley, 1998.

These are the first three volumes of a series that was originally intended to grow to seven volumes. They can be somewhat intimidating. In addition to the English description of the algorithms, they're described in mathematical notation or MIX, an assembly language for the imaginary MIX computer. The books contain exhaustive details on a huge number of topics, and if you have an intense interest in a particular algorithm, you won't find a better reference.

Sedgewick, Robert. *Algorithms in Java*, [Parts I-IV](#), 3d ed. Boston, MA: Addison-Wesley, 2002. This book's four parts contain a survey of the best methods of solving a wide variety of problems. Its subject areas include fundamentals, sorting, searching, abstract data type implementation, and advanced topics. Sedgewick's *Algorithms in Java*, [Part V](#), 3d ed. (2003) covers graph algorithms. Sedgewick's *Algorithms in C++*, [Parts I-IV](#), 3d ed. (1998), *Algorithms in C++*, [Part V](#), 3d ed. (2002), *Algorithms in C*, [Parts I-IV](#), 3d ed. (1997), and *Algorithms in C*, [Part V](#), 3d ed. (2001) are similarly organized. Sedgewick was a Ph.D. student of Knuth's.

[cc2e.com/2506](#)

Checklist: Code-Tuning Strategies

Overall Program Performance

-

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- Performance is only one aspect of overall software quality, and it's usually not the most important. Finely tuned code is only one aspect of overall performance, and it's usually not the most significant. Program architecture, detailed design, and data-structure and algorithm selection usually have more influence on a program's execution speed and size than the efficiency of its code does.
- Quantitative measurement is a key to maximizing performance. It's needed to find the areas in which performance improvements will really count, and it's needed again to verify that optimizations improve rather than degrade the software.
- Most programs spend most of their time in a small fraction of their code. You won't know which code that is until you measure it.
- Multiple iterations are usually needed to achieve desired performance improvements through code tuning.
- The best way to prepare for performance work during initial coding is to write clean code that's easy to understand and modify.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 26. Code-Tuning Techniques

cc2e.com/2665

Contents

- [Logic page 610](#)
- [Loops page 616](#)
- [Data Transformations page 624](#)
- [Expressions page 630](#)
- [Routines page 639](#)
- [Recoding in a Low-Level Language page 640](#)
- [The More Things Change, the More They Stay the Same page 643](#)

Related Topics

- Code-tuning strategies: [Chapter 25](#)
- Refactoring: [Chapter 24](#)

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 (bearing in mind the warnings from [Chapter 25](#), "Code-Tuning Strategies"), you have a rich set of techniques at your disposal.

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.

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 internal structure," these changes degrade the internal structure in exchange for gains in performance. This is true by definition. If the changes 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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

26.1. Logic

Much of programming consists of manipulating logic. This section describes how to manipulate logical expressions to your advantage.

Cross-Reference

For other details on using statement logic, see [Chapters 14–19](#).

Stop Testing When You Know the Answer

Suppose you have a statement like

```
if ( 5 < x ) and ( x < 10 ) then ...
```

Once you've determined that x is greater than 5, you don't need to perform the second half of the test.

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.

Cross-Reference

For more on short-circuit evaluation, see "[Knowing How Boolean Expressions Are Evaluated](#)" in [Section 19.1](#).

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:

```
if ( 5 < x ) then  
  if ( x < 10 ) then ...
```

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

```
negativeInputFound = false;  
  
for ( i = 0; i < count; i++ ) {  
  
  if ( input[ i ] < 0 ) {  
    negativeInputFound = true;  
  }  
}
```

A better approach would be to stop scanning as soon as you find a negative value. Any of these approaches would solve the problem:

-

- Add a `break` statement after the `negativeInputFound = true` line.

-

- If your language doesn't have `break`, emulate a `break` with a `goto` that goes to the first statement after the

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

26.2. Loops

Because loops are executed many times, the hot spots in a program are often inside loops. The techniques in this section make the loop itself faster.

Cross-Reference

For other details on loops, see [Chapter 16, "Controlling Loops."](#)



Unswitching

Switching refers to making a decision inside a loop every time it's executed. If the decision doesn't change while the loop is executing, you can unswitch the loop by making the decision outside the loop. Usually this requires turning the loop inside out, putting loops inside the conditional rather than putting the conditional inside the loop. Here's an example of a loop before unswitching:

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++ ) {
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 book, but modest changes in the implementation of a specific data type can also improve 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:

Cross-Reference

For details on using integers and floating point, see [Chapter 12, "Fundamental Data Types."](#)



