The Pragmatic Programmer

from journeyman to master

Andrew Hunt
David Thomas

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

What others in the trenches say about *The Pragmatic Programmer*. . .

"The cool thing about this book is that it's great for keeping the programming process fresh. *[The book]* helps you to continue to grow and clearly comes from people who have been there."

► **Kent Beck,** author of *Extreme Programming Explained: Embrace Change*

"I found this book to be a great mix of solid advice and wonderful analogies!"

► **Martin Fowler,** author of *Refactoring* and *UML Distilled*

"I would buy a copy, read it twice, then tell all my colleagues to run out and grab a copy. This is a book I would never loan because I would worry about it being lost."

► **Kevin Ruland,** Management Science, MSG-Logistics

"The wisdom and practical experience of the authors is obvious. The topics presented are relevant and useful. . . . By far its greatest strength for me has been the outstanding analogies—tracer bullets, broken windows, and the fabulous helicopter-based explanation of the need for orthogonality, especially in a crisis situation. I have little doubt that this book will eventually become an excellent source of useful information for journeymen programmers and expert mentors alike."

► **John Lakos,** author of *Large-Scale C++ Software Design*

“This is the sort of book I will buy a dozen copies of when it comes out so I can give it to my clients.”

► **Eric Vought,** Software Engineer

“Most modern books on software development fail to cover the basics of what makes a great software developer, instead spending their time on syntax or technology where in reality the greatest leverage possible for any software team is in having talented developers who really know their craft well. An excellent book.”

► **Pete McBreen,** Independent Consultant

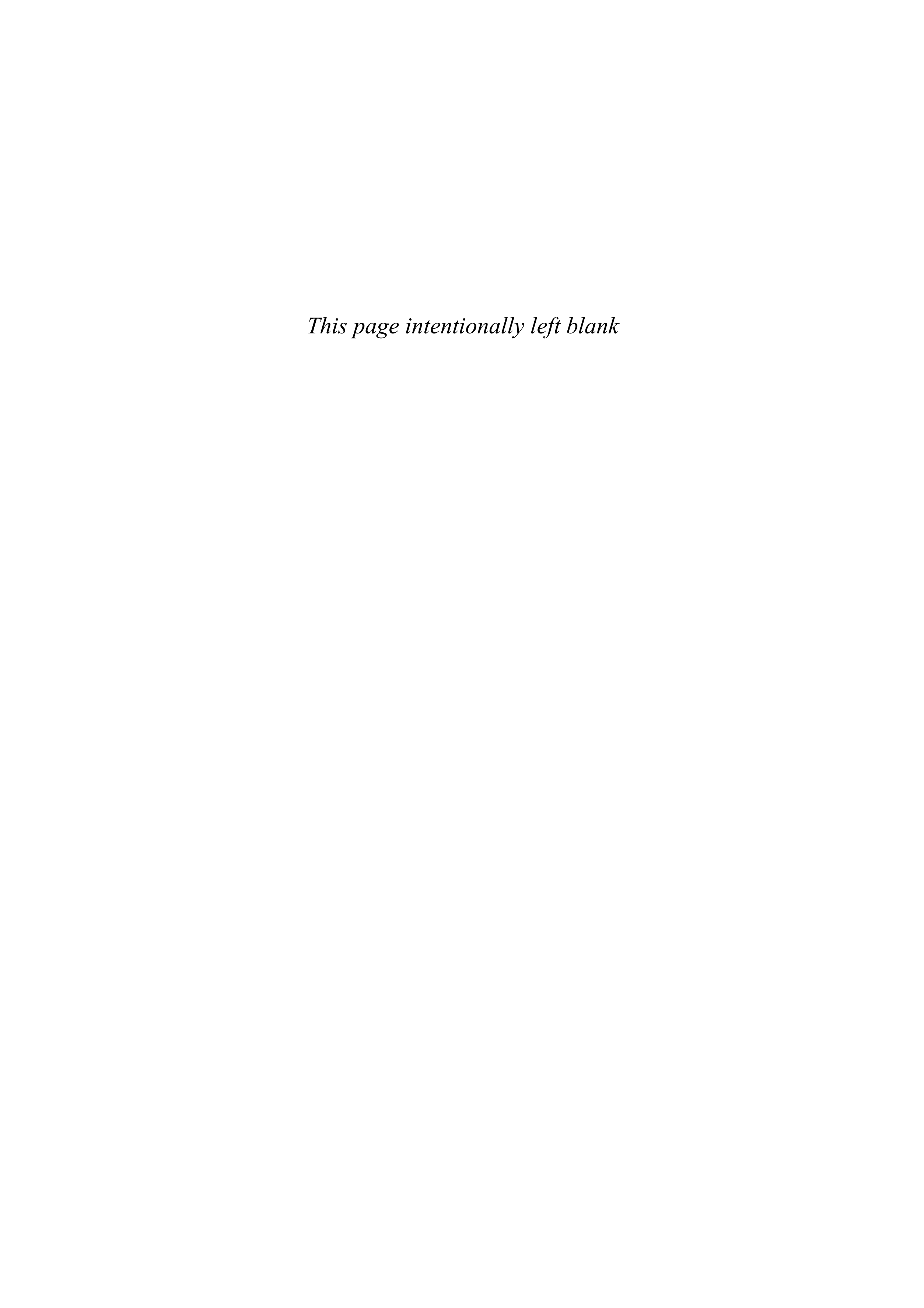
“Since reading this book, I have implemented many of the practical suggestions and tips it contains. Across the board, they have saved my company time and money while helping me get my job done quicker! This should be a desktop reference for everyone who works with code for a living.”