Visual Basic Example of a Loop That Uses a Time-Consuming Floating-Point Loop Index

```
Dim x As Single  
  
For x = 0 to 99  
  
    a( x ) = 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  
  
    a( 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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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.

Cross-Reference

For more information on expressions, see [Section 19.1, "Boolean Expressions."](#)

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 $\text{sqrt}()$ 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:

- Replace multiplication with addition.
- Replace exponentiation with multiplication.
- Replace trigonometric routines with their trigonometric identities.
- Replace longlong integers with longs or ints (but watch for performance issues associated with using native-length vs. non-native-length integers)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 a low-level language. Long, tortuous routines are hard enough to understand on their own; in a low-level language like assembler, they're impossible.

Cross-Reference

For details on working with routines, see [Chapter 7, "High-Quality Routines."](#)

Rewrite Routines Inline

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.

Modern computers collect a far smaller toll for calling a routine. Here are the results of putting a string-copy routine inline:

Language	Routine Time	Inline-Code Time	Time Savings
C++	0.471	0.431	8%
Java	13.1	14.4	-10%

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 via 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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

26.6. Recoding in a Low-Level Language

One long-standing 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 a low-level language. If you're coding in C++, the low-level language might be assembler. If you're coding in Python, the low-level language might be C. Recoding in a low-level language tends to improve both speed and code size. Here is a typical approach to optimizing with a low-level language:

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.

Cross-Reference

For details on the phenomenon of a small percentage of a program accounting for most of its run time, see "[The Pareto Principle](#)" in [Section 25.2](#).

4.

Recode a few small pieces in a low-level language to improve overall performance.

Whether you follow this well-beaten path depends on how comfortable you are with low-level languages, how well-suited the problem is to low-level languages, and on your level of desperation. I got my first exposure to this technique on the Data Encryption Standard 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;
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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 for embedded systems, real-time systems, and other systems with strict speed or space restrictions can still benefit from it.

The need to measure the impact of each and every attempt at code tuning has been a constant since Donald Knuth published his study of Fortran programs in 1971. According to the measurements in this chapter, the effect of any specific optimization is actually less predictable than it was 10 years ago. The effect of each code tuning is affected by the programming language, compiler, compiler version, code libraries, library versions, and compiler settings, among other things.

Code tuning invariably involves tradeoffs among complexity, readability, simplicity, and maintainability on the one hand and a desire to improve performance on the other. It introduces a high degree of maintenance overhead because of all the reprofiling that's required.

I have found that insisting on measurable improvement is a good way to resist the temptation to optimize prematurely and a good way to enforce a bias toward clear, straightforward code. If an optimization is important enough to haul out the profiler and measure the optimization's effect, then it's probably important enough to allow—as long as it works. But if an optimization isn't important enough to haul out the profiling machinery, it isn't important enough to degrade readability, maintainability, and other code characteristics. The impact of unmeasured code tuning on performance is speculative at best, whereas the impact on readability is as certain as it is detrimental.

Additional Resources

cc2e.com/2679

My favorite reference on code tuning is *Writing Efficient Programs* (Bentley, Englewood Cliffs, NJ: Prentice Hall, 1982). The book is out of print but worth reading if you can find it. It's an expert treatment of code tuning, broadly considered. Bentley describes techniques that trade time for space and space for time. He provides several examples of redesigning data types to reduce both space and time. His approach is a little more anecdotal than the one taken here, and his anecdotes are interesting. He takes a few routines through several optimization steps so that you can see the effects of first, second, and third attempts on a single problem. Bentley strolls through the primary contents of the book in 135 pages. The book has an unusually high signal-to-noise ratio—it's one of the rare gems that every practicing programmer should own.

Appendix 4 of *Bentley's Programming Pearls*, 2d ed. (Boston, MA: Addison-Wesley, 2000) contains a summary of the code-tuning rules from his earlier book.

cc2e.com/2686

You can also find a full array of technology-specific optimization books. Several are listed below, and the Web link to the left contains an up-to-date list.

Booth, Rick. *Inner Loops: A Sourcebook for Fast 32-bit Software Development*. Boston, MA: Addison-Wesley, 1997.

Gerber, Richard. *Software Optimization Cookbook: High-Performance Recipes for the Intel Architecture*. Intel Press, 2002.

Hasan, Jeffrey and Kenneth Tu. *Performance Tuning and Optimizing ASP.NET Applications*. Berkeley, CA: Apress, 2003.

Killelea, Patrick. *Web Performance Tuning*, 2d ed. Sebastopol, CA: O'Reilly & Associates, 2002.

Larman, Craig and Rhett Guthrie. *Java 2 Performance and Idiom Guide*. Englewood Cliffs, NJ: Prentice Hall, 2000.

Shirazi, Jack. *Java Performance Tuning*. Sebastopol, CA: O'Reilly & Associates, 2000.

Wilson, Steve and Jeff Kesselman. *Java Platform Performance: Strategies and Tactics*. Boston, MA: Addison-Wesley, 2000.

Key Points

- Results of optimizations vary widely with different languages, compilers, and environments. Without measuring each specific optimization, you'll have no idea whether it will help or hurt your program.
-
- The first optimization is often not the best. Even after you find a good one, keep looking for one that's better.
-
- Code tuning is a little like nuclear energy. It's a controversial, emotional topic. Some people think it's so detrimental to reliability and maintainability that they won't do it at all. Others think that with proper safeguards, it's beneficial. If you decide to use the techniques in this chapter, apply them with care.

Part VI: System Considerations

[In this part:](#)

[Chapter 27. How Program Size Affects Construction](#)

[Chapter 28. Managing Construction](#)

[Chapter 29. Integration](#)

[Chapter 30. Programming Tools](#)

In this part:

[Chapter 27: How Program Size Affects Construction](#)

[Chapter 28: Managing Construction](#)

[Chapter 29: Integration](#)

[Chapter 30: Programming Tools](#)

Chapter 27. How Program Size Affects Construction

cc2e.com/2761

Contents

- [Communication and Size page 650](#)
- [Range of Project Sizes page 651](#)
- [Effect of Project Size on Errors page 651](#)
- [Effect of Project Size on Productivity page 653](#)
- [Effect of Project Size on Development Activities page 654](#)

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.

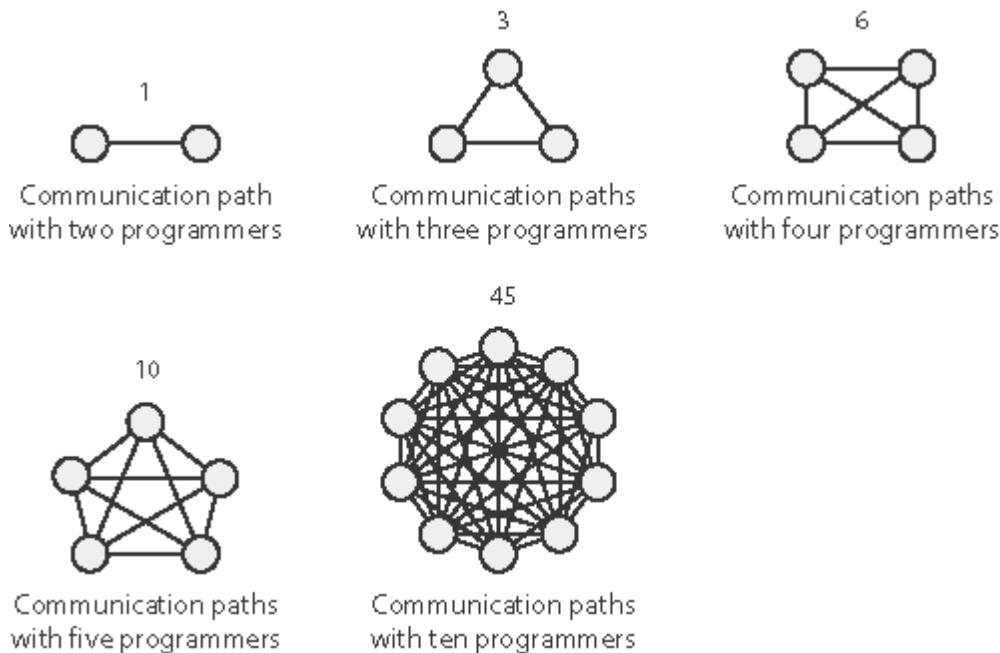
Even though it's 10 times as large as the original Gigatron, the Gigatron Deluxe won't take 10 times the effort to develop; it'll take 30 times the effort. Moreover, 30 times the total effort doesn't imply 30 times as much construction. It probably implies 25 times as much construction and 40 times as 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 contrast, if you're accustomed to working on large projects, you might use approaches that are too formal on a small project. This chapter describes how you can economize to keep a small project from toppling under the weight of its own overhead.

27.1. Communication and Size

If you're the only person on a project, the only communication path is between you and the customer, unless you count the path across your corpus callosum, the path that connects the left side of your brain to the right. As the number of people on a project increases, the number of communication paths increases, too. The number doesn't increase additively as the number of people increases. It increases multiplicatively, proportionally to the square of the number of people, as illustrated in [Figure 27-1](#).

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



KEY POINT As you can see, a two-person project has only one path of communication. A five-person project has 10 paths. A ten-person project has 45 paths, assuming that every person talks to every other person. The 10 percent of projects that have 50 or more programmers have at least 1,200 potential paths. The more communication paths you have, the more time you spend communicating and the more opportunities are created for communication mistakes. Larger-size projects demand organizational techniques that streamline communication or limit it in a sensible way.

The typical approach taken to streamlining communication is to formalize it in documents. Instead of having 50 people talk to each other in every conceivable combination, 50 people read and write documents. Some are text documents; some are graphic. Some are printed on paper; others are kept in electronic form.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

27.2. Range of Project Sizes

Is the size of the project you're working on typical? The wide range of project sizes means that you can't consider any single size to be typical. One way of thinking about project size is to think about the size of a project team. Here's a crude estimate of the percentages of all projects that are done by teams of various sizes:

Team Size	Approximate Percentage of Projects
1–3	25%
4–10	30%
11–25	20%
26–50	15%
50+	10%

Source: Adapted from "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).

One aspect of project size data that might not be immediately apparent is the difference between the percentages of projects of various sizes and the number of programmers who work on projects of each size. Because larger projects use more programmers on each project than do small ones, they employ a large percentage of all programmers. Here's a rough estimate of the percentage of all programmers who 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%

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).

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

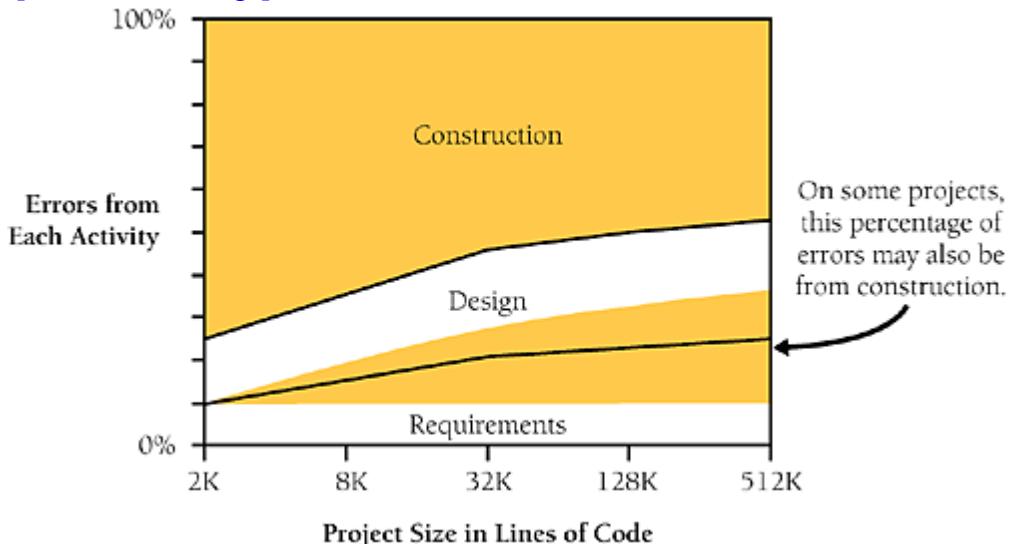
[NEXT ▶](#)

27.3. Effect of Project Size on Errors

Both quantity and type of errors are affected by project size. You might not think that error type would be affected, but as project size increases, a larger percentage of errors can usually be attributed to mistakes in requirements and design, as shown in [Figure 27-2](#).

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)

[View full size image]



Cross-Reference

For more details on errors, see [Section 22.4, "Typical Errors."](#)



On small projects, construction errors make up about 75 percent of all the errors found. Methodology has less influence on code quality, and the biggest influence on program quality is often the skill of the individual writing the program (Jones 1998).

On larger projects, construction errors can taper off to about 50 percent of the total errors; requirements and architecture errors make up the difference. Presumably this is related to the fact that more requirements development and 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).



KEY POINT

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 1000 lines 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.

Table 27-1. Project Size and Typical Error Density

Project Size (in Lines of Code)

Typical Error Density

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

27.4. Effect of Project Size on Productivity

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.



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 and three for the large (1984). [Table 27-2](#) gives the inside scoop on the general relationship between project size and productivity.

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)

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).

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 nonprogrammer 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.

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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

27.5. Effect of Project Size on Development Activities

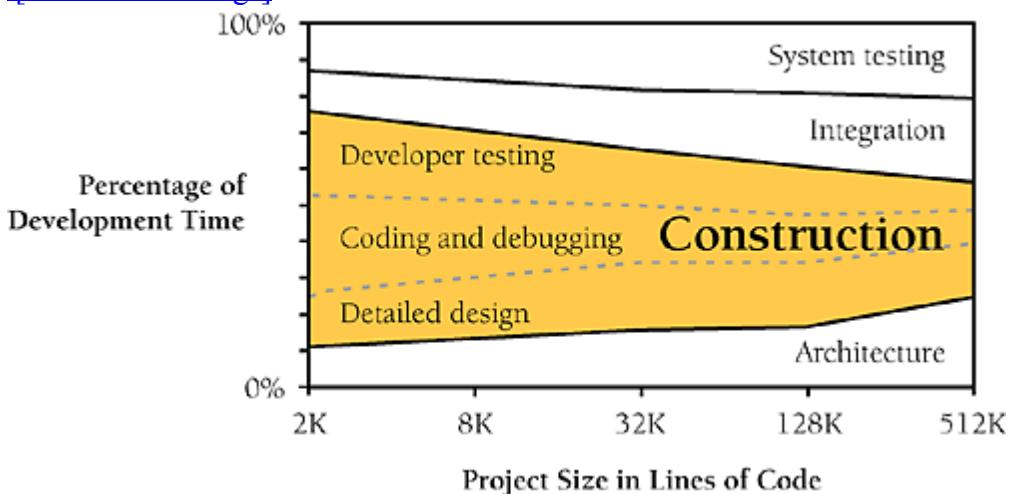
If you are working on a one-person project, the biggest influence on the project's success or failure is you. If you're working on a 25-person project, it's conceivable that you're still the biggest influence, but it's more likely that no one person will wear the medal for that distinction; your organization will be a stronger influence on the project's success or failure.

Activity Proportions and Size

As project size increases and the need for formal communications increases, the kinds of activities a project needs change dramatically. [Figure 27-3](#) shows the proportions of development activities for projects of different sizes.

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)

[\[View full size image\]](#)



KEY POINT

On a small project, construction is the most prominent activity by far, taking up as much as 65 percent of the total development time. On a medium-size project, construction is still the dominant activity but its share of the total effort falls to about 50 percent. On very large projects, architecture, integration, and system testing take up more time and construction becomes less dominant. In short, as project size increases, construction becomes a smaller part of the total effort. The chart looks as though you could extend it to the right and make construction disappear altogether, so in the interest of protecting my job, I've cut it off at 512K.

Construction becomes less predominant because as project size increases, the construction activities—detailed design, coding, debugging, and unit testing—scale up proportionately but many other activities scale up faster. [Figure 27-4](#) provides an illustration.

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



[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/2768

Use the following resources to investigate this chapter's subject further:

Boehm, Barry and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison-Wesley, 2004. Boehm and Turner describe how project size affects the use of agile and plan-driven methods, along with other agile and plandriven issues.

Cockburn, Alistair. *Agile Software Development*. Boston, MA: Addison-Wesley, 2002. [Chapter 4](#) discusses issues involved in selecting appropriate project methodologies, including project size. [Chapter 6](#) introduces Cockburn's Crystal Methodologies, which are defined approaches for developing projects of various sizes and degrees of criticality.

Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981. Boehm's book is an extensive treatment of the cost, productivity, and quality ramifications of project size and other variables in the software-development process. It includes discussions of the effect of size on construction and other activities. [Chapter 11](#) is an excellent explanation of software's diseconomies of scale. Other information on project size is spread throughout the book. Boehm's 2000 book *Software Cost Estimation with Cocomo II* contains much more up-to-date information on Boehm's Cocomo estimating model, but the earlier book provides more in-depth background discussions that are still relevant.

Jones, Capers. *Estimating Software Costs*. New York, NY: McGraw-Hill, 1998. This book is packed with tables and graphs that dissect the sources of software development productivity. For the impact of project size specifically, Jones's 1986 book, *Programming Productivity*, contains an excellent discussion in the section titled "The Impact of Program Size" in [Chapter 3](#).

Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition (2d ed.). Reading, MA: Addison-Wesley, 1995. Brooks was the manager of IBM's OS/360 development, a mammoth project that took 5000 staff-years. He discusses management issues pertaining to small and large teams and presents a particularly vivid account of chief-programmer teams in this engaging collection of essays.

DeGrace, Peter, and Leslie Stahl. *Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms*. Englewood Cliffs, NJ: Yourdon Press, 1990. As the title suggests, this book catalogs approaches to developing software. As noted throughout this chapter, your approach needs to vary as the size of the project varies, and DeGrace and Stahl make that point clearly. The section titled "Attenuating and Truncating" in [Chapter 5](#) discusses customizing software-development processes based on project size and formality. The book includes descriptions of models from NASA and the Department of Defense and a remarkable number of edifying illustrations.

Jones, T. Capers. "Program Quality and Programmer Productivity." IBM Technical Report TR 02.764 (January 1977): 42–78. Also available in Jones's *Tutorial: Programming Productivity: Issues for the Eighties*, 2d ed. Los Angeles, CA: IEEE Computer Society Press, 1986. This paper contains the first in-depth analysis of the reasons large projects have different spending patterns than small ones. It's a thorough discussion of the differences between large and small projects, including requirements and quality-assurance measures. It's dated but still interesting.

Key Points

- As project size increases, communication needs to be supported. The point of most methodologies is to reduce communications problems, and a methodology should live or die on its merits as a communication facilitator.
- All other things being equal, productivity will be lower on a large project than on a small one.
- All other things being equal, a large project will have more errors per thousand lines of code than a small one.
- Activities that are taken for granted on small projects must be carefully planned on larger ones. Construction becomes less predominant as project size increases.
- Scaling up a lightweight methodology tends to work better than scaling down a heavyweight methodology. The most effective approach of all is using a "rightweight" methodology.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 28. Managing Construction

cc2e.com/2836

Contents

- [Encouraging Good Coding page 662](#)
- [Configuration Management page 664](#)
- [Estimating a Construction Schedule page 671](#)
- [Measurement page 677](#)
- [Treating Programmers as People page 680](#)
- [Managing Your Manager page 686](#)

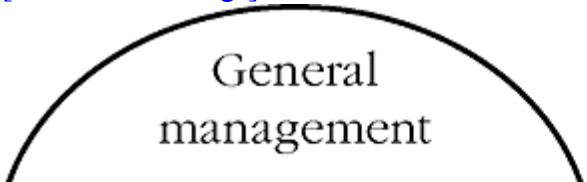
Related Topics

- Prerequisites to construction: [Chapter 3](#)
- Determining the kind of software you're working on: [Section 3.2](#)
- Program size: [Chapter 27](#)
- Software quality: [Chapter 20](#)

Managing software development has been a formidable challenge for the past several decades. As [Figure 28-1](#) suggests, the general topic of software-project management extends beyond the scope of this book, but this chapter discusses a few specific management topics that apply directly to construction. If you're a developer, this chapter will help you understand the issues that managers need to consider. If you're a manager, this chapter will help you understand how management looks to developers as well as how to manage construction effectively. Because the chapter covers a broad collection of topics, several of its sections also describe where you can go for more information.

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

[\[View full size image\]](#)



General
management

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

28.1. Encouraging Good Coding

Because 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 that person.

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.

Considerations in Setting Standards

Standards are more useful in some organizations than in others. Some developers welcome standards because they reduce arbitrary variance in the project. If your group resists adopting strict standards, consider a few alternatives: flexible guidelines, a collection of suggestions rather than guidelines, or a set of examples that embody the best practices.

Techniques for Encouraging Good Coding

This section describes several techniques for achieving good coding practices that are less heavy-handed than laying down rigid coding standards:

Assign two people to every part of the project If two people have to work on each line of code, you'll guarantee that at least two people think it works and is readable. The mechanisms for teaming two people can range from pair programming to mentor/trainee pairs to buddy-system reviews.

Cross-Reference

For more details on pair programming, see [Section 21.2, "Pair Programming."](#)

Review every line of code A code review typically involves the programmer and at least two reviewers. That means that at least three people read every line of code. Another name for peer review is "peer pressure." In addition to providing a safety net in case the original programmer leaves the project, reviews improve code quality because the programmer knows that the code will be read by others. Even if your shop hasn't created explicit coding standards, reviews provide a subtle way of moving toward a group coding standard—decisions are made by the group during reviews, and over time the group derives its own standards.

Cross-Reference

For details on reviews, see [Section 21.3, "Formal Inspections,"](#) and [Section 21.4, "Other Kinds of Collaborative Development Practices."](#)

Require code sign-offs In other fields, technical drawings are approved and signed by the managing engineer. The signature means that to the best of the engineer's knowledge, the drawings are technically competent and error-free. Some companies treat code the same way. Before code is considered to be complete, senior technical personnel must sign the code listing.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

28.2. Configuration Management

A software project is dynamic. The code changes, the design changes, and the requirements change. What's more, changes in the requirements lead to more changes in the design, and changes in the design lead to even more changes in the code and test cases.

What Is Configuration Management?

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.

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.

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.

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.



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 percent had adequate configuration management (SEI 2003).

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" (SCM). SCM focuses on a program's requirements, source code, documentation, and test data.

The systemic problem with SCM is overcontrol. The surest way to stop car 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.

On a small, one-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. 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.

Cross-Reference

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

28.3. Estimating a Construction Schedule



Managing a software project is one of the formidable challenges of the twenty-first century, and estimating the size of a project and the effort required to complete it is one of the most challenging aspects of software-project management. The average large software project is one year late and 100 percent over budget (Standish Group 1994, Jones 1997, Johnson 1999). At the individual level, surveys of estimated vs. actual schedules have found that developers' estimates tend to have an optimism factor of 20 to 30 percent (van Genuchten 1991). This has as much to do with poor size and effort estimates as with poor development efforts. This section outlines the issues involved in estimating software projects and indicates where to look for more information.

Estimation Approaches

You can estimate the size of a project and the effort required to complete it in any of several ways:

Further Reading

For further reading on schedule estimation techniques, see [Chapter 8](#) of Rapid Development (McConnell 1996) and Software Cost Estimation with Cocomo II (Boehm et al. 2000).

- Use estimating 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 walk-through meeting for estimates.
-
- Estimate pieces of the project, and then add the pieces together.
-
- Have people estimate their own tasks, and then add the task estimates together.
-
- Refer to experience on previous projects.
-
- Keep previous estimates and see how accurate they were. Use them to adjust new estimates.

Pointers to more information on these approaches are given in "[Additional Resources on Software Estimation](#)" at the end of this section. Here's a good approach to estimating a project:

Establish objectives Why do you need an estimate? What are you estimating? Are you estimating only construction activities, or all of development? Are you estimating only the effort for your project, or your project plus vacations, holidays, training, and other nonproject activities? 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?

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

28.5. Treating Programmers as People



KEY POINT The abstractness of the programming activity calls for an offsetting naturalness in the office environment and rich contacts among coworkers. Highly technical companies offer parklike corporate campuses, organic organizational structures, comfortable offices, and other "high-touch" environmental features to balance the intense, sometimes arid intellectuality of the work itself. The most successful technical companies combine elements of high-tech and high-touch (Naisbitt 1982). This section describes ways in which programmers are more than organic reflections of their silicon alter egos.

How Do Programmers Spend Their Time?

Programmers spend their time programming, but they also spend time in meetings, on training, on reading their mail, and on just thinking. A 1964 study at Bell Laboratories found that programmers spent their time this way as described in [Table 28-3](#).

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

Activity	Source	Business Code	Person al	Meetin gs	Trainin g	Mail/M isc. Docum ents	Technic al Manual	Operati ng Procedu res,	Progra m Test Misc.	Totals	
Talk or listen		4%		17%	7%	3%			1%	32%	
Talk with manager				1%						1%	
Telepho ne			2%		1%					3%	
Read		14%					2%	2%		18%	
Write/re cord		13%					1%			14%	
Away or out			4%	1%	4%	6%				15%	
Walking		2%		2%	1%		1%			6%	
Miscella neous		2%		3%			1%		1%	11%	
Totals		35%		29%	13%	7%	6%	5%	2%	1%	100%

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

28.6. Managing Your Manager

In software development, nontechnical managers are common, as are managers who have technical experience but who are 10 years behind the times. Technically competent, technically current managers are rare. If you work for one, do whatever you can to keep your job. It's an unusual treat.

In a hierarchy, every employee tends to rise to his level of incompetence.

—The Peter Principle

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:

Plant ideas for what you want to do, and then wait for your manager to have a brainstorm (your idea) about doing what you want to do.

Educate your manager about the right way to do things. This is an ongoing job because managers are often promoted, transferred, or fired.

Focus on your manager's interests, doing what he or she really wants you to do, and don't distract your manager with unnecessary implementation details. (Think of it as "encapsulation" of your job.)

Refuse to do what your manager tells you, and insist on doing your job the right way.

Find another job.

The best long-term solution is to try to educate your manager. That's not always an easy task, but one way you can prepare for it is by reading Dale Carnegie's *How to Win Friends and Influence People*.

Additional Resources on Managing Construction

cc2e.com/2813

Here are a few books that cover issues of general concern in managing software projects:

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley, 1988. Gilb has charted his own course for thirty years, and most of the time he's been ahead of the pack whether or not the pack realizes it. This book is a good example. This was one of the first books to discuss evolutionary development practices, risk management, and the use of formal inspections. Gilb is keenly aware of leading-edge approaches; indeed, this book published more than 15 years ago contains most of the good practices currently flying under the "Agile" banner. Gilb is incredibly pragmatic, and the book is still one of the best software-management books.

McConnell, Steve. *Rapid Development*. Redmond, WA: Microsoft Press, 1996. This book covers project-leadership and project-management issues from the perspective of projects that are experiencing significant schedule pressure, which in my experience is most projects.

Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition (2d ed). Reading, MA: Addison-Wesley, 1995. This book is a hodgepodge of metaphors and folklore related to managing programming projects. It's entertaining, and it will give you many illuminating insights into your own projects. It's based

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- Good coding practices can be achieved either through enforced standards or through more light-handed approaches.
- Configuration management, when properly applied, makes programmers' jobs easier. This especially includes change control.
- Good software estimation is a significant challenge. Keys to success are using multiple approaches, tightening down your estimates as you work your way into the project, and making use of data to create the estimates.
- Measurement is a key to successful construction management. You can find ways to measure any aspect of a project that are better than not measuring it at all. Accurate measurement is a key to accurate scheduling, to quality control, and to improving your development process.
- Programmers and managers are people, and they work best when treated as such.

Chapter 29. Integration

cc2e.com/2985

Contents

- [Importance of the Integration Approach page 689](#)
- [Integration Frequency—Phased or Incremental? page 691](#)
- [Incremental Integration Strategies page 694](#)
- [Daily Build and Smoke Test page 702](#)

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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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, as shown in [Figure 29-1](#).

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



It doesn't matter that the stadium 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.



You can expect some of these benefits from careful integration:

KEY POINT

- Easier defect diagnosis
- Fewer defects
- Less scaffolding
- Shorter time to first working product

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

29.2. Integration Frequency—Phased or Incremental?

Programs are integrated by means of either the phased or the incremental approach.

Phased Integration

Until a few years ago, phased integration was the norm. It follows these well-defined steps, or phases:

1.

Design, code, test, and debug each class. This step is called "unit development."

2.

Combine the classes into one whopping-big system ("system integration").

3.

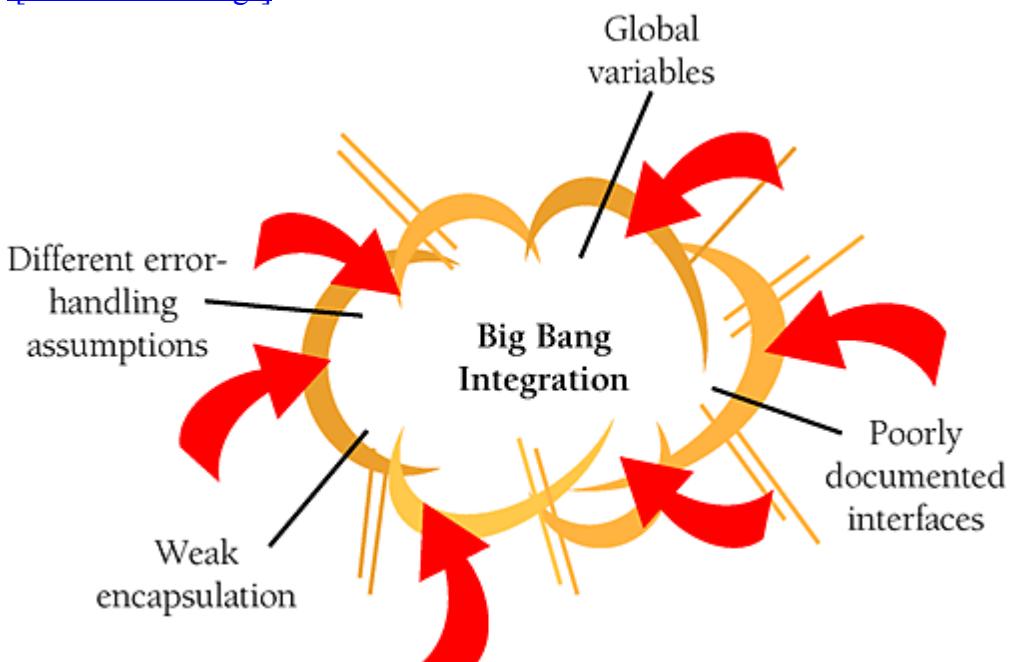
Test and debug the whole system. This is called "system dis-integration." (Thanks to Meilir Page-Jones for this witty observation.)

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 class, an error in the interface between two classes, or an error caused by an interaction between two classes. All classes are suspect.

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," as shown in [Figure 29-2](#).

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

[\[View full size image\]](#)



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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

29.3. Incremental Integration Strategies

With phased integration, you don't have to plan the order in which project components are built. All components are integrated at the same time, so you can build them in any order as long as they're all ready by D-day.

With incremental integration, you have to plan more carefully. Most systems will call for the integration of some components before the integration of others. Planning for integration thus affects planning for construction; the order in which components are constructed has to support the order in which they will be integrated.

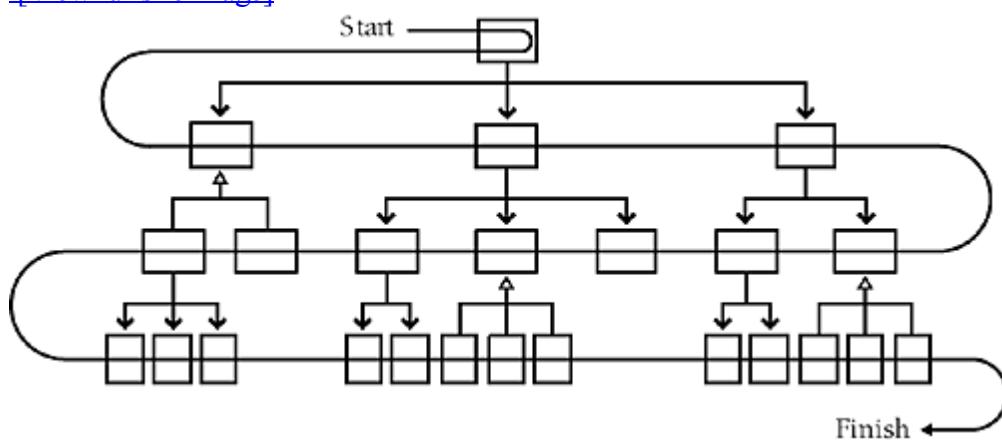
Integration-order strategies come in a variety of shapes and sizes, and none is best in every case. The best integration approach varies from project to project, and the best solution is always the one that you create to meet the specific demands of a specific project. Knowing the points on the methodological number line will give you insight into the possible solutions.

Top-Down Integration

In top-down integration, the class at the top of the hierarchy is written and integrated first. The top is the main window, the applications control loop, the object that contains `main()` in Java, `WinMain()` for Microsoft Windows programming, or similar. Stubs have to be written to exercise the top class. Then, as classes are integrated from the top down, stub classes are replaced with real ones. This kind of integration proceeds as illustrated in [Figure 29-5](#).

Figure 29-5. In top-down integration, you add classes at the top first, at the bottom last

[View full size image]



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 lowlevel 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."

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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.

Further Reading

Much of this discussion is adapted from [Chapter 18](#) of Rapid Development (McConnell 1996). If you've read that discussion, you might skip ahead to the "[Continuous Integration](#)" section.

This simple process produces several significant benefits. It reduces the risk of low quality, which is a risk related to the risk of unsuccessful or problematic integration. By 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.

This process also 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 struggle out of. Teams that haven't used the daily build process 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 allowed to get a little out of sync between these pulses, but every time there's a sync pulse, the code has to come back into alignment. When you insist on keeping the pulses close together, you prevent developers from getting out of sync entirely.

Some organizations build every week, rather than every day. The problem with this is that if the build is broken one week, you might go for several weeks before the next good build. When that happens, you lose virtually all of the benefit of frequent builds.

Check for broken builds For the daily-build process to work, the software that's built has to work. If the software isn't usable, the build is considered to be broken and fixing it becomes top priority.

Each project sets its own standard for what constitutes "breaking the build." The standard needs to set a quality level that's strict enough to keep showstopper defects out but lenient enough to disregard trivial defects, which can paralyze progress if given undue attention.

At a minimum, a "good" build should



[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/2999

Following are additional resources related to this chapter's subjects:

Integration

Lakos, John. Large-Scale C++ Software Design. Boston, MA: 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.

Myers, Glenford J. The Art of Software Testing. New York, NY: John Wiley & Sons, 1979. This classic testing book discusses integration as a testing activity.

Incrementalism

McConnell, Steve. Rapid Development. Redmond, WA: Microsoft Press, 1996. [Chapter 7](#), "Lifecycle Planning," goes into much detail about the tradeoffs involved with more flexible and less flexible life-cycle models. [Chapters 20, 21, 35](#), and 36 discuss specific life-cycle models that support various degrees of incrementalism. [Chapter 19](#) describes "designing for change," a key activity needed to support iterative and incremental development models.

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 experts on the big-picture issues of software development, and the clarity of his explanations reflects the quality of his understanding.

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.

Beck, Kent. Extreme Programming Explained: Embrace Change. Reading, MA: 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 the depth of analysis presented in Gilb's book, but some readers may find Beck's presentation more accessible or more directly applicable to the kind of project they're working on.

Key Points

- The construction sequence and integration approach affect the order in which classes are designed, coded, and tested.
- A well-thought-out integration order reduces testing effort and eases debugging.
- Incremental integration comes in several varieties, and, unless the project is trivial, any one of them is better than phased integration.
- The best integration approach for any specific project is usually a combination of top-down, bottom-up, risk-oriented, and other integration approaches. Tshaped integration and vertical-slice integration are two approaches that often work well.
- Daily builds can reduce integration problems, improve developer morale, and provide useful project management information.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 30. Programming Tools

cc2e.com/3084

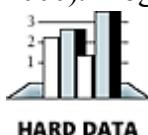
Contents

- [Design Tools page 710](#)
- [Source-Code Tools page 710](#)
- [Executable-Code Tools page 716](#)
- [Tool-Oriented Environments page 720](#)
- [Building Your Own Programming Tools page 721](#)
- [Tool Fantasyland page 722](#)

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.

A programmer can work for many years without discovering some of the most valuable tools available. The mission of this chapter is to survey available tools and help you determine whether you've overlooked any tools that might be useful. If you're a tool expert, you won't find much new information in this chapter. You might skim the earlier parts of

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

30.1. Design Tools

Current design tools consist mainly of graphical tools that create design diagrams. Design tools are sometimes embedded in a computer-aided software engineering (CASE) tool with broader functions; some vendors advertise standalone design tools as CASE tools. Graphical design tools generally allow you to express a design in common graphical notations: UML, architecture block diagrams, hierarchy charts, entity relationship diagrams, or class diagrams. Some graphical design tools support only one notation. Others support a variety.

Cross-Reference

For details on design, see [Chapters 5](#) through [9](#).

In one sense, these design tools are just fancy drawing packages. Using a simple graphics package or pencil and paper, you can draw everything that the tool can draw. But the tools offer valuable capabilities that a simple graphics package can't. If you've drawn a bubble chart and you delete a bubble, a graphical design tool will automatically rearrange the other bubbles, including connecting arrows and lower-level bubbles connected to the bubble. The tool takes care of the housekeeping when you add a bubble, too. A design tool can enable you to move between higher and lower levels of abstraction. A design tool will check the consistency of your design, and some tools can create code directly from your design.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

30.2. Source-Code Tools

The tools available for working with source code are richer and more mature than the tools available for working with designs.

Editing

This group of tools relates to editing source code.

Integrated Development Environments (IDEs)



Some programmers estimate that they spend as much as 40 percent of their time editing source code (Parikh 1986, Ratliff 1987). If that's the case, spending a few extra dollars for the best possible IDE is a good investment.

In addition to basic word-processing functions, good IDEs offer these features:

- Compilation and error detection from within the editor
- Integration with source-code control, build, test, and debugging tools
- Compressed or outline views of programs (class names only or logical structures without the contents, also known as "folding")
- Jump to definitions of classes, routines, and variables
- Jump to all places where a class, routine, or variable is used
- Language-specific formatting
- Interactive help for the language being edited
- Brace (begin-end) matching
- Templates for common language constructs (the editor completing the structure of a for loop after the programmer types for, for example)
- Smart indenting (including easily changing the indentation of a block of statements when logic changes)
- Automated code transforms or refactorings
- Macros programmable in a familiar programming language

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

30.3. Executable-Code Tools

Tools for working with executable code are as rich as the tools for working with source code.

Code Creation

The tools described in this section help with code creation.

Compilers and Linkers

Compilers convert source code to executable code. Most programs are written to be compiled, although some are still interpreted.

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.

An overlay linker helps you put 10 pounds in a five-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.

Build Tools

The purpose of a build tool is to minimize the time needed to build a program using current versions of the program's source files. For each target file in your project, you specify the source files that the target file depends on and how to make it. Build tools also eliminate errors related to sources being in inconsistent states; the build tool ensures they are all brought to a consistent state. Common build tools include the make utility that's associated with UNIX and the ant tool that's used for Java programs.

Suppose you have a target 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.

When you build your program, the make tool checks all the dependencies you've described and determines the files that need to be recompiled. If five of your 250 source files depend on data definitions in userface.h and it changes, make automatically recompiles the five files that depend on it. It doesn't recompile the 245 files that don't depend on userface.h. Using make or ant beats the alternatives of recompiling all 250 files or recompiling each file manually, forgetting one, and getting weird out-of-synch errors. Overall, build tools like make or ant substantially improve the time and reliability of the average compile-link-run cycle.

Some groups have found interesting alternatives to dependency-checking tools like make. For example, the Microsoft Word group found that simply rebuilding all source files was faster than performing extensive dependency checking with make as long as the source files themselves were optimized (header file contents and so on). With this approach, the average developer's machine on the Word project could rebuild the entire Word executable—several million lines of code—in about 13 minutes.

Code Libraries

A good way to write high-quality code in a short amount of time is not to write it all but to find an open source version or buy it instead. You can find high-quality libraries in at least these areas:

-

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

30.4. Tool-Oriented Environments

Some environments have proven to be better suited to tool-oriented programming than others.

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 Windows and other environments. One tribute to the success of the UNIX paradigm is the availability of tools that put a UNIX costume on other machines. For example, cygwin provides UNIXequivalent tools that work under Windows (<http://www.cygwin.com>).

Eric Raymond's The Art of Unix Programming (2004) contains an insightful discussion of the UNIX programming culture.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

30.5. Building Your Own Programming Tools

Suppose you're given five hours to do the job and you have a choice:

- Do the job comfortably in five hours, or
- 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. Doing so might not be costeffective, but there aren't any mountainous technical barriers to doing it.

Project-Specific Tools

Most medium-sized 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:

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

30.6. Tool Fantasyland

For decades, tool vendors and industry pundits have promised that the tools needed to eliminate programming are just over the horizon. The first, and perhaps most ironic, tool to receive this moniker was Fortran. Fortran or "Formula Translation Language" was conceived so that scientists and engineers could simply type in formulas, thus supposedly eliminating the need for programmers.

Cross-Reference

Tool availability depends partly on the maturity of the technical environment. For more on this, see [Section 4.3, "Your Location on the Technology Wave."](#)

Fortran did succeed in making it possible for scientists and engineers to write programs, but from our vantage point today, Fortran appears to be a comparatively lowlevel programming language. It hardly eliminated the need for programmers, and what the industry experienced with Fortran is indicative of progress in the software industry as a whole.

The software industry constantly develops new tools that reduce or eliminate some of the most tedious aspects of programming: details of laying out source statements; steps needed to edit, compile, link, and run a program; work needed to find mismatched braces; the number of steps needed to create standard message boxes; and so on. As each of these new tools begins to demonstrate incremental gains in productivity, pundits extrapolate those gains out to infinity, assuming that the gains will eventually "eliminate the need for programming." But what's happening in reality is that each new programming innovation arrives with a few blemishes. As time goes by, the blemishes are removed and that innovation's full potential is realized. However, once the fundamental tool concept is realized, further gains are achieved by stripping away the accidental difficulties that were created as side effects of creating the new tool. Elimination of these accidental difficulties does not increase productivity per se; it simply eliminates the "one step back" from the typical "two steps forward, one step back" equation.

Over the past several decades, programmers have seen numerous tools that were supposed to eliminate programming. First it was third-generation languages. Then it was fourth generation languages. Then it was automatic programming. Then it was CASE tools. Then it was visual programming. Each of these advances spun off valuable, incremental improvements to computer programming—and collectively they have made programming unrecognizable to anyone who learned programming before these advances. But none of these innovations succeeded in eliminating programming.

The reason for this dynamic is that, at its essence, programming is fundamentally hard—even with good tool support. No matter what tools are available, programmers will have to wrestle with the messy real world; we will have to think rigorously about sequences, dependencies, and exceptions; and we'll have to deal with end users who can't make up their minds. We will always have to wrestle with ill-defined interfaces to other software and hardware, and we'll have to account for regulations, business rules, and other sources of complexity that arise from outside the world of computer programming.

Cross-Reference

Reasons for the difficulty of programming are described in "[Accidental and Essential Difficulties](#)" in [Section 5.2](#).

We will always need people who can bridge the gap between the real-world problem to be solved and the computer that is supposed to be solving the problem. These people will be called programmers regardless of whether we're manipulating machine registers in assembler or dialog boxes in Microsoft Visual Basic. As long as we have computers, we'll need people who tell the computers what to do, and that activity will be called programming.

When you hear a tool vendor claim "This new tool will eliminate computer programming," run! Or at least smile to yourself at the vendor's naive optimism.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/3098

Take a look at these additional resources for more on programming tools:

cc2e.com/3005

<http://www.sdmagazine.com/jolts>. Software Development Magazine's annual Jolt Productivity award website is a good source of information about the best current tools.

Hunt, Andrew and David Thomas. *The Pragmatic Programmer*. Boston, MA: Addison-Wesley, 2000. Section 3 of this book provides an in-depth discussion of programming tools, including editors, code generators, debuggers, source-code control, and related tools.

cc2e.com/3012

Vaughn-Nichols, Steven. "Building Better Software with Better Tools," *IEEE Computer*, September 2003, pp. 12–14. This article surveys tool initiatives led by IBM, Microsoft Research, and Sun Research.

Glass, Robert L. *Software Conflict: Essays on the Art and Science of Software Engineering*. Englewood Cliffs, NJ: Yourdon Press, 1991. The chapter titled "Recommended: A Minimum Standard Software Toolset" provides a thoughtful counterpoint to the moretools-is-better view. Glass argues for the identification of a minimum set of tools that should be available to all developers and proposes a starting kit.

Jones, Capers. *Estimating Software Costs*. New York, NY: McGraw-Hill, 1998.

Boehm, Barry, et al. *Software Cost Estimation with Cocomo II*. Reading, MA: Addison-Wesley, 2000. Both the Jones and the Boehm books devote sections to the impact of tool use on productivity.

cc2e.com/3019

Checklist: Programming Tools

- Do you have an effective IDE?
- Does your IDE support integration with source-code control; build, test, and debugging tools; and other useful functions?
- Do you have tools that automate common refactorings?
- Are you using version control to manage source code, content, requirements, designs, project plans, and other project artifacts?
- If you're working on a very large project, are you using a data dictionary or some other central repository that contains authoritative descriptions of each class used in the system?

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- Programmers sometimes overlook some of the most powerful tools for years before discovering them.
- Good tools can make your life a lot easier.
- Tools are readily available for editing, analyzing code quality, refactoring, version control, debugging, testing, and code tuning.
- You can make many of the special-purpose tools you need.
- Good tools can reduce the more tedious aspects of software development, but they can't eliminate the need for programming, although they will continue to reshape what we mean by "programming."

Part VII: Software Craftsmanship

[In this part:](#)

[Chapter 31. Layout and Style](#)

[Chapter 32. Self-Documenting Code](#)

[Chapter 33. Personal Character](#)

[Chapter 34. Themes in Software Craftsmanship](#)

[Chapter 35. Where to Find More Information](#)

[◀ PREVIOUS](#)[< Free Open Study >](#)[NEXT ▶](#)**In this part:**[Chapter 31: Layout and Style](#)[Chapter 32: Self-Documenting Code](#)[Chapter 33: Personal Character](#)[Chapter 34: Themes in Software Craftsmanship](#)[Chapter 35: Where to Find More Information](#)[◀ PREVIOUS](#)[< Free Open Study >](#)[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 31. Layout and Style

cc2e.com/3187

Contents

- [Layout Fundamentals page 730](#)
- [Layout Techniques page 736](#)
- [Layout Styles page 738](#)
- [Laying Out Control Structures page 745](#)
- [Laying Out Individual Statements page 753](#)
- [Laying Out Comments page 763](#)
- [Laying Out Routines page 766](#)
- [Laying Out Classes page 768](#)

Related Topics

- Self-documenting code: [Chapter 32](#)
- Code formatting tools: "Editing" in [Section 30.2](#)

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.

You might not agree with everything you read here, but my point is less to win your agreement than to convince you

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

31.1. Layout Fundamentals

This section explains the theory of good layout. The rest of the chapter explains the practice.

Layout Extremes

Consider the routine shown in [Listing 31-1](#):



Listing 31-1: Java layout example #1