► **Jared Richardson,** Senior Software Developer,
iRenaissance, Inc.

“I would like to see this issued to every new employee at my company. . . .”

► **Chris Cleeland,** Senior Software Engineer,
Object Computing, Inc.

The Pragmatic Programmer



This page intentionally left blank

The Pragmatic Programmer

From Journeyman to Master

Andrew Hunt
David Thomas

ADDISON–WESLEY
An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts   Harlow, England   Menlo Park, California
Berkeley, California   Don Mills, Ontario   Sydney
Bonn   Amsterdam   Tokyo   Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison–Wesley was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals.

Lyrics from the song "The Boxer" on page 157 are Copyright ©1968 Paul Simon. Used by permission of the Publisher: Paul Simon Music. Lyrics from the song "Alice's Restaurant" on page 220 are by Arlo Guthrie, ©1966, 1967 (renewed) by APPLESEED MUSIC INC. All Rights Reserved. Used by Permission.

The authors and publisher have taken care in the preparation of this book, but make no express or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact:

AWL Direct Sales
Addison Wesley Longman, Inc.
One Jacob Way
Reading, Massachusetts 01867
(781) 944-3700

Visit AWL on the Web: www.awl.com/cseng

Library of Congress Cataloging-in-Publication Data

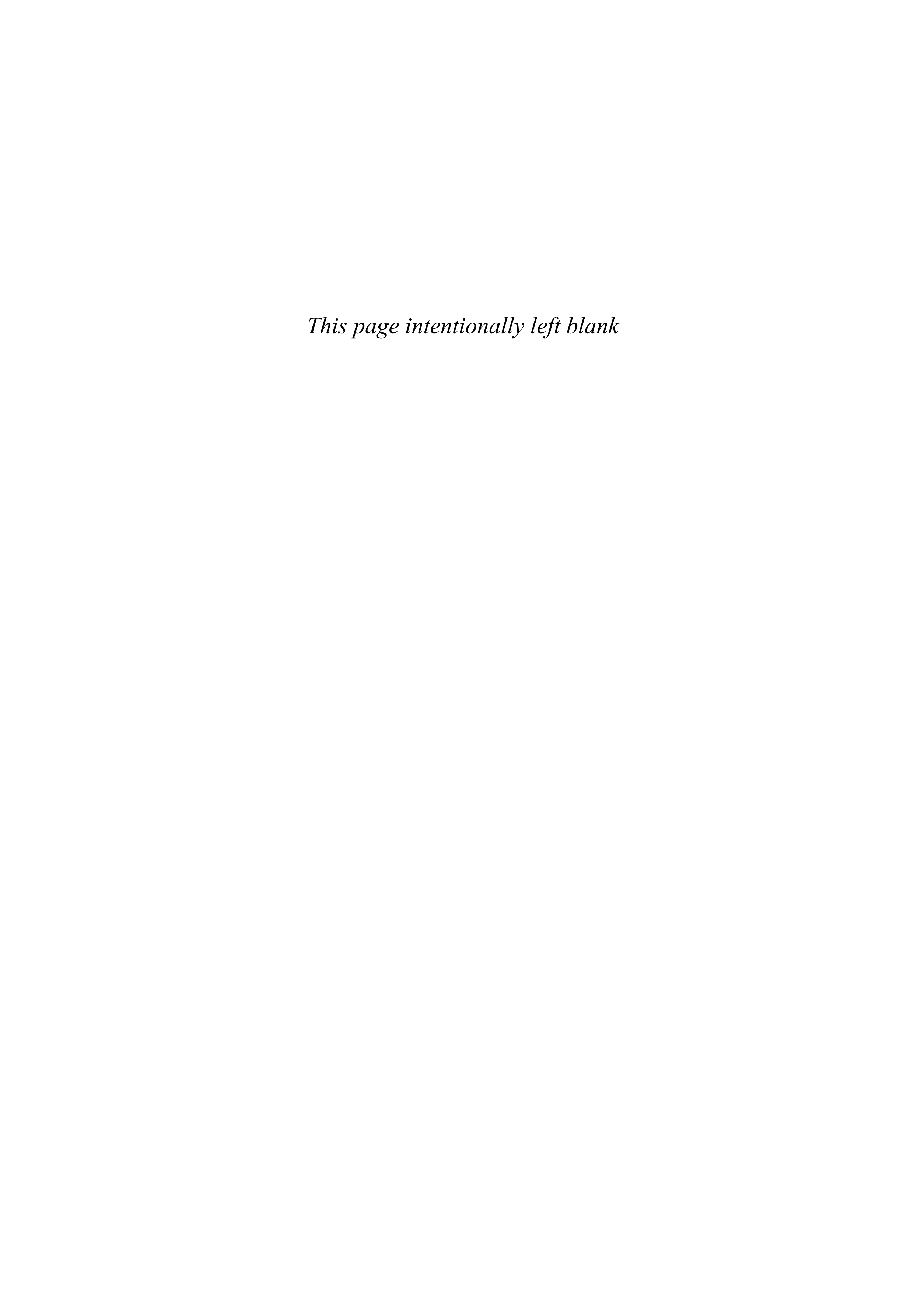
Hunt, Andrew, 1964–
The Pragmatic Programmer / Andrew Hunt, David Thomas.
p. cm.
Includes bibliographical references.
ISBN 0-201-61622-X
1. Computer programming. I. Thomas, David, 1956– .
II. Title.
QA76.6.H857 1999
005.1--dc21 99–43581
CIP