```
/* Use the insertion sort technique to sort the "data" array
in

→ ascending order.

This routine assumes that data[ firstElement ] is not the
first

→ element in data and

that data[ firstElement-1 ] can be accessed. */ public void

→ InsertionSort( int[]

data, int firstElement, int lastElement ) { /* Replace
element at

→ lower boundary

with an element guaranteed to be first in a sorted list. */
int

→ lowerBoundary =

data[ firstElement-1 ]; data[ firstElement-1 ] = SORT_MIN; /*

The

→ elements in

positions firstElement through sortBoundary-1 are always
sorted.

→ In each pass

through the loop, sortBoundary is increased, and the element
at

→ the position of the

new sortBoundary probably isn't in its sorted place in the
array,

→ so it's inserted

into the proper place somewhere between firstElement and

→ sortBoundary. */ for (
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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.

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](#).

Breaking a book into chapters, paragraphs, and sentences shows a reader how to mentally organize a topic. If the organization isn't evident, the reader has to provide the organization, which puts a much greater burden on the reader and adds the possibility that the reader may never figure out how the topic is organized.

The information contained in a program is denser than the information contained in most books. Whereas you might read and understand a page of a book in a minute or two, most programmers can't read and understand a naked program listing at anything close to that rate. A program should give more organizational clues than a book, not fewer.

Grouping From the other side of the looking glass, white space is grouping, making sure that related statements are grouped together.

In writing, thoughts are grouped into paragraphs. A well-written paragraph contains only sentences that relate to a particular thought. It shouldn't contain extraneous sentences. Similarly, a paragraph of code should contain statements that accomplish a single task and that are related to each other.

Blank lines Just as it's important to group related statements, it's important to separate unrelated statements from each other. The start of a new paragraph in English is identified with indentation or a blank line. The start of a new paragraph of code should be identified with a blank line.

Using blank lines is a way to indicate how a program is organized. You can use them to divide groups of related statements into paragraphs, to separate routines from one another, and to highlight comments.



Although this particular statistic may be hard to put to work, a study by Gorla, Benander, and Benander found that the optimal number of blank lines in a program is about 8 to 16 percent. Above 16 percent, debug time increases dramatically (1990).

Indentation Use indentation to show the logical structure of a program. As a rule, you should indent statements under the statement to which they are logically subordinate.



Indentation has been shown to be correlated with increased programmer comprehension. The article "Program Indentation and Comprehensibility" reported that several studies found correlations between indentation and improved comprehension (Miarra et al. 1983). Subjects scored 20 to 30 percent higher on a test of comprehension when programs had a

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

31.3. Layout Styles

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:

- Pure blocks
- Emulating pure blocks
- Using begin-end pairs (braces) to designate block boundaries
- Endline layout

Pure Blocks

Much of the layout controversy stems from the inherent awkwardness of the more popular programming languages. A well-designed language has clear block structures that lend themselves to a natural indentation style. In Visual Basic, for example, each control construct has its own terminator and you can't use a control construct without using the terminator. Code is blocked naturally. Some examples in Visual Basic are shown in [Listing 31-6](#), [Listing 31-7](#), and [Listing 31-8](#):

Listing 31-6. Visual Basic example of a pure if block

```
If pixelColor = Color_Red Then  
    statement1  
    statement2  
    ...  
End If
```

Listing 31-7. Visual Basic example of a pure while block

```
While pixelColor = Color_Red  
    statement1  
    statement2  
    ...  
Wend
```

Listing 31-8. Visual Basic example of a pure case block

```
Select Case pixelColor  
    Case Color_Red  
        statement1  
        statement2
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

31.4. Laying Out Control Structures

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.

Cross-Reference

For details on documenting control structures, see "[Commenting Control Structures](#)" in [Section 32.5](#). For a discussion of other aspects of control structures, see [Chapters 14](#) through [19](#).

Fine Points of Formatting Control-Structure Blocks

Working with control-structure blocks requires attention to some fine details. Here are some guidelines:

Avoid unindented begin-end pairs In the style shown in [Listing 31-24](#), the begin-end pair is aligned with the control structure, and the statements that begin and end enclose are indented under begin.

Listing 31-24. Java example of unindented begin-end pairs

```
for ( int i = 0; i < MAX_LINES; i++ )           <-- 1
{
    <-- 1
    ReadLine( i );      <-- 2
    ProcessLine( i );
}           <-- 3
```

(1)The begin is aligned with the for.

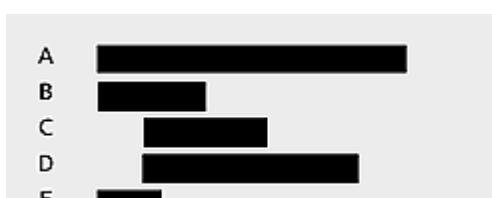
(2)The statements are indented under begin.

(3)The end is aligned with the for.

Although this approach looks fine, it violates the Fundamental Theorem of Formatting; it doesn't show the logical structure of the code. Used this way, the begin and end aren't part of the control construct, but they aren't part of the statement(s) after it either.

[Listing 31-25](#) is an abstract view of this approach:

Listing 31-25. Abstract example of misleading indentation



[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

31.5. Laying Out Individual Statements

This section explains many ways to improve individual statements in a program.

Statement Length

A common and somewhat outdated rule is to limit statement line length to 80 characters. Here are the reasons:

Cross-Reference

For details on documenting individual statements, see "[Commenting Individual Lines](#)" in [Section 32.5](#).

- 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, especially when code is printed "2 up" (2 pages of code to each physical printout page).

With larger screens, narrow typefaces, and landscape mode, the 80-character limit appears increasingly arbitrary. 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 I dare you to read this.

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
```

Some software artists might recommend enhancing this particular expression with additional spaces to emphasize its logical structure, this way:

```
while ( pathName[ startPath + position ] <> ';' ) and
```

```
( ( startPath + position ) < length( pathName ) ) do
```

This is fine, although the first use of spaces was sufficient to ensure readability. Extra spaces hardly ever hurt, however, so be generous with them.

Use spaces to make array references readable

The expression

```
grossRate[census[groupId].gender,census[groupId].ageGroup]
```

is no more readable than the earlier dense while expression. Use spaces around each index in the array to make the

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

31.6. Laying Out Comments

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.

Cross-Reference

For details on other aspects of comments, see [Chapter 32, "Self-Documenting Code."](#)

Indent a comment with its corresponding code Visual indentation is a valuable aid to understanding a program's logical structure, and good comments don't interfere with the visual indentation. For example, what is the logical structure of the routine shown in [Listing 31-58](#)?



Visual Basic example of poorly indented comments

Listing 31-58.

```
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 )
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

31.7. Laying Out 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.

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 by Using the PPP](#)." For a discussion of the differences between good and bad routines, see [Chapter 7](#), "[High-Quality 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 31-62](#) shows two examples of routine headers with no conscious layout:

Listing 31-62. C++ examples of routine headers with no conscious layout

```
bool ReadEmployeeData(int maxEmployees, EmployeeList *employees,  
                      EmployeeFile *inputFile, int *employeeCount, bool *isInputError)  
...  
  
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?

The second approach in routine-header layout is the endline layout, which usually works all right. [Listing 31-63](#) shows the same routine headers reformatted:

Listing 31-63. C++ example of routine headers with mediocre endline layout

```
bool ReadEmployeeData( int           maxEmployees,  
                      EmployeeList *employees,  
                      EmployeeFile  *inputFile,  
                      int           *employeeCount,  
                      bool          *isInputError )  
...  
  
void InsertionSort( SortArray   data,  
                    int         firstElement,  
                    int         lastElement )
```

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 that of

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

31.8. Laying Out Classes

This section presents guidelines for laying out code in classes. The first subsection describes how to lay out the class interface. The second subsection describes how to lay out the class implementations. The final subsection discusses laying out files and programs.

Laying Out Class Interfaces

In laying out class interfaces, the convention is to present the class members in the following order:

Cross-Reference

For details on documenting classes, see "[Commenting Classes, Files, and Programs](#)" in [Section 32.5](#). For a discussion of the differences between good and bad classes, see [Chapter 6](#), "[Working Classes](#)."

1. Header comment that describes the class and provides any notes about the overall usage of the class
2. Constructors and destructors
3. Public routines
4. Protected routines
5. Private routines and member data

Laying Out Class Implementations

Class implementations are generally laid out in this order:

1. Header comment that describes the contents of the file the class is in
2. Class data
3. Public routines
4. Protected routines
5. Private routines

If you have more than one class in a file, identify each class clearly Routines that are related should be grouped together into classes. A reader scanning your code should be able to tell easily which class is which. Identify each class clearly by using several blank lines between it and the classes next to it. A class is like a chapter in a book. In a book, you start each chapter on a new page and use big print for the chapter title. Emphasize the start of each class similarly. An example of separating classes is shown in [Listing 31-66](#):

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/3101

Most programming textbooks say a few words about layout and style, but thorough discussions of programming style are rare; discussions of layout are rarer still. The following books talk about layout and programming style:

Kernighan, Brian W. and Rob Pike. *The Practice of Programming* Reading, MA: Addison-Wesley, 1999. [Chapter 1](#) of this book discusses programming style focusing on C and C++.

Vermeulen, Allan, et al. *The Elements of Java Style*. Cambridge University Press, 2000.

Misfeldt, Trevor, Greg Bumgardner, and Andrew Gray. *The Elements of C++ Style*. Cambridge University Press, 2004.

Kernighan, Brian W., and P. J. Plauger. *The Elements of Programming Style*, 2d ed. New York, NY: McGraw-Hill, 1978. This is the classic book on programming style—the first in the genre of programming-style books.

For a substantially different approach to readability, take a look at the following book:

Knuth, Donald E. *Literate Programming*. Cambridge University Press, 2001. This is a collection of papers describing the "literate programming" approach of combining a programming language and a documentation language. Knuth has been writing about the virtues of literate programming for about 20 years, and in spite of his strong claim to the title Best Programmer on the Planet, literate programming isn't catching on. Read some of his code to form your own conclusions about the reason.

Key Points

- The first priority of visual layout is to illuminate the logical organization of the code. Criteria used to assess whether that priority is achieved include accuracy, consistency, readability, and maintainability.
- Looking good is secondary to the other criteria—a distant second. If the other criteria are met and the underlying code is good, however, the layout will look fine.
- Visual Basic has pure blocks and the conventional practice in Java is to use pureblock style, so you can use a pure-block layout if you program in those languages. In C++, either pure-block emulation or begin-end block boundaries work well.
- Structuring code is important for its own sake. The specific convention you follow is less important than the fact that you follow some convention consistently. A layout convention that's followed inconsistently can actually hurt readability.
- Many aspects of layout are religious issues. Try to separate objective preferences from subjective ones. Use explicit criteria to help ground your discussions about style preferences.

Chapter 32. Self-Documenting Code

cc2e.com/3245

Contents

- [External Documentation page 777](#)
- [Programming Style as Documentation page 778](#)
- [To Comment or Not to Comment page 781](#)
- [Keys to Effective Comments page 785](#)
- [Commenting Techniques page 792](#)
- [IEEE Standards page 813](#)

Related Topics

- Layout: [Chapter 31](#)
- The Pseudocode Programming Process: [Chapter 9](#)
- Working classes: [Chapter 6](#)
- High-quality routines: [Chapter 7](#)
- Programming as communication: [Sections 35.5](#) 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."

Code as if whoever maintains your program is a violent psychopath who knows where you live.

—Anonymous

32.1. External Documentation

Documentation on a software project consists of information both inside the sourcecode 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 the problem definition, requirements, and architecture activities.

Cross-Reference

For more on external documentation, see [Section 32.6, "IEEE Standards."](#)

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, although it could also mean a package or a component. 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.

Further Reading

For a detailed description, see "The Unit Development Folder (UDF): An Effective Management Tool for Software Development" (Ingrassia 1976) or "The Unit Development Folder (UDF): A Ten-Year Perspective" (Ingrassia 1987).

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 developers' notes collected into a UDF. And sometimes—often—it exists only in the code itself.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

32.2. Programming Style as Documentation

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.

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.

Here's a code fragment with poor style:



Java Example of Poor Documentation Resulting from Bad Programming Style

```
for ( i = 2; i <= num; i++ ) {  
    meetsCriteria[ i ] = true;  
}  
  
for ( i = 2; i <= num / 2; i++ ) {  
    j = i + i;  
  
    while ( j <= num ) {  
        meetsCriteria[ j ] = false;  
        j = j + i;  
    }  
}  
  
for ( i = 1; i <= num; i++ ) {  
    if ( meetsCriteria[ i ] ) {  
        System.out.println ( i + " meets criteria." );  
    }  
}
```

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 = 2; primeCandidate <= num; primeCandidate++ ) {
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

32.3. To Comment or Not to Comment

Comments are easier to write poorly than well, and commenting can be more damaging than helpful. The heated discussions over the virtues of commenting often sound like philosophical debates over moral virtues, which makes me think that if Socrates had been a computer programmer, he and his students might have had the following discussion.

The Comments

Characters:

THRASYMACHUS A green, theoretical purist who believes everything he reads

CALLICLES A battle-hardened veteran from the old school—a "real" programmer

GLAUCON A young, confident, hot-shot computer jock

ISMENE A senior programmer tired of big promises, just looking for a few practices that work

SOCRATES The wise old programmer

Setting:

END OF THE TEAM'S DAILY STANDUP MEETING

"Does anyone have any other issues before we get back to work?" Socrates asked.

"I want to suggest a commenting standard for our projects," Thrasymachus said. "Some of our programmers barely comment their code, and everyone knows that code without comments is unreadable."

"You must be fresher out of college than I thought," Callicles responded. "Comments are an academic panacea, but everyone who's done any real programming knows that comments make the code harder to read, not easier. English is less precise than Java or Visual Basic and makes for a lot of excess verbiage. Programming-language statements are short and to the point. If you can't make the code clear, how can you make the comments clear? Plus, comments get out of date as the code changes. If you believe an out-of-date comment, you're sunk."

"I agree with that," Glaucon joined in. "Heavily commented code is harder to read because it means more to read. I already have to read the code; why should I have to read a lot of comments, too?"

"Wait a minute," Ismene said, putting down her coffee mug to put in her two drachmas' worth. "I know that commenting can be abused, but good comments are worth their weight in gold. I've had to maintain code that had comments and code that didn't, and I'd rather maintain code with comments. I don't think we should have a standard that says use one comment for every x lines of code, but we should encourage everyone to comment."

"If comments are a waste of time, why does anyone use them, Callicles?" Socrates asked.

"Either because they're required to or because they read somewhere that they're useful. No one who's thought about it could ever decide they're useful."

"Ismene thinks they're useful. She's been here three years, maintaining your code without comments and other code with comments, and she prefers the code with comments. What do you make of that?"

"Comments are useless because they just repeat the code in a more verbose—"



"Wait right there," Thrasymachus interrupted. "Good comments don't repeat the code or explain it. They clarify its intent. Comments should explain, at a higher level of abstraction than the code, what you're trying to do."

KEY POINT

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

32.4. Keys to Effective Comments

What does the following routine do?

As long as there are illdefined goals, bizarre bugs, and unrealistic schedules, there will be Real Programmers willing to jump in and Solve The Problem, saving the documentation for later. Long live Fortran!

—Ed Post

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 );
```

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:

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

32.5. Commenting Techniques

Commenting is amenable to several different techniques depending on the level to which the comments apply: program, file, routine, paragraph, or individual line.

Commenting Individual Lines

In good code, the need to comment individual lines of code is rare. Here are two possible reasons a line of code would need a comment:

- 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.

Here are some guidelines for commenting a line of code:

Avoid self-indulgent comments Many years ago, I heard the story of a maintenance programmer who was called out of bed to fix a malfunctioning program. The program's author had left the company and couldn't be reached. The maintenance programmer hadn't worked on the program before, and after examining the documentation carefully, he found only one comment. It looked like this:

```
MOV AX, 723h ; R. I. P. L. V. B.
```

After working with the program through the night and puzzling over the comment, the programmer made a successful patch and went home to bed. Months later, he met the program's author at a conference and found out that the comment stood for "Rest in peace, Ludwig van Beethoven." Beethoven died in 1827 (decimal), which is 723 (hexadecimal). The fact that 723h was needed in that spot had nothing to do with the comment. Aaarrrrghhhh!

Endline Comments and Their Problems

Endline comments are comments that appear at the ends of lines of code:

Visual Basic Example of Endline Comments

```
For employeeId = 1 To employeeCount

    GetBonus( employeeId, employeeType, bonusAmount )