Copyright © 2000 by Addison Wesley Longman, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

ISBN 0-201-61622-X
Text printed in the United States on recycled paper at Courier Stoughton in Stoughton, Massachusetts.
25th Printing February 2010

For Ellie and Juliet,
Elizabeth and Zachary,
Stuart and Henry


This page intentionally left blank

Contents

Foreword xiii

Preface xvii

1 A Pragmatic Philosophy 1

1. The Cat Ate My Source Code 2
2. Software Entropy 4
3. Stone Soup and Boiled Frogs 7
4. Good-Enough Software 9
5. Your Knowledge Portfolio 12
6. Communicate! 18

2 A Pragmatic Approach 25

7. The Evils of Duplication 26
8. Orthogonality 34
9. Reversibility 44
10. Tracer Bullets 48
11. Prototypes and Post-it Notes 53
12. Domain Languages 57
13. Estimating 64

3 The Basic Tools 71

14. The Power of Plain Text 73
15. Shell Games 77
16. Power Editing 82
17. Source Code Control 86
18. Debugging 90
19. Text Manipulation 99
20. Code Generators 102

4 PRAGMATIC PARANOIA 107
21. Design by Contract 109
22. Dead Programs Tell No Lies 120
23. Assertive Programming 122
24. When to Use Exceptions 125
25. How to Balance Resources 129

5 BEND, OR BREAK 137
26. Decoupling and the Law of Demeter 138
27. Metaprogramming 144
28. Temporal Coupling 150
29. It's Just a View 157
30. Blackboards 165

6 WHILE YOU ARE CODING 171
31. Programming by Coincidence 172
32. Algorithm Speed 177
33. Refactoring 184
34. Code That's Easy to Test 189
35. Evil Wizards 198

7 BEFORE THE PROJECT 201
36. The Requirements Pit 202
37. Solving Impossible Puzzles 212
38. Not Until You're Ready 215
39. The Specification Trap 217
40. Circles and Arrows 220

8 PRAGMATIC PROJECTS 223
41. Pragmatic Teams 224
42. Ubiquitous Automation 230
43. Ruthless Testing 237
44. It's All Writing 248
45. Great Expectations 255
46. Pride and Prejudice 258