    If employeeType = EmployeeType_Manager Then

        PayManagerBonus( employeeId, bonusAmount ) ' pay full amount

    Else

        If employeeType = EmployeeType_Programmer Then

            If bonusAmount >= MANAGER_APPROVAL_LEVEL Then

                PayProgrammerBonus( employeeId, StdAmt() ) ' pay std. amount

            Else

                PayProgrammerBonus( employeeId, bonusAmount ) ' pay full amount

            End If

        End If

    End If
```

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

32.6. IEEE Standards

For documentation beyond the source-code level, valuable sources of information are the IEEE (Institute for Electric and Electrical Engineers) Software Engineering Standards. IEEE standards are developed by groups composed of practitioners and academicians who are expert in a particular area. Each standard contains a summary of the area covered by the standard and typically contains the outline for the appropriate documentation for work in that area.

Several national and international organizations participate in standards work. The IEEE is a group that has taken the lead in defining software engineering standards. Some standards are jointly adopted by ISO (International Standards Organization), EIA (Electronic Industries Alliance), or IEC (International Engineering Consortium).

Standards names are composed of the standards number, the year the standard was adopted, and the name of the standard. So, IEEE/EIA Std 12207-1997, Information Technology—Software Life Cycle Processes, refers to standard number 12207.2, which was adopted in 1997 by the IEEE and EIA.

Here are some of the national and international standards most applicable to software projects:

cc2e.com/3266

The top-level standard is ISO/IEC Std 12207, Information Technology—Software Life Cycle Processes, which is the international standard that defines a life-cycle framework for developing and managing software projects. This standard was adopted in the United States as IEEE/EIA Std 12207, Information Technology—Software Life Cycle Processes.

Software-Development Standards

Here are software-development standards to consider:

IEEE Std 830-1998, Recommended Practice for Software Requirements Specifications

IEEE Std 1233-1998, Guide for Developing System Requirements Specifications

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

IEEE Std 828-1998, Standard for Software Configuration Management Plans

IEEE Std 1063-2001, Standard for Software User Documentation

IEEE Std 1219-1998, Standard for Software Maintenance

Software Quality-Assurance Standards

cc2e.com/3280

And here are software quality-assurance standards:

IEEE Std 730-2002, Standard for Software Quality Assurance Plans

IEEE Std 1028-1997, Standard for Software Reviews

IEEE Std 1008-1987 (R1993), Standard for Software Unit Testing

IEEE Std 829-1998, Standard for Software Test Documentation

IEEE Std 1061-1998, Standard for a Software Quality Metrics Methodology

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Additional Resources

cc2e.com/3208

In addition to the IEEE standards, numerous other resources are available on program documentation.

Spinellis, Diomidis. *Code Reading: The Open Source Perspective*. Boston, MA: 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.

cc2e.com/3215

SourceForge.net. For decades, a perennial problem in teaching software development has been finding life-size examples of production code to share with students. Many people learn quickest from studying real-life examples, but most life-size 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 opensource 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 the website to see much larger real-world examples than *Code Complete*, Second Edition, is able to show in its short code examples. Junior programmers who haven't previously seen extensive examples of production code will find this website especially valuable as a source of both good and bad coding practices.

I wonder how many great novelists have never read someone else's work, how many great painters have never studied another's brush strokes, how many skilled surgeons never learned by looking over a colleague's shoulder.... And yet that's what we expect programmers to do.

—Dave Thomas

cc2e.com/3222

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 by 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:

McConnell, Steve. *Software Project Survival Guide*. Redmond, WA: Microsoft Press, 1998. This book describes the documentation required by a medium-sized businesscritical project. A related website provides numerous related document templates.

cc2e.com/3229

<http://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.

cc2e.com/3236

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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- 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.
- 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.
- 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.
- Comments should say things about the code that the code can't say about itself—at the summary level or the intent level.
- Some commenting styles require a lot of tedious clerical work. Develop a style that's easy to maintain.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Chapter 33. Personal Character

cc2e.com/3313

Contents

- [Isn't Personal Character Off the Topic? page 820](#)
- [Intelligence and Humility page 821](#)
- [Curiosity page 822](#)
- [Intellectual Honesty page 826](#)
- [Communication and Cooperation page 828](#)
- [Creativity and Discipline page 829](#)
- [Laziness page 830](#)
- [Characteristics That Don't Matter As Much As You Might Think page 830](#)
- [Habits page 833](#)

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. Titles such as The Psychology of Bridge Construction and "Exploratory Experiments in Attorney Behavior" might seem absurd, but 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 loadbearing 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 programming is building air castles—it's one of the most purely mental activities you can do.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

33.1. Isn't Personal Character Off the Topic?

The intense inwardness of programming makes personal character especially important. You know how difficult it is to put in eight concentrated hours in one day. You've probably had the experience of being burned out one day from concentrating too hard the day before or burned out one month from concentrating too hard the month before. You've probably had days on which you've worked well from 8:00 A.M. to 2:00 P.M. and then felt like quitting. You didn't quit, though; you pushed on from 2:00 P.M. to 5:00 P.M. and then spent the rest of the week fixing what you wrote from 2:00 to 5:00.

Programming work is essentially unsupervisable because no one ever really knows what you're working on. We've all had projects in which we spent 80 percent of the time working on a small piece we found interesting and 20 percent of the time building the other 80 percent of the program.

Your employer can't force you to be a good programmer; a lot of times your employer isn't even in a position to judge whether you're good. If you want to be great, you're responsible for making yourself great. It's a matter of your personal character.



Once you decide to make yourself a superior programmer, the potential for improvement is huge. Study after study has found differences on the order of 10 to 1 in the time required to create a program. They have also found differences on the order of 10 to 1 in the time required to debug a program and 10 to 1 in the resulting size, speed, error rate, and number of errors detected (Sackman, Erikson, and Grant 1968; Curtis 1981; Mills 1983; DeMarco and Lister 1985; Curtis et al. 1986; Card 1987; Valett and McGarry 1989).

You can't do anything about your intelligence, so the classical wisdom goes, but you can do something about your character. And it turns out that character is the more decisive factor in the makeup of a superior programmer.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

33.2. Intelligence and Humility

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.

We become authorities and experts in the practical and scientific spheres by so many separate acts and hours of work. If a person keeps faithfully busy each hour of the working day, he can count on waking up some morning to find himself one of the competent ones of his generation.

—William James

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 \("Design in Construction"\)](#) 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.

The purpose of many good programming practices is to reduce the load on your gray cells. Here are a few examples:

- The point of "decomposing" a system is to make it simpler to understand. (See "[Levels of Design](#)" in [Section 5.2](#) 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 lowlevel implementation details reduces your mental workload.
- Using conventions of all sorts frees your brain from the relatively mundane aspects of programming, which offer little payback.

You might think that the high road would be to develop better mental abilities so that you wouldn't need these programming crutches. You might think that a programmer who uses mental crutches is taking the low road. Empirically, however, it's been shown that humble programmers who compensate for their fallibilities write code that's easier for themselves and others to understand and that has fewer errors. The real low road is the road of errors and delayed schedules.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

33.3. Curiosity

Once you admit that your brain is too small to understand most programs and you realize that effective programming is a search for ways to offset that fact, you begin a career-long search for ways to compensate. In the development of a superior programmer, curiosity about technical subjects must be a priority. The relevant technical information changes continually. Many Web programmers have never had to program in Microsoft Windows, and many Windows programmers never had to deal with DOS or UNIX or punch cards. Specific features of the technical environment change every 5 to 10 years. If you aren't curious enough to keep up with the changes, you might find yourself down at the old-programmers' home playing cards with T-Bone Rex and the Brontosaurus sisters.

Programmers are so busy working they often don't have time to be curious about how they might do their jobs better. If this is true for you, you're not alone. The following subsections describe a few specific actions you can take to exercise your curiosity and make learning a priority.

Build your awareness of the development process The more aware you are of the development process, whether from reading or from your own observations about software development, the better position you're in to understand changes and to move your group in a good direction.

Cross-Reference

For a fuller discussion of the importance of process in software development, see [Section 34.2, "Pick Your Process."](#)

If your workload consists entirely of short-term assignments that don't develop your skills, be dissatisfied. If you're working in a competitive software market, half of what you now need to know to do your job will be out of date in three years. If you're not learning, you're turning into a dinosaur.



You're in too much demand to spend time working for management that doesn't have your interests in mind. Despite some ups and downs and some jobs moving overseas, the average number of software jobs available in the U.S. is expected to increase dramatically between 2002 and 2012. Jobs for systems analysts are expected to increase by about 60 percent and for software engineers by about 50 percent. For all computer-job categories combined, about 1 million new jobs will be created beyond the 3 million that currently exist (Hecker 2001, BLS 2004). If you can't learn at your job, find a new one.

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.

Cross-Reference

Several key aspects of programming revolve around the idea of experimentation. For details, see "[Experimentation](#)" in [Section 34.9](#).

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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

33.4. Intellectual Honesty

Part of maturing as a programming professional is developing an uncompromising sense of intellectual honesty. Intellectual honesty commonly manifests itself in several ways:

- Refusing to pretend you're an expert when you're not
- Readily admitting your mistakes
- Trying to understand a compiler warning rather than suppressing the message
- Clearly understanding your program—not compiling it to see if it works
- Providing realistic status reports
- Providing realistic schedule estimates and holding your ground when management asks you to adjust them

The first two items on this list—admitting that you don't know something or that you made a mistake—echo the theme of intellectual humility discussed earlier. How can you learn anything new if you pretend that you know everything already? You'd be better off pretending that you don't know anything. Listen to people's explanations, learn something new from them, and assess whether they know what they are talking about.

Be ready to quantify your degree of certainty on any issue. If it's usually 100 percent, that's a warning sign.

Refusing to admit mistakes is a particularly annoying habit. If Sally refuses to admit a mistake, she apparently believes that not admitting the mistake will trick others into believing that she didn't make it. The opposite is true. Everyone will 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.

Any fool can defend his or her mistakes—and most fools do.

—Dale Carnegie

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.

Pretending to understand compiler messages when you don't is another common blind spot. If you don't understand a compiler warning or if you think you know what it means but are too pressed for time to check it, guess what's really a waste of time? You'll probably end up trying to solve the problem from the ground up while the compiler waves the solution in your face. I've had several people ask for help in debugging programs. I'll ask if they have a clean compile, and they'll say yes. Then they'll start to explain the symptoms of the problem, and I'll say, "Hmmmm. That sounds like it would be an uninitialized pointer, but the compiler should have warned you about that." Then they'll say, "Oh yeah—it did warn about that. We thought it meant something else." It's hard to fool other people about your mistakes. It's even harder to fool the computer, so don't waste your time trying.

A related kind of intellectual sloppiness occurs when you don't quite understand your program and "just compile it to see if it works." One example is running the program to see whether you should use `<` or `<=`. In that situation, it doesn't really matter whether the program works because you don't understand it well enough to know why it works. Remember that testing can show only the presence of errors, not their absence. If you don't understand the program,

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

33.5. Communication and Cooperation

Truly excellent programmers learn how to work and play well with others. Writing readable code is part of being a team player. The computer probably reads your program as often as other people do, but it's a lot better at reading poor code than people are. As a readability guideline, keep the person who has to modify your code in mind. Programming is communicating with another programmer first and communicating with the computer second.

33.6. Creativity and Discipline

When I got out of school, I thought I was the best programmer in the world. I could write an unbeatable tic-tac-toe program, use five different computer languages, and create 1000-line programs that WORKED (really!). Then I got out into the Real World. My first task in the Real World was to read and understand a 200,000-line Fortran program and then speed it up by a factor of two. Any Real Programmer will tell you that all the Structured Coding in the world won't help you solve a problem like that—it takes actual talent.

—Ed Post

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. Can you imagine a website on which each page used different fonts, colors, text alignment, graphics styles, and navigation clues? The effect would be chaotic, not creative. 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:

Laziness: The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful, and document what you wrote so that you don't have to answer so many questions about it.

—Larry Wall

-
- Deferring an unpleasant task
-
- Doing an unpleasant task quickly to get it out of the way
-
- Writing a tool to do the unpleasant task so that you never have to do the task again

Some of these manifestations of laziness are better than others. The first is hardly ever beneficial. You've probably had the experience of spending several hours futzing with jobs that didn't really need to be done so that you wouldn't have to face a relatively minor job that you couldn't avoid. I detest data entry, and many programs require a small amount of data entry. I've been known to delay working on a program for days just to delay the inevitable task of entering several pages of numbers by hand. This habit is "true laziness." It manifests itself again in the habit of compiling a class to see if it works so that you can avoid the exercise of checking the class with your mind.

The small tasks are never as bad as they seem. If you develop the habit of doing them right away, you can avoid the procrastinating kind of laziness. This habit is "enlightened laziness"—the second kind of laziness. You're still lazy, but you're getting around the problem by spending the smallest possible amount of time on something that's unpleasant.

The third option is to write a tool to do the unpleasant task. This is "long-term laziness." It is undoubtedly the most productive kind of laziness (provided that you ultimately save time by having written the tool). In these contexts, a certain amount of laziness is beneficial.

When you step through the looking glass, you see the other side of the laziness picture. "Hustle" or "making an effort" doesn't have the rosy glow it does in high-school physical education class. Hustle is extra, unnecessary effort. It shows that you're eager but not that you're getting your work done. It's easy to confuse motion with progress, busy-ness with being productive. The most important work in effective programming is thinking, and people tend not to look busy when they're thinking. If I worked with a programmer who looked busy all the time, I'd assume that he was not a good programmer because he wasn't using his most valuable tool, his brain.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

33.8. Characteristics That Don't Matter As Much As You Might Think

Hustle isn't the only characteristic that you might admire in other aspects of your life but that doesn't work very well in software development.

Persistence

Depending on the situation, persistence can be either an asset or a liability. Like most value-laden concepts, it's identified by different words depending on whether you think it's a good quality or a bad one. If you want to identify persistence as a bad quality, you say it's "stubbornness" or "pigheadedness." If you want it to be a good quality, you call it "tenacity" or "perseverance."

Most of the time, persistence in software development is pigheadedness—it has little value. Persistence when you're stuck on a piece of new code is hardly ever a virtue. Try redesigning the class, try an alternative coding approach, or try coming back to it later. When one approach isn't working, that's a good time to try an alternative (Pirsig 1974).

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.

Cross-Reference

For a more detailed discussion of persistence in debugging, see "[Tips for Finding Defects](#)" in [Section 23.2](#).

It's hard to know when to give up, but it's essential that you ask. When you notice that you're frustrated, that's a good time to ask the question. Asking doesn't necessarily mean that it's time to give up, but it probably means that it's time to set some parameters on the activity: "If I don't solve the problem using this approach within the next 30 minutes, I'll take 10 minutes to brainstorm about different approaches and try the best one for the next hour."

Experience

The value of hands-on experience as compared to book learning is smaller in software development than in many other fields for several reasons. In many other fields, basic knowledge changes slowly enough that someone who graduated from college 10 years after you did probably learned the same basic material that you did. In software development, even basic knowledge changes rapidly. The person who graduated from college 10 years after you did probably learned twice as much about effective programming techniques. Older programmers tend to be viewed with suspicion not just because they might be out of touch with specific technology but because they might never have been exposed to basic programming concepts that became well known after they left school.

In other fields, what you learn about your job today is likely to help you in your job tomorrow. In software, if you can't shake the habits of thinking you developed while using your former programming language or the code-tuning techniques that worked on your old machine, your experience will be worse than none at all. A lot of software people spend their time preparing to fight the last war rather than the next one. If you can't change with the times, experience is more a handicap than a help.

Aside from the rapid changes in software development, people often draw the wrong conclusions from their experiences. It's hard to view your own life objectively. You can overlook key elements of your experience that would cause you to draw different conclusions if you recognized them. Reading studies of other programmers is helpful because the studies reveal other people's experience—filtered enough that you can examine it objectively.

People also put an absurd emphasis on the amount of experience programmers have. "We want a programmer with

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

33.9. Habits

The moral virtues, then, are engendered in us neither by nor contrary to nature...their full development in us is due to habit....Anything that we have to learn to do we learn by the actual doing of it....Men will become good builders as a result of building well and bad ones as a result of building badly....So it is a matter of no little importance what sort of habits we form from the earliest age—it makes a vast difference, or rather all the difference in the world. —Aristotle

Good habits matter because most of what you do as a programmer you do without consciously thinking about it. For example, at one time, you might have thought about how you wanted to format indented loops, but now you don't think about it again each time you write a new loop. You do it the way you do it out of habit. This is true of virtually all aspects of program formatting. When was the last time you seriously questioned your formatting style? Chances are good that if you've been programming for five years, you last questioned it four and a half years ago. The rest of the time you've relied on habit.

You have habits in many areas. For example, programmers tend to check loop indexes carefully and not to check assignment statements, which makes 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.

Cross-Reference

For details on errors in assignment statements, see "[Errors by Classification](#)" in [Section 22.4](#).

Study the quotation from Aristotle and substitute "programming virtues" for "moral virtues." He points out that you are not predisposed to either good or bad behavior but are constituted in such a way that you can become either a good or a bad programmer. The main way you become good or bad at what you do is by doing—builders by building and programmers by programming. What you do becomes habit, and over time your good and bad habits determine whether you're a good or a bad programmer.

Bill Gates says that any programmer who will ever be good is good in the first few years. After that, whether a programmer is good or not is cast in concrete (Lammers 1986). After you've been programming a long time, it's hard to suddenly start saying, "How do I make this loop faster?" or "How do I make this code more readable?" These are habits that good programmers develop early.

When you first learn something, learn it the right way. When you first do it, you're actively thinking about it and you still have an easy choice between doing it in a good way and doing it in a bad way. After you've done it a few times, you pay less attention to what you're doing and "force of habit" takes over. Make sure that the habits that take over are the ones you want to have.

What if you don't already have the most effective habits? How do you change a bad habit? If I had the definitive answer to that, I could sell self-help tapes on late-night TV. But here's at least part of an answer. You can't replace a bad habit with no habit at all. That's why people who suddenly stop smoking or swearing or overeating have such a hard time unless they substitute something else, like chewing gum. It's easier to replace an old habit with a new one than it is to eliminate one altogether. In programming, try to develop new habits that work. Develop the habits of writing a class in pseudocode before coding it and carefully reading the code before compiling it, for instance. You won't have to worry about losing the bad habits; they'll naturally drop by the wayside as new habits take their places.

Additional Resources

cc2e.com/3327

Following are additional resources on the human aspects of software development:

cc2e.com/3334

Dijkstra, Edsger. "The Humble Programmer." Turing Award Lecture. Communications of the ACM 15, no. 10 (October 1972): 859–66. This classic paper helped begin the inquiry into how much computer programming depends on the programmer's mental abilities. Dijkstra has persistently stressed the message that the essential task of programming is mastering the enormous complexity of computer science. He argues that programming is the only activity in which humans have to master nine orders of magnitude of difference between the lowest level of detail and the highest. This paper would be interesting reading solely for its historical value, but many of its themes sound fresh decades later. It also conveys a good sense of what it was like to be a programmer in the early days of computer science.

Weinberg, Gerald M. *The Psychology of Computer Programming*: Silver Anniversary Edition. New York, NY: Dorset House, 1998. This classic book contains a detailed exposition of the idea of egoless programming and of many other aspects of the human side of computer programming. It contains many entertaining anecdotes and is one of the most readable books yet written about software development.

Pirsig, Robert M. *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values*. William Morrow, 1974. Pirsig provides an extended discussion of "quality," ostensibly as it relates to motorcycle maintenance. Pirsig was working as a software technical writer when he wrote ZAMM, and his insightful comments apply as much to the psychology of software projects as to motorcycle maintenance.

Curtis, Bill, ed. *Tutorial: Human Factors in Software Development*. Los Angeles, CA: IEEE Computer Society Press, 1985. This is an excellent collection of papers that address the human aspects of creating computer programs. The 45 papers are divided into sections on mental models of programming knowledge, learning to program, problem solving and design, effects of design representations, language characteristics, error diagnosis, and methodology. If programming is one of the most difficult intellectual challenges that humankind has ever faced, learning more about human mental capacities is critical to the success of the endeavor. These papers about psychological factors also help you to turn your mind inward and learn about how you individually can program more effectively.

McConnell, Steve. *Professional Software Development*. Boston, MA: Addison-Wesley, 2004. [Chapter 7](#), "Orphans Preferred," provides more details on programmer personalities and the role of personal character.

Key Points

- Your personal character directly affects your ability to write computer programs.
-
- The characteristics that matter most are humility, curiosity, intellectual honesty, creativity and discipline, and enlightened laziness.
-
- The characteristics of a superior programmer have almost nothing to do with talent and everything to do with a commitment to personal development.
-
- Surprisingly, raw intelligence, experience, persistence, and guts hurt as much as they help.
-
- Many programmers don't actively seek new information and techniques and instead rely on accidental, on-the-job exposure to new information. If you devote a small percentage of your time to reading and learning about programming, after a few months or years you'll dramatically distinguish yourself from the programming mainstream.
-
- Good character is mainly a matter of having the right habits. To be a great programmer, develop the right habits and the rest will come naturally.

Chapter 34. Themes in Software Craftsmanship

cc2e.com/3444

Contents

- [Conquer Complexity page 837](#)
- [Pick Your Process page 839](#)
- [Write Programs for People First, Computers Second page 841](#)
- [Program into Your Language, Not in It page 843](#)
- [Focus Your Attention with the Help of Conventions page 844](#)
- [Program in Terms of the Problem Domain page 845](#)
- [Watch for Falling Rocks page 848](#)
- [Iterate, Repeatedly, Again and Again page 850](#)
- [Thou Shalt Rend Software and Religion Asunder page 851](#)

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 deemphasized abstract topics 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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

34.1. Conquer Complexity

The drive to reduce complexity is at the heart of software development, to such a degree that [Chapter 5, "Design in Construction,"](#) described managing complexity as Software's Primary Technical Imperative. 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:

Cross-Reference

For details on the importance of attitude in conquering complexity, see [Section 33.2, "Intelligence and Humility."](#)

- 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 nonlinearity 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 nonlinear control structure that's 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

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

34.2. Pick Your Process

A second major thread in this book is the idea that the process you use to develop software matters a surprising amount. On a small project, the talents of the individual programmer are the biggest influence on the quality of the software. Part of what makes an individual programmer successful is his or her choice of processes.

On projects with more than one programmer, organizational characteristics make a bigger difference than the skills of the individuals involved do. Even if you have a great team, its collective ability isn't simply the sum of the team members' individual abilities. The way in which people work together determines whether their abilities are added to each other or subtracted from each other. The process the team uses determines whether one person's work supports the work of the rest of the team or undercuts it.

One example of the way in which process matters is the consequence of not making requirements stable before you begin designing and coding. If you don't know what you're building, you can't very well create a superior design for it. If the requirements and subsequently the design change while the software is under development, the code must change too, which risks degrading the quality of the system.

Cross-Reference

For details on making requirements stable, see [Section 3.4, "Requirements Prerequisite."](#) For details on variations in development approaches, see [Section 3.2, "Determine the Kind of Software You're Working On."](#)

"Sure," you say, "but in the real world, you never really have stable requirements, so that's a red herring." Again, the process you use determines both how stable your requirements are and how stable they need to be. If you want to build more flexibility into the requirements, you can set up an incremental development approach in which you plan to deliver the software in several increments rather than all at once. This is an attention to process, and it's the process you use that ultimately determines whether your project succeeds or fails. [Table 3-1](#) in [Section 3.1](#) makes it clear that requirements errors are far more costly than construction errors, so focusing on that part of the process also affects cost and schedule.

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.

My message to the serious programmer is: spend a part of your working day examining and refining your own methods. Even though programmers are always struggling to meet some future or past deadline, methodological abstraction is a wise long-term investment.

—Robert W. Floyd

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, and 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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

34.3. Write Programs for People First, Computers Second

your program n. A maze of non sequiturs littered with clever-clever tricks and irrelevant comments. Compare MY PROGRAM.

my program n. A gem of algoristic 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.

—Stan Kelly-Bootle

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.

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:

- Comprehensibility
- Reviewability
- Error rate
- Debugging
- Modifiability
- Development time—a consequence of all of the above
- External quality—a consequence of all of the above

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.

In the early years of programming, a program was regarded as the private property of the programmer. One would no more think of reading a colleague's program unbidden than of picking up a love letter and reading it. This is essentially what a program was, a love letter from the programmer to the hardware, full of the intimate details known only to partners in an affair. Consequently, programs became larded with the pet names and verbal shorthand so popular with lovers who live in the blissful abstraction that assumes that theirs is the only existence in the universe. Such programs are unintelligible to those outside the partnership.

—Michael Marcotty

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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

34.4. Program into Your Language, Not in It

Don't limit your programming thinking only to the concepts that are supported automatically by your language. The best programmers think of what they want to do, and then they assess how to accomplish their objectives with the programming tools at their disposal.

Should you use a class member routine that's inconsistent with the class's abstraction just because it's more convenient than using one that provides more consistency? You should write code in a way that preserves the abstraction represented by the class's interface as much as possible. You don't need to use global data or gotos just because your language supports them. You can choose not to use those hazardous programming capabilities and instead use programming conventions to make up for weaknesses of the language. Programming using the most obvious path amounts to programming in a language rather than programming into a language; it's the programmer's equivalent of "If Freddie jumped off a bridge, would you jump off a bridge, too?" Think about your technical goals, and then decide how best to accomplish those goals by programming into your language.

Your language doesn't support assertions? Write your own assert() routine. It might not function exactly the same as a built-in assert(), but you can still realize most of assert()'s benefits by writing your own routine. Your language doesn't support enumerated types or named constants? That's fine; you can define your own enumerations and named constants with a disciplined use of global variables supported by clear naming conventions.

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'll 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.

34.5. Focus Your Attention with the Help of Conventions

A set of conventions is one of the intellectual tools used to manage complexity. Earlier chapters talk about specific conventions. This section lays out the benefits of conventions with many examples.

Cross-Reference

For an analysis of the value of conventions as they apply to program layout, see "[How Much Is Good Layout Worth?](#)" and "[Objectives of Good Layout](#)" in [Section 31.1](#).

Many of the details of programming are somewhat arbitrary. How many spaces do you indent a loop? How do you format a comment? How should you order class routines? Most of the questions like these have several correct answers. The specific way in which such a question is answered is less important than that it be answered consistently each time. Conventions save programmers the trouble of answering the same questions—making the same arbitrary decisions—again and again. On projects with many programmers, using conventions prevents the confusion that results when different programmers make the arbitrary decisions differently.

A convention conveys important information concisely. In naming conventions, a single character can differentiate among local, class, and global variables; capitalization can concisely differentiate among types, named constants, and variables. Indentation conventions can concisely show the logical structure of a program. Alignment conventions can indicate concisely that statements are related.

Conventions protect against known hazards. You can establish conventions to eliminate the use of dangerous practices, to restrict such practices to cases in which they're needed, or to compensate for their known hazards. You could eliminate a dangerous practice, for example, by prohibiting global variables or prohibiting multiple statements on a line. You could compensate for a hazardous practice by requiring parentheses around complicated expressions or requiring pointers to be set to NULL immediately after they're deleted to help prevent dangling pointers.

Conventions add predictability to low-level tasks. Having conventional ways of handling memory requests, error processing, input/output, and class interfaces adds a meaningful structure to your code and makes it easier for another programmer to figure out—as long as the programmer knows your conventions. As mentioned in an earlier chapter, one of the biggest benefits of eliminating global data is that you eliminate potential interactions among different classes and subsystems. A reader knows roughly what to expect from local and class data. But it's hard to tell when changing global data will break some bit of code four subsystems away. Global data increases the reader's uncertainty. With good conventions, you and your readers can take more for granted. The amount of detail that has to be assimilated will be reduced, and that in turn will improve program comprehension.

Conventions can compensate for language weaknesses. In languages that don't support named constants (such as Python, Perl, UNIX shell script, and so on), a convention can differentiate between variables intended to be both read and written and those that are intended to emulate read-only constants. Conventions for the disciplined use of global data and pointers are other examples of compensating for language weaknesses with conventions.

Programmers on large projects sometimes go overboard with conventions. They establish so many standards and guidelines that remembering them becomes a fulltime job. But programmers on small projects tend to go "underboard," not realizing the full benefits of intelligently conceived conventions. You should understand their real value and take advantage of them; you should use them to provide structure in areas in which structure is needed.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

34.6. Program in Terms of the Problem Domain

Another specific method of dealing with complexity is to work at the highest possible level of abstraction. One way of working at a high level of abstraction is to work in terms of the programming problem rather than the computer-science solution.

Top-level code shouldn't be filled with details about files and stacks and queues and arrays and characters whose parents couldn't think of better names for them than i, j, and k. Top-level code should describe the problem that's being solved. It should be packed with descriptive class names and routine calls that indicate exactly what the program is doing, not cluttered with details about opening a file as "read only." Top-level code shouldn't contain clumps of comments that say "i is a variable that represents the index of the record from the employee file here, and then a little later it's used to index the client account file there."

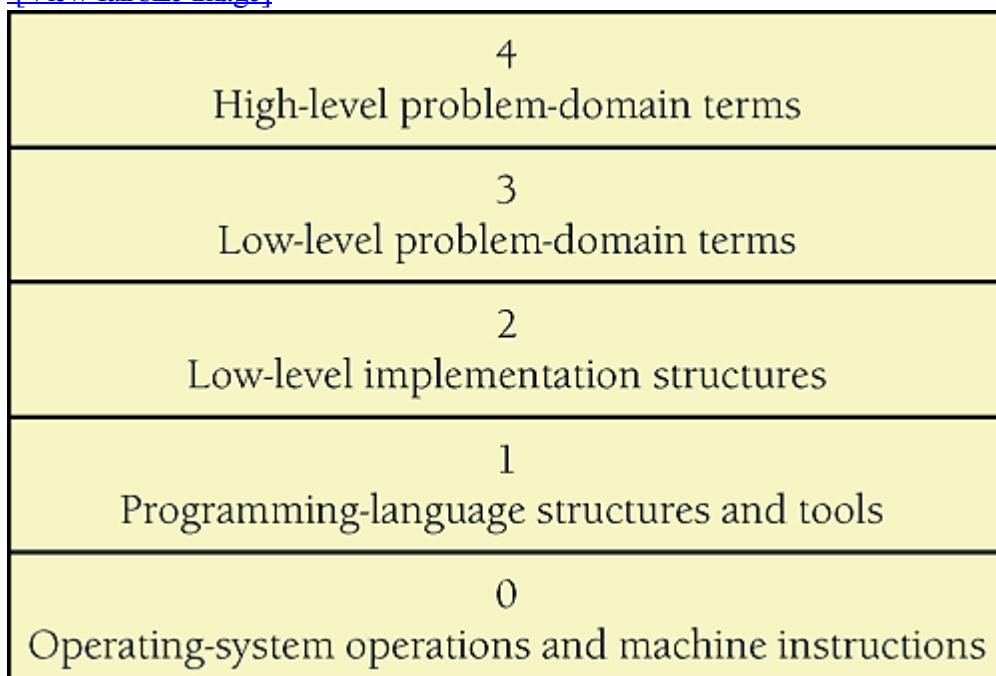
That's clumsy programming practice. At the top level of the program, you don't need to know that the employee data comes as records or that it's stored as a file. Information at that level of detail should be hidden. At the highest level, you shouldn't have any idea how the data is stored. Nor do you need to read a comment that explains what i means and that it's used for two purposes. You should see different variable names for the two purposes instead, and they should also have distinctive names such as employeeIndex and clientIndex.

Separating a Program into Levels of Abstraction

Obviously, you have to work in implementation-level terms at some level, but you can isolate the part of the program that works in implementation-level terms from the part that works in problem-domain terms. If you're designing a program, consider the levels of abstraction shown in [Figure 34-1](#).

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

[\[View full size image\]](#)



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.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

34.7. Watch for Falling Rocks

Programming is neither fully an art nor fully a science. As it's typically practiced, it's a "craft" that's somewhere between art and science. At its best, it's an engineering discipline that arises from the synergistic fusion of art and science (McConnell 2004). Whether art, science, craft, or engineering, it still takes plenty of individual judgment to create a working software product. And part of having good judgment in computer programming is being sensitive to a wide array of warning signs, subtle indications of problems in your program. Warning signs in programming alert you to the possibility of problems, but they're usually not as blatant as a road sign that says "[Watch for falling rocks.](#)"

When you or someone else says "This is really tricky code," that's a warning sign, usually of poor code. "Tricky code" is a code phrase for "bad code." If you think code is tricky, think about rewriting it so that it's not.

A class's having more errors than average is a warning sign. A few error-prone classes tend to be the most expensive part of a program. If you have a class that has had more errors than average, it will probably continue to have more errors than average. Think about rewriting it.

If programming were a science, each warning sign would imply a specific, welldefined corrective action. Because programming is still a craft, however, a warning sign merely points to an issue that you should consider. You can't necessarily rewrite tricky code or improve an error-prone class.

Just as an abnormal number of defects in a class warns you that the class has low quality, an abnormal number of defects in a program implies that your process is defective. A good process wouldn't allow error-prone code to be developed. It would include the checks and balances of architecture followed by architecture reviews, design followed by design reviews, and code followed by code reviews. By the time the code was ready for testing, most errors would have been eliminated. Exceptional performance requires working smart in addition to working hard. Lots of debugging on a project is a warning sign that implies people aren't working smart. Writing a lot of code in a day and then spending two weeks debugging it is not working smart.

You can use design metrics as another kind of warning sign. Most design metrics are heuristics that give an indication of the quality of a design. The fact that a class contains more than seven members doesn't necessarily mean that it's poorly designed, but it's a warning that the class is complicated. Similarly, more than about 10 decision points in a routine, more than three levels of logical nesting, an unusual number of variables, high coupling to other classes, or low class or routine cohesion should raise a warning flag. None of these signs necessarily means that a class is poorly designed, but the presence of any of them should cause you to look at the class skeptically.

Any warning sign should cause you to doubt the quality of your program. As Charles Saunders Peirce says, "Doubt is an uneasy and dissatisfied state from which we struggle to free ourselves and pass into the state of belief." Treat a warning sign as an "irritation of doubt" that prompts you to look for the more satisfied state of belief.

If you find yourself working on repetitious code or making similar modifications in several areas, you should feel "uneasy and dissatisfied," doubting that control has been adequately centralized in classes or routines. If you find it hard to create scaffolding for test cases because you can't use an individual class easily, you should feel the "irritation of doubt" and ask whether the class is coupled too tightly to other classes. If you can't reuse code in other programs because some classes are too interdependent, that's another warning sign that the classes are coupled too tightly.

When you're deep into a program, pay attention to warning signs that indicate that part of the program design isn't defined well enough to code. Difficulties in writing comments, naming variables, and decomposing the problem into cohesive classes with clear interfaces all indicate that you need to think harder about the design before coding. Wishy-washy names and difficulty in describing sections of code in concise comments are other signs of trouble. When the design is clear in your mind, the lowlevel details come easily.

Be sensitive to indications that your program is hard to understand. Any discomfort is a clue. If it's hard for you, it will be even harder for the next programmers. They'll appreciate the extra effort you make to improve it. If you're figuring out code instead of reading it, it's too complicated. If it's hard, it's wrong. Make it simpler.



If you want to take full advantage of warning signs, program in such a way that you create

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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, "Managing Construction,"](#) points out, schedule estimates during initial project planning 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.

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.

One definition of engineering is to do for a dime what anyone can do for a dollar. Iterating in the late stages is doing for two dollars what anyone can do for one dollar. Fred Brooks suggested that you "build one to throw away; you will, anyhow" (Brooks 1995). The trick of software engineering is to build the disposable parts as quickly and inexpensively as possible, which is the point of iterating in the early stages.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

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, or as a zealous avoidance of global data. Whatever the case, it's always inappropriate.

Software Oracles

Unfortunately, the zealous 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 difference, however, between disseminating a new methodology and selling software snake oil. The idea of technology transfer is poorly served by dogmatic methodology peddlers who try to convince you that their new one-size-fits-all, high-tech cow pies will solve all your problems. Forget everything you've already learned because this new method is so great it will improve your productivity 100 percent in everything!

Cross-Reference

For details on handling programming religion as a manager, see "[Religious Issues](#)" in [Section 28.5](#).

Rather than latching on to the latest miracle fad, use a mixture of methods. Experiment with the exciting, recent methods, but bank on the old and dependable ones.

Eclecticism

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. But software development isn't a deterministic process; 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.

Cross-Reference

For more on the difference between algorithmic and heuristic approaches, see [Section 2.2](#), "[How to Use Software Metaphors](#)." For information on eclecticism in design, see "[Iterate](#)" in [Section 5.4](#).

Adherence to a single method is also harmful in that it makes you force-fit the problem to the solution. If you decide on the solution method before you fully understand the problem, you act prematurely. Over-constrain the set of possible solutions, and you might rule out the most effective solution.

You'll be uncomfortable with any new methodology initially, and the advice that you avoid religion in programming isn't meant to suggest that you should stop using a new method as soon as you have a little trouble solving a problem with it. Give the new method a fair shake, but give the old methods their fair shakes, too.

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 presented here 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 tradeoffs among competing techniques. You can't make a tradeoff if you've prematurely limited your

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Key Points

- One primary goal of programming is managing complexity.
- The programming process significantly affects the final product.
- Team programming is more an exercise in communicating with people than in communicating with a computer. Individual programming is more an exercise in communicating with yourself than with a computer.
- When abused, a programming convention can be a cure that's worse than the disease. Used thoughtfully, a convention adds valuable structure to the development environment and helps with managing complexity and communication.
- Programming in terms of the problem rather than the solution helps to manage complexity.
- Paying attention to intellectual warning signs like the "irritation of doubt" is especially important in programming because programming is almost purely a mental activity.
- The more you iterate in each development activity, the better the product of that activity will be.
- Dogmatic methodologies and high-quality software development don't mix. Fill your intellectual toolbox with programming alternatives, and improve your skill at choosing the right tool for the job.

Chapter 35. Where to Find More Information

[cc2e.com/3560](#)

Contents

- [Information About Software Construction page 856](#)
- [Topics Beyond Construction page 857](#)
- [Periodicals page 859](#)
- [A Software Developer's Reading Plan page 860](#)
- [Joining a Professional Organization page 862](#)

Related Topics

-
- Web resources: <http://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 information 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 contexts within which programming goes on. And detailed references on languages, operating systems, environments, and hardware contain information that's useful for specific projects.

[cc2e.com/3581](#)

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. Of the other kinds of books, it's useful to have a core set that discusses each of the major software-development activities in depth: books on requirements, design, construction, management, testing, and so on. The following sections describe construction resources in depth and then provide an overview of materials available in other software knowledge areas. [Section 35.4](#) wraps these resources into a neat package by defining a software developer's reading program.

35.1. Information About Software Construction

cc2e.com/3588

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 \("Importance of Prerequisites"\)](#) 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.

Cross-Reference

For more in the economics of Extreme Programming and agile programming, see cc2e.com/3545.

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 computer-science knowledge and hands-on lessons. It includes discussions of programming style, design, debugging, and testing. It assumes familiarity with C/C++.

cc2e.com/3549

Although it's out of print and hard to find, Programmers at Work, by Susan Lammers (1986), is worth the search. It contains interviews with the industry's high-profile programmers. The interviews explore their personalities, work habits, and programming philosophies. The luminaries interviewed include Bill Gates (founder of Microsoft), John Warnock (founder of Adobe), Andy Hertzfeld (principal developer of the Macintosh operating system), Butler Lampson (a senior engineer at DEC, now at Microsoft), Wayne Ratliff (inventor of dBase), Dan Bricklin (inventor of VisiCalc), and a dozen others.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

35.2. Topics Beyond Construction

Beyond the core books described in the previous section, here are some books that range further afield from the topic of software construction.

Overview Material

cc2e.com/3595

The following books provide software-development overviews from a variety of vantage points:

Robert L. Glass's *Facts and Fallacies of Software Engineering* (2003) provides a readable introduction to the conventional wisdom of software development dos and don'ts. The book is well researched and provides numerous pointers to additional resources.

My own *Professional Sofware Development* (2004) surveys the field of software development as it is practiced now and as it could be if it were routinely practiced at its best.

The Swebok: Guide to the Software Engineering Body of Knowledge (Abran 2001) provides a detailed decomposition of the software-engineering body of knowledge. This book has dived into detail in the software-construction area. The Guide to the Swebok shows just how much more knowledge exists in the field.

Gerald Weinberg's *The Psychology of Computer Programming* (Weinberg 1998) is packed with fascinating anecdotes about programming. It's far-ranging because it was written at a time when anything related to software was considered to be about programming. The advice in the original review of the book in the ACM Computing Reviews is as good today as it was when the review was written:

Every manager of programmers should have his own copy. He should read it, take it to heart, act on the precepts, and leave the copy on his desk to be stolen by his programmers. He should continue replacing the stolen copies until equilibrium is established (Weiss 1972).

If you can't find *The Psychology of Computer Programming*, look for *The Mythical ManMonth* (Brooks 1995) or *PeopleWare* (DeMarco and Lister 1999). They both drive home the theme that programming is first and foremost something done by people and only secondarily something that happens to involve computers.

A final excellent overview of issues in software development is *Software Creativity* (Glass 1995). This book should have been a breakthrough book on software creativity the way that *Peopleware* was on software teams. Glass discusses creativity versus discipline, theory versus practice, heuristics versus methodology, process versus product, and many of the other dichotomies that define the software field. After years of discussing this book with programmers who work for me, I have concluded that the difficulty with the book is that it is a collection of essays edited by Glass but not entirely written by him. For some readers, this gives the book an unfinished feel. Nonetheless, I still require every developer in my company to read it. The book is out of print and hard to find but worth the effort if you are able to find it.

Software-Engineering Overviews

Every practicing computer programmer or software engineer should have a high-level reference on software engineering. Such books survey the methodological landscape rather than painting specific features in detail. They provide an overview of effective software-engineering practices and capsule descriptions of specific software-engineering techniques. The capsule descriptions aren't detailed enough to train you in the techniques, but a single book would have to be several thousand pages long to do that. They provide enough information so that you can learn how the techniques fit together and can choose techniques for further investigation.

Roger S. Pressman's *Software Engineering: A Practitioner's Approach*, 6th ed. (Pressman 2004), is a balanced treatment of requirements, design, quality validation, and management. Its 900 pages pay little attention to programming practices, but that's a minor limitation, especially if you already have a book on construction such as the

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

35.3. Periodicals

Lowbrow Programmer Magazines

These magazines are often available at local newsstands:

cc2e.com/3516

Software Development. <http://www.sdmagazine.com>. This magazine focuses on programming issues—less on tips for specific environments than on the general issues you face as a professional programmer. The quality of the articles is quite good. It also includes product reviews.

cc2e.com/3523

Dr. Dobb's Journal. <http://www.ddj.com>. This magazine is oriented toward hard-core programmers. Its articles tend to deal with detailed issues and include lots of code.

If you can't find these magazines at your local newsstand, many publishers will send you a complimentary issue, and many articles are available online.

Highbrow Programmer Journals

You don't usually buy these magazines at the newsstand. You usually have to go to a major university library or subscribe to them for yourself or your company:

cc2e.com/3530

IEEE Software. <http://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.

cc2e.com/3537

IEEE Computer. <http://www.computer.org/computer/>. This monthly magazine is the flagship publication of the IEEE (Institute of Electrical and Electronics Engineers) 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.

cc2e.com/3544

Communications of the ACM. <http://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 obfuscating academic style. The good side is that it contains leading-edge information that won't filter down to the lowbrow magazines for years.

Special-Interest Publications

Several publications provide in-depth coverage of specialized topics.

Professional Publications

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

35.4. A Software Developer's Reading Plan

cc2e.com/3507

This section describes the reading program that a software developer needs to work through to achieve full professional standing at my company, Construx Software. The plan described is a generic baseline plan for a software professional who wants to focus on development. Our mentoring program provides for further tailoring of the generic plan to support an individual's interests, and within Construx this reading is also supplemented with training and directed professional experiences.

Introductory Level

To move beyond "introductory" level at Construx, a developer must read the following books:

Adams, James L. *Conceptual Blockbusting: A Guide to Better Ideas*, 4th ed. Cambridge, MA: Perseus Publishing, 2001.

Bentley, Jon. *Programming Pearls*, 2d ed. Reading, MA: Addison-Wesley, 2000.

Glass, Robert L. *Facts and Fallacies of Software Engineering*. Boston, MA: Addison-Wesley, 2003.

McConnell, Steve. *Software Project Survival Guide*. Redmond, WA: Microsoft Press, 1998.

McConnell, Steve. *Code Complete*, 2d ed. Redmond, WA: Microsoft Press, 2004.

Practitioner Level

To achieve "intermediate" status at Construx, a programmer needs to read the following additional materials:

Berczuk, Stephen P. and Brad Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Boston, MA: Addison-Wesley, 2003.

Fowler, Martin. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3d ed. Boston, MA: Addison-Wesley, 2003.

Glass, Robert L. *Software Creativity*. Reading, MA: Addison-Wesley, 1995.

Kaner, Cem, Jack Falk, Hung Q. Nguyen. *Testing Computer Software*, 2d ed. New York, NY: John Wiley & Sons, 1999.

Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2d ed. Englewood Cliffs, NJ: Prentice Hall, 2001.

McConnell, Steve. *Rapid Development*. Redmond, WA: Microsoft Press, 1996.

Wiegers, Karl. *Software Requirements*, 2d ed. Redmond, WA: Microsoft Press, 2003.

cc2e.com/3514

"Manager's Handbook for Software Development," NASA Goddard Space Flight Center. Downloadable from sel.gsfc.nasa.gov/website/documents/online-doc.htm.

Professional Level

A software developer must read the following materials to achieve full professional standing at Construx ("leadership" level). Additional reading may be required for individual development interests, based on the developer's interests.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

35.5. Joining a Professional Organization

cc2e.com/3535

One of the best ways to learn more about programming is to get in touch with other programmers who are as dedicated to the profession as you are. Local user groups for specific hardware and language products are one kind of group. Other kinds are national and international professional organizations. The most practitioner-oriented organization is the IEEE Computer Society, which publishes the IEEE Computer and IEEE Software magazines. For membership information, see <http://www.computer.org>.

cc2e.com/3542

The original professional organization was the ACM, which publishes Communications of the ACM and many special-interest magazines. It tends to be somewhat more academically oriented than the IEEE Computer Society. For membership information, see <http://www.acm.org>.

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

Bibliography

- "A C Coding Standard." 1991. Unix Review 9, no. 9 (September): 42–43.
- Abdel-Hamid, Tarek K. 1989. "The Dynamics of Software Project Staffing: A System Dynamics Based Simulation Approach." IEEE Transactions on Software Engineering SE-15, no. 2 (2): 109–19.
- Abran, Alain, et al. 2001. Swebok: Guide to the Software Engineering Body of Knowledge: Trial Version 1.00-May 2001. Los Alamitos, CA: IEEE Computer Society Press.
- Abrash, Michael. 1992. "Flooring It: The Optimization Challenge." PC Techniques 2, no. 6 (6): 82–88.
- Ackerman, A. Frank , Lynne S. Buchwald , and Frank H. Lewski . 1989. "Software Inspections: An Effective Verification Process." IEEE Software, May/June 1989, 31–36.
- Adams, James L. 2001. Conceptual Block-busting: A Guide to Better Ideas, 4th ed. Cambridge, MA: Perseus Publishing.
- Aho, Alfred V. , Brian W. Kernighan , and Peter J. Weinberg . 1977. The AWK Programming Language. Reading, MA: Addison-Wesley.
- Aho, Alfred V. , John E. Hopcroft , and Jeffrey D. Ullman . 1983. Data Structures and Algorithms. Reading, MA: Addison- Wesley.
- Albrecht, Allan J. 1979. "Measuring Application Development Productivity." Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium, October 1979: 83–92.
- Ambler, Scott. 2003. Agile Database Techniques. New York, NY: John Wiley & Sons.
- Anand, N. 1988. "Clarify Function!" ACM Sigplan Notices 23, no. 6 (6): 69–79.
- Aristotle. The Ethics of Aristotle: The Nicomachean Ethics. Trans. by J.A.K. Thomson. Rev. by Hugh Tredennick. Harmondsworth, Middlesex, England: Penguin, 1976.
- Armenise, Pasquale. 1989. "A Structured Approach to Program Optimization." IEEE Transactions on Software Engineering SE-15, no. 2 (2): 101–8.
- Arnold, Ken , James Gosling, and David Holmes. 2000. The Java Programming Language, 3d ed. Boston, MA: Addison- Wesley.
- Arthur, Lowell J. 1988. Software Evolution: The Software Maintenance Challenge. New York, NY: John Wiley & Sons.
- Augustine, N. R. 1979. "Augustine's Laws and Major System Development Programs." Defense Systems Management Review: 50–76.
- Babich, W. 1986. Software Configuration Management. Reading, MA: Addison-Wesley.
- Bachman, Charles W. 1973. "The Programmer as Navigator." Turing Award Lecture. Communications of the ACM 16, no. 11 (11): 653.
- Baecker, Ronald M. , and Aaron Marcus . 1990. Human Factors and Typography for More Readable Programs. Reading, MA: Addison-Wesley.
- Baird, E. F. 1964. "Research Studies of Programmers and Programming." Unpublished studies reported in Boehm (1971).

[◀ PREVIOUS](#)

[**< Free Open Study >**](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[& \(pointer reference symbol\)](#)

[* \(pointer declaration symbol\) 2nd 3rd](#)

[-> \(pointer symbol\)](#)

[80/20 rule](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[abbreviation of names](#)

[abstract data types](#)

[Abstract Factory pattern](#)

abstraction

[access routines for](#)

[air lock analogy](#)

[checklist](#)

[classes for 2nd](#)

[cohesion with](#)

[complexity, for handling](#)

[consistent level for class interfaces](#)

[defined](#)

[erosion under modification problem](#)

[evaluating](#)

[exactness goal](#)

[forming consistently](#)

[good example for class interfaces](#)

[guidelines for creating class interfaces](#)

[high-level problem domain terms](#)

[implementation structures, low-level](#)

[inconsistent 2nd](#)

[interfaces, goals for](#)

[levels of](#)

[opposites, pairs of](#)

[OS level](#)

[patterns for](#)

[placing items in inheritance trees](#)

[poor example for class interfaces](#)

[problem domain terms, low-level](#)

[programming-language level](#)

[routines for](#)

access routines

[abstraction benefit \[See \[global variables, access routines for\]\(#\); \[routines, access\]\(#\)\]](#)

[abstraction, level of](#)

[advantages of](#)

[barricaded variables benefit](#)

[centralized control from](#)

[creating](#)

[g_prefix guideline](#)

[information hiding benefit](#)

[lack of support for, overcoming](#)

[locking](#)

[parallelism from](#)

[requiring](#)

[accidental problems](#)

[accreting a system metaphor](#)

[accuracy](#)

Ada

[description of](#)

[parameter order](#)

1

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[backup plans](#) 2nd

[bad data, testing for](#)

barricades

[assertions, relation to](#)

[class-level](#)

[input data conversions](#)

[interfaces as boundaries](#)

[operating room analogy](#)

[purpose of](#)

base classes

abstract overridable routines [See [classes, base](#)]

[abstraction aspect of](#)

[coupling, too tight](#)

[Liskov Substitution Principle](#)

[overridable vs. non-overridable routines](#)

[protected data](#)

[routines overridden to do nothing](#)

[single classes from](#)

[Basic](#) [See also [Visual Basic](#)]

[basis testing, structured](#) 2nd

[BCD \(binary coded decimal\) type](#)

[BDUF \(big design up front\)](#)

[beauty](#)

[begin-end pairs](#)

[bibliographies, software](#)

[big design up front \(BDUF\)](#)

[big-bang integration](#)

[binary searches](#)

binding

[compile time](#)

[heuristic design with](#)

[in code](#)

[just in time](#)

[key point](#)

[load time](#)

[run time](#)

[variables, timing of](#)

[black-box testing](#)

[blank lines for formatting](#) 2nd

blocks

[braces writing rule](#)

[comments on](#)

[conditionals, clarifying](#)

[defined](#)

[emulated pure layout style](#)

[pure, layout style](#)

[single statements](#)

[Book Paradigm](#)

boolean expressions

0, comparisons to [See [boolean variables, expressions with](#); [control structures, boolean expressions in](#);

[else if](#) 1, 2, 3]

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

C language

[ADTs with](#)
 [boolean expression syntax](#)
 [description of](#)
 [naming conventions for 2nd](#)
 [pointers](#)
 [string data types 2nd](#)
 [string index errors](#)

C#

C++

[assertion example](#)
 [boolean expression syntax](#)
 [debugging stubs with](#)
 [description of](#)
 [DoNothing\(\) macros](#)
 [exceptions in](#)
 [inline routines](#)
 [interface considerations](#)
 [layout recommended](#)
 [macro routines](#)
 [naming conventions for](#)
 [null statements](#)
 [parameters, by reference vs. by value](#)
 [pointers 2nd 3rd](#)
 [preprocessors, excluding debug code](#)
 [resources for](#)
 [side effects](#)
 [source files, layout in](#)
 [caching, code tuning with](#)

Capability Maturity Model (CMM)

capturing design work

Cardinal Rule of Software Evolution

CASE (computer-aided software engineering) tools

case statements

 alpha ordering [See [conditional statements, case statements](#); [conditional statements, switch statements](#); [control structures, case](#)]
 [checklist](#)
 [debugging](#)
 [default clauses](#)
 [drop-throughs](#)
 [end of case statements](#)
 [endline layout](#)
 [error detection in](#)
 [frequency of execution ordering 2nd](#)
 [if statements, comparing performance with](#)
 [key points](#)
 [language support for](#)
 [nested ifs, converting from 2nd](#)
 [normal case first rule](#)
 [numeric ordering](#)
 [switch statements](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

daily build and smoke tests

[automation of](#)
 [benefits of](#)
 [broken builds 2nd](#)
 [build groups](#)
 [checklist](#)
 [defined](#)
 [diagnosis benefit](#)
 [holding area for additions](#)
 [importance of](#)
 [morning releases](#)
 [pressure](#)
 [pretest requirement](#)
 [revisions](#)
 [smoke tests](#)
 [unsurfaced work](#)

data

[architecture prerequisites](#)
 [bad classes, testing for](#)
 [change, identifying areas of](#)
 [combined states](#)
 [defined state](#)
 [defined-used paths, testing](#)
 [design](#)
 [entered state](#)
 [exited state](#)
 [good classes, testing](#)
 [killed state](#)
 [legacy, compatibility with](#)
 [nominal case errors](#)
 [test, generators for](#)
 [used state](#)
[data dictionaries](#)
[data flow testing](#)
[data literacy test](#)
[data recorder tools](#)
[data structures](#)

data transformations for code tuning

 array dimension minimization [See [data, code tuning](#)]
 [array reference minimization](#)
 [caching data](#)
 [floating point to integers](#)
 [indexing data](#)
 [purpose of](#)

data types

 'a' prefix convention [See [data, types; variables, types of](#)]
 [BCD](#)
 [change, identifying areas of](#)
 [checklist](#)
 [control structures, relationship to](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[early-wave environments](#)

[ease of maintenance design goal](#)

[eclecticism](#)

[editing tools](#)

[beautifiers](#) [See [tools, editing](#)]

[class-hierarchy generators](#)

[cross-reference tools](#)

[Diff tools](#)

[grep](#)

[IDEs](#)

[interface documentation](#)

[merge tools](#)

[multiple-file string searches](#)

[templates](#)

[efficiency](#)

[eighty/twenty \(80/20\) rule](#)

[else clauses](#)

[boolean function calls with](#)

[case statements instead of](#)

[chains, in](#)

[common cases first guideline](#)

[correctness testing](#)

[default for covering all cases](#)

[gotos with](#)

[null](#)

[embedded life-critical systems](#)

[emergent nature of design process](#)

[emulated pure blocks layout style](#)

[encapsulation](#)

[assumptions about users](#)

[checklist](#)

[classes, role for](#)

[coupling classes too tightly](#)

[downcast objects](#)

[friend class concern](#)

[heuristic design with](#)

[minimizing accessibility](#)

[private details in class interfaces](#)

[public data members](#)

[public members of classes](#)

[public routines in interfaces concern](#)

[semantic violations of](#)

[weak](#)

[endless loops 2nd](#)

[endline comments](#)

[endline layout 2nd 3rd](#)

[enumerated types](#)

 benefits of [See [data types, enumerated types](#)]

[booleans, alternative to](#)

[C++ 2nd](#)

[1](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[Facade pattern](#)

[factorials](#)

[factoring](#) [See also [refactoring](#)]

factory methods

[Factory Method pattern](#)

[nested ifs refactoring example](#)

[refactoring to](#)

[fan-in](#)

[fan-out](#)

[farming metaphor](#)

[fault tolerance](#)

[feature-oriented integration](#)

[Fibonacci numbers](#)

files

[ADTs, treating as](#)

[authorship records for](#)

[C++ source file order](#)

[deleting multiple example](#)

[documenting](#)

[layout within](#)

[naming 2nd](#)

[routines in](#)

[final keyword, Java](#)

[finally statements](#)

fixing defects

[checking fixes](#) [See [defects in code, fixing](#)]

[checklist](#)

[diagnosis confirmation](#)

[hurrying, impact of](#)

[initialization defects](#)

[maintenance issues](#)

[one change at a time rule](#)

[reasoning for changes](#)

[saving unfixed code](#)

[similar defects, looking for](#)

[special cases](#)

[symptoms, fixing instead of problems](#)

[understand first guideline](#)

[unit tests for](#)

flags

[change, identifying areas of](#)

[comments for bit-level meanings](#)

[enumerated types for](#)

[gotos, rewriting with](#)

[names for](#)

[semantic coupling with](#)

flexibility

[coupling criteria for](#)

[defined](#)

floating-point data types

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

General Principle of Software Quality

[collaboration effects](#)

[costs](#)

[debugging](#)

[defined](#)

global variables

aliasing problems with [See [variables, global](#)]

[alternatives to](#)

[annotating](#)

[changes to, inadvertent](#)

[checklist for](#)

[class variable alternatives](#)

[code reuse problems](#)

[commenting 2nd](#)

[enumerated types emulation by](#)

[g_ prefix guideline](#)

[hiding implementation in classes](#)

[information hiding problems with](#)

[initialization problems](#)

[intermediate results, avoiding](#)

[key points](#)

[local first guideline](#)

[locking](#)

[modularity damaged by](#)

[named constants emulation by](#)

[naming 2nd 3rd 4th 5th 6th](#)

[objects for, monster](#)

[overview of](#)

[persistence of](#)

[preservation of values with](#)

[re-entrant code problems](#)

[refactoring](#)

[risk reduction strategies](#)

[routines using as parameters](#)

[semantic coupling with](#)

[streamlining data use with](#)

[tramp data, eliminating with](#)

god classes

[gonzo programming](#)

[good data, testing](#)

goto statements

Ada, inclusion in [See [control structures, gotos](#)]

[advantages of](#)

[alternatives compared with](#)

[checklist](#)

[deallocation with](#)

[disadvantages of](#)

[duplicate code, eliminating with](#)

[else clauses with](#)

[error processing with](#)

[fall-through with](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[habits of programmers](#)

[hacking approach to design](#)

hardware

[dependencies, changing](#)

[performance enhancement with](#)

[has a relationships](#)

heuristic design

abstractions, forming consistent [See [design, heuristic](#); [design, practice heuristics](#); [heuristics, design with](#)]

[alternatives from patterns](#)

[avoiding failure](#)

[binding time considerations](#)

[bottom-up approach to design](#)

[brute force](#)

[capturing work](#)

[central points of control](#)

[change, identifying areas of](#)

[checklist for](#)

[collaboration](#)

[communications benefit from patterns](#)

[completion of determining](#)

[coupling considerations](#)

[diagrams, drawing](#)

[divide and conquer technique](#)

[encapsulation](#)

[error reduction with patterns](#)

[formality of determining](#)

[formalizing class contracts](#)

[goals checklist](#)

[guidelines for using](#)

[hierarchies for](#)

[information hiding 2nd](#)

[inheritance](#)

[interfaces, formalizing as contracts](#)

[iteration practice](#)

[key points](#)

[level of detail needed](#)

[modularity](#)

[multiple approach suggestion](#)

[nature of design process](#)

[nondeterministic basis for](#)

[object-oriented, resource for](#)

[objects, real world, finding](#)

[patterns 2nd](#)

[practices 2nd](#)

[prototyping](#)

[resources for](#)

[responsibilities, assigning to objects](#)

[strong cohesion](#)

[summary list of rules](#)

[testing, anticipating](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

I/O (input/output)

[architecture prerequisites](#)

[change, identifying areas of performance considerations](#)

[IDEs \(Integrated Development Environments\)](#)

[IEEE \(Institute for Electric and Electrical Engineers\)](#)

if statements

boolean function calls with [See [conditional statements, if statements](#); [control structures, if statements](#)]

[break blocks, simplification with](#)

[case statements, compared to 2nd](#)

[case statements, converting to 2nd](#)

[chains of](#)

[checklist](#)

[common cases first guideline](#)

[continuation lines in](#)

[covering all cases](#)

[else clauses 2nd](#)

[equality, branching on](#)

[error processing examples](#)

[factoring to routines](#)

[flipped](#)

[frequency, testing in order of](#)

[gotos rewritten with 2nd](#)

[if-then-else statements, converting to](#)

[key points](#)

[lookup tables, substituting](#)

[multiple returns nested in](#)

[negatives in, making positive](#)

[normal case first guideline](#)

[normal path first guideline](#)

[null if clauses](#)

[plain if-then statements](#)

[refactoring](#)

[simplification](#)

[single-statement layout](#)

[tables, replacing with](#)

[types of](#)

[implicit declarations](#)

[implicit instancing](#)

[in keyword, creating](#)

[incomplete preparation, causes of](#)

[incremental development metaphor](#)

incremental integration

benefits of [See [integration, incremental](#)]

[bottom-up strategy](#)

[classes 2nd](#)

[customer relations benefit](#)

[defined](#)

[disadvantages of top-down strategy](#)

[errors, locating](#)

[functionalities 2nd](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W]

[jamming loops](#)

[Java](#)

[assertion example in](#)

[boolean expression syntax](#)

[description of](#)

[exceptions](#)

[layout recommended](#)

[live time examples](#)

[naming conventions for 2nd](#)

[parameters example](#)

[persistence of variables](#)

[resources for](#)

[Javadoc 2nd](#)

[JavaScript](#)

[JUnit](#)

[just in time binding](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[key construction decisions](#)

[killed data state](#)

[kinds of software projects](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[languages, programming](#)

[Law of Demeter](#)

[layout](#)

array references [See [coding, style](#); [conventions, coding, formatting](#); [programming conventions, formatting rules](#); [readability, formatting for](#); [style issues, formatting](#)]

[assignment statement continuations](#)

[begin-end pairs](#)

[blank lines 2nd](#)

[block style](#)

[brace styles 2nd](#)

[C++ side effects](#)

[checklist](#)

[classes](#)

[closely related statement elements](#)

[comments](#)

[complicated expressions](#)

[consistency requirement](#)

[continuing statements](#)

[control statement continuations](#)

[control structure styles](#)

[declarations](#)

[discourse rules](#)

[documentation in code](#)

[double indented begin-end pairs](#)

[emulating pure blocks](#)

[endline layout 2nd](#)

[ends of continuations](#)

[files, within](#)

[Fundamental Theorem of Formatting](#)

[gotos](#)

[incomplete statements](#)

[indentation](#)

[interfaces](#)

[key points](#)

[language-specific guidelines](#)

[logical expressions](#)

[logical structure, reflecting 2nd](#)

[mediocre example](#)

[misleading indentation example](#)

[misleading precedence](#)

[modifications guideline](#)

[multiple statements per line](#)

[negative examples](#)

[objectives of](#)

[parentheses for](#)

[pointers, C++](#)

[pure blocks style](#)

[readability goal](#)

[religious aspects of](#)

[resources on](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[Macintosh naming conventions](#)

[macro routines](#)

alternatives for [See [routines, macro](#)]

[limitations on](#)

[multiple statements in](#)

[naming 2nd](#)

[parentheses with](#)

[magazines on programming](#)

[magic variables, avoiding 2nd 3rd](#)

[maintenance](#)

[comments requiring](#)

[design goal for](#)

[error-prone routines, prioritizing for](#)

[fixing defects, problems from](#)

[maintainability defined](#)

[readability benefit for](#)

[structures for reducing](#)

[major construction practices checklist](#)

[managing construction](#)

code ownership attitudes [See [construction, managing](#)]

[complexity](#)

[good coding, encouraging](#)

[inspections, management role in](#)

[key points](#)

[managers](#)

[measurements](#)

[programmers, treatment of](#)

[readability standard](#)

[resources on](#)

[reviewing all code](#)

[rewarding good practices](#)

[schedules, estimating](#)

[signing off on code](#)

[standards, authority to set](#)

[standards, IEEE 2nd](#)

[two-person teams](#)

[markers, defects from](#)

[matrices](#)

[mature technology environments](#)

[maximum normal configurations](#)

[maze recursion example](#)

[McCabe's complexity metric 2nd](#)

[measure twice, cut once](#)

[measurement](#)

[advantages of](#)

[arguing against](#)

[goals for](#)

[outlier identification](#)

[resources for](#)

[side effects of](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[named constants](#)

naming conventions

"a" prefix convention [See [declarations, naming; readability, naming variables for](#)]

[abbreviating names](#)

[abbreviation guidelines](#)

[arrays](#)

[benefits of](#)

[C language 2nd](#)

[C++](#)

[capitalization 2nd](#)

[case-insensitive languages](#)

[characters, hard to read](#)

[checklist 2nd](#)

[class member variables](#)

[class vs. object names](#)

[common operations, for](#)

[constants](#)

[cross-project benefits](#)

[descriptiveness guideline](#)

[documentation 2nd](#)

[enumerated types 2nd 3rd](#)

[files](#)

[formality, degrees of](#)

[function return values](#)

[global variables 2nd](#)

[homonyms](#)

[Hungarian](#)

[informal](#)

[input parameters](#)

[Java 2nd](#)

[key points](#)

[kinds of information in names](#)

[language-independence guidelines](#)

[length, not limiting](#)

[Macintosh](#)

[meanings in names, too similar](#)

[misleading names](#)

[misspelled words](#)

[mixed-language considerations](#)

[multiple natural languages](#)

[numbers, differentiating solely by](#)

[numerals](#)

[opposites, use of](#)

[parameters](#)

[phonic abbreviations](#)

[prefix standardization](#)

[procedure descriptions](#)

[proliferation reduction benefit](#)

[pronunciation guideline](#)

[purpose of](#)

[readability](#)

[variables](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

object-oriented programming

[resources for 2nd](#)

[object-parameter coupling](#)

[objectives, software quality 2nd](#)

objects

ADTs as [See [object-oriented programming, objects](#)]

[attribute identification](#)

[class names, differentiating from](#)

[classes, contrasted to](#)

[containment, identifying](#)

[deleting objects](#)

[factory methods 2nd 3rd](#)

[identifying](#)

[inheritance, identifying](#) [See also [inheritance](#)]

[interfaces, designing](#) [See also [interfaces, class](#)]

[operations, identifying](#)

[parameters, using as 2nd](#)

[protected interfaces, designing](#)

[public vs. private members, designing](#)

[real world, finding](#)

[refactoring](#)

[reference objects](#)

[responsibilities, assigning to](#)

[singleton property, enforcing](#)

[steps in designing](#)

[Observer pattern](#)

off-by-one errors

[boundary analysis](#)

[fixing, approaches to](#)

[offensive programming](#)

[one-in, one-out control constructs](#)

[operating systems](#)

[operations, costs of common](#)

[opposites for variable names](#)

[optimization, premature](#) [See also [performance tuning](#)]

[oracles, software](#)

[out keyword creation](#)

[overengineering](#)

[overflows, integer](#)

[overlay linkers](#)

[overridable routines 2nd](#)

[oyster farming metaphor](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[packages](#)

[paging operations](#)

pair programming

 benefits of [See [collaboration, pair programming](#)]

[checklist](#)

[coding standards support for](#)

[compared to other collaboration](#)

[defined](#)

[inexperienced pairs](#)

[key points](#)

[pace, matching](#)

[personality conflicts](#)

[resources](#)

[rotating pairs](#)

[team leaders](#)

[visibility of monitor](#)

[watching](#)

[when not to use](#)

parameters of routines

 abstraction and object parameters [See [routines, parameters](#)]

[actual, matching to formal](#)

[asterisk \(*\) rule for pointers](#)

[behavior dependence on](#)

[by reference vs. by value](#)

[checklist for, 185 C-library order](#)

[commenting](#)

[const prefix 2nd 3rd](#)

[dependencies, clarifying](#)

[documentation](#)

[enumerated types for](#)

[error variables](#)

[formal, matching to actual](#)

[global variables for](#)

[guidelines for use in routines](#)

[in keyword creation](#)

[input-modify-output order](#)

[Java](#)

[list size as refactoring indicator](#)

[matching actual to formal](#)

[naming 2nd 3rd 4th 5th 6th](#)

[number of, limiting](#)

[objects, passing](#)

[order for](#)

[out keyword creation](#)

[passing, types of](#)

[refactoring 2nd](#)

[status](#)

[structures as](#)

[using all of rule](#)

[variables, using as](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[quality assurance](#)

[checklist](#)

[good practices table for](#)

[prerequisites role in](#)

[requirements checklist](#)

[quality gates](#)

[quality of software](#)

[accuracy \[See \[construction, quality of\]\(#\)\]](#)

[adaptability](#)

[change-control procedures](#)

[checklist for](#)

[correctness](#)

[costs of finding defects](#)

[costs of fixing defects](#)

[debugging, role of 2nd](#)

[detection of defects by various techniques, table of](#)

[development process assurance activities](#)

[efficiency](#)

[engineering guidelines](#)

[explicit activity for](#)

[external audits](#)

[external characteristics of](#)

[Extreme Programming](#)

[flexibility](#)

[gates](#)

[General Principle of Software Quality](#)

[integrity](#)

[internal characteristics](#)

[key points](#)

[maintainability](#)

[measurement of results](#)

[multiple defect detection techniques recommended](#)

[objectives, setting 2nd](#)

[optimization conflicts](#)

[percentage of defects measurement](#)

[portability](#)

[programmer performance, objectives based](#)

[prototyping](#)

[readability](#)

[recommended combination for](#)

[relationships of characteristics](#)

[reliability](#)

[resources for](#)

[reusability](#)

[reviews](#)

[robustness](#)

[standards, IEEE 2nd](#)

[testing 2nd 3rd](#)

[understandability](#)

[usability](#)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 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 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[random-data generators](#)

readability

[as management standard](#)

[defects exposing lack of defined](#)

[importance of 2nd](#)

[maintenance benefit from](#)

[positive effects from](#)

[private vs. public programs](#)

[professional development, importance to](#)

[structures, importance of](#)

[warning sign, as a](#)

[reading as a skill](#)

[reading plan for software developers](#)

[records, refactoring](#)

recursion

alternatives to [See [control structures, recursive](#)]

[checklist](#)

[defined](#)

[factorials using](#)

[Fibonacci numbers using](#)

[guidelines for](#)

[key points](#)

[maze example](#)

[safety counters for](#)

[single routine guideline](#)

[sorting example](#)

[stack space concerns](#)

[terminating](#)

refactoring

[80/20 rule](#)

[adding routines](#)

[algorithms](#)

[arrays](#)

[backing up old code](#)

[bidirectional class associations](#)

[boolean expressions](#)

[case statements](#)

[checklists for 2nd](#)

[checkpoints for](#)

[class cohesion indicator](#)

[class interfaces](#)

[classes 2nd 3rd 4th](#)

[code tuning, compared to](#)

[collections](#)

[comments on bad code](#)

[complex modules](#)

[conditional expressions](#)

[constant values varying among subclass](#)

[constructors to factory methods](#)

[coupling, controlling](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[safety counters in loops](#)

[sandwich integration](#)

scaffolding

[debugging with](#)

[testing 2nd](#)

[scalability](#) [See also [size of projects](#)]

[schedules, estimating](#)

[scientific method, classic steps in](#)

[SCM \(software configuration management\)](#) [See also [configuration management](#)]

scope of variables

[convenience argument](#) [See [variables, scope of](#)]

[defined](#)

[global scope, problems with](#)

[grouping related statements](#)

[key point](#)

[language differences](#)

[live time, minimizing](#)

[localizing references to variables](#)

[loop initializations](#)

[manageability argument](#)

[minimizing, guidelines for](#)

[restrict and expand tactic](#)

[span of variables](#)

[value assignments](#)

[variable names, effects on](#)

[scribe role in inspections](#)

scripts

[programming tools, as](#)

[slowness of](#)

[SDFs \(software development folders\)](#)

[security](#)

[selections, code](#)

[selective data](#)

[self-documenting code 2nd](#)

[semantic coupling](#)

[semantic prefixes](#)

[semantics checkers](#)

[sentinel tests for loops](#)

[sequences, code](#)

[hiding with routines](#)

[structured programming concept of](#)

[sequential approach](#)

[sequential cohesion](#)

[Set\(\) routines](#)

[setup code, refactoring](#)

[setup tools](#)

[short-circuit evaluation 2nd](#)

[side effects, C++](#)

[signing off on code](#)

[simple-data-parameter coupling](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[T-shaped integration](#)

table-driven methods

[advantages of](#)

[binary searches with](#)

[case statement approach](#)

[checklist](#)

[code-tuning with](#)

[creating from expressions](#)

[days-in-month example](#)

[defined](#)

[design method](#)

[endpoints of ranges](#)

[flexible-message-format example](#)

[fudging keys for](#)

[indexed access tables 2nd](#)

[insurance rates example](#)

[issues in](#)

[key points](#)

[keys for](#)

[lookup issue](#)

[miscellaneous examples](#)

[object approach](#)

[precomputing calculations](#)

[purpose of](#)

[stair-step access tables](#)

[storage issue](#)

[transforming keys](#)

[Tacoma Narrows bridge](#)

[takedown code, refactoring](#)

[Team Software Process \(TSP\)](#)

[teams](#)

[build groups](#)

[checklist](#)

[development processes used by](#)

[expanding to meet schedules](#)

[managers](#)

[physical environment](#)

[privacy of offices](#)

[process, importance to](#)

[religious issues](#)

[resources on](#)

[size of projects, effects of](#)

[style issues](#)

[time allocations](#)

[variations in performance](#)

[technology waves, determining your location in](#)

[Template Method pattern](#)

[template tools](#)

[temporal cohesion](#)

[temporary variables](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[UDFs \(unit development folders\)](#)

[UDT \(user-defined type\) abbreviations](#)

[UML diagrams 2nd](#)

[understandability](#) [See also [readability](#)]

[Unicode](#)

[unit development folders \(UDFs\)](#)

[unit testing](#)

[UNIX programming environment](#)

[unrolling loops](#)

[unswitching loops](#)

[upstream prerequisites](#)

[usability](#)

[used data state](#)

user interfaces

[architecture prerequisites](#)

[refactoring data from](#)

[subsystem design](#)

[user-defined type \(UDT\) abbreviations](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

validation

[assumptions to check, list of](#)
[data types, suspicious](#)
[enumerated types for](#)
[external data sources rule](#)
[input parameters rule](#)

variable names

abbreviation guidelines [See [naming conventions, variables, for; readability, naming variables for; variables, naming](#)]
[accurate description rule](#)
[bad names, examples of 2nd](#)
[boolean variables](#)
[C language 2nd](#)
[C++ 2nd](#)
[capitalization](#)
[characters, hard to read](#)
[checklist](#)
[class member variables](#)
[computed-value qualifiers](#)
[constants](#)
[enumerated types](#)
[full description rule](#)
[global qualifiers for](#)
[good names, examples of 2nd](#)
[homonyms](#)
[Java conventions](#)
[key points](#)
[kinds of information in](#)
[length, optimum](#)
[loop indexes](#)
[misspelled words](#)
[multiple natural languages](#)
[namespaces](#)
[numerals in](#)
[opposite pairs for](#)
[phonic abbreviations](#)
[problem orientation rule](#)
[psychological distance](#)
[purpose of](#)
[reserved names](#)
[routine names, differentiating from](#)
[scope, effects of](#)
[similarity of names, too much](#)
[specificity rule](#)
[status variables](#)
[temporary variables](#)
[type names, differentiating from](#)
[Visual Basic](#)

variables

[binding time for](#)
[change, identifying areas of](#)
[declaration](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W]

walk-throughs 2nd

warning signs

while loops

advantages of

break statements

do-while loops

exits in

infinite loops

misconception of evaluation

null statements with

purpose of

tests, position of

white space

blank lines 2nd

defined

grouping with

importance of

indentation

individual statements with

white-box testing 2nd

wicked problems

Wikis

WIMP syndrome

WISCA syndrome

workarounds, documenting

writing metaphor for coding