Appendices

A RESOURCES **261**

Professional Societies 262

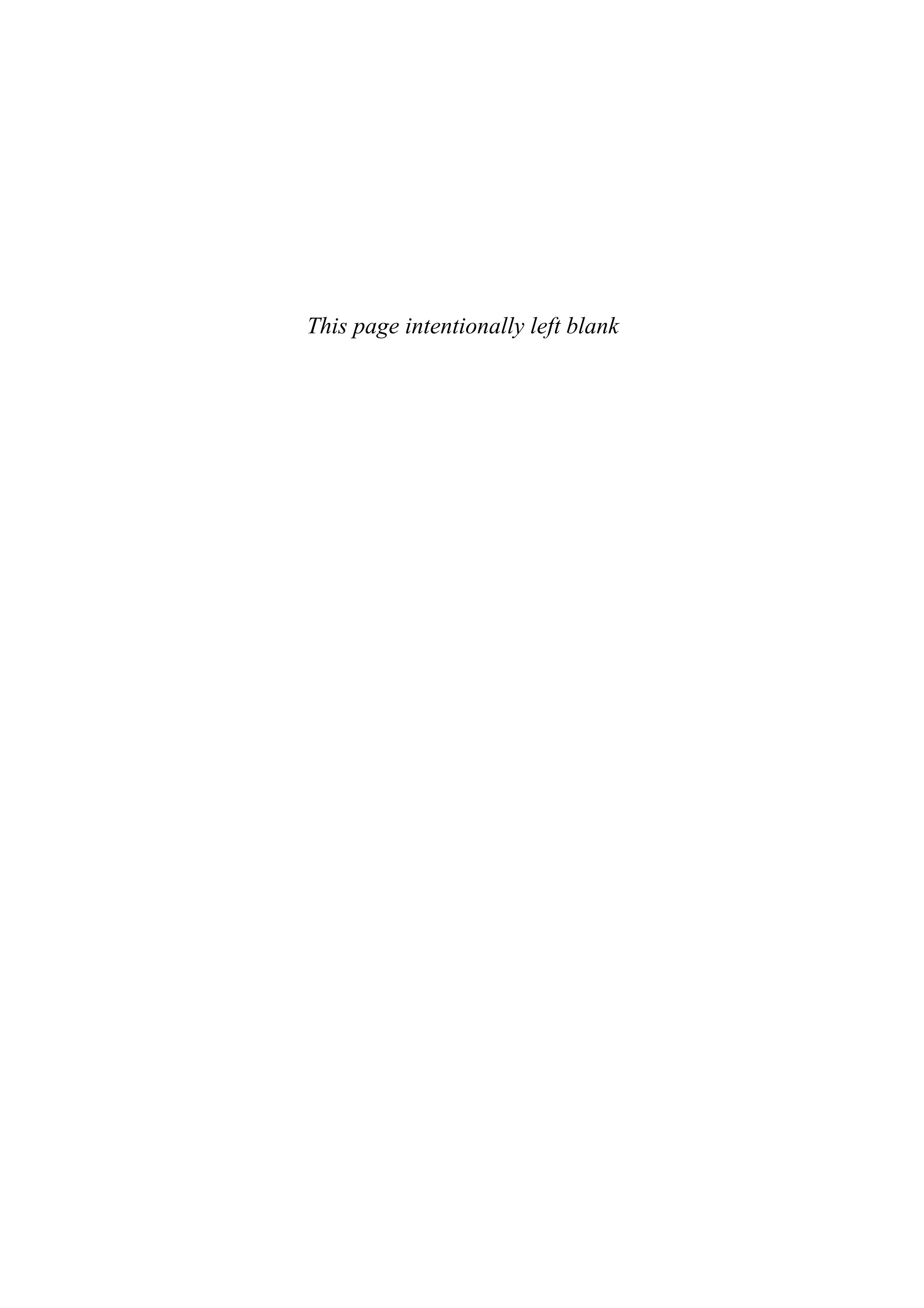
Building a Library 262

Internet Resources 266

Bibliography 275

B ANSWERS TO EXERCISES **279**

INDEX **309**


This page intentionally left blank

Foreword

As a reviewer I got an early opportunity to read the book you are holding. It was great, even in draft form. Dave Thomas and Andy Hunt have something to say, and they know how to say it. I saw what they were doing and I knew it would work. I asked to write this foreword so that I could explain why.

Simply put, this book tells you how to program in a way that you can follow. You wouldn't think that that would be a hard thing to do, but it is. Why? For one thing, not all programming books are written by programmers. Many are compiled by language designers, or the journalists who work with them to promote their creations. Those books tell you how to *talk* in a programming language—which is certainly important, but that is only a small part of what a programmer does.

What does a programmer do besides talk in programming language? Well, that is a deeper issue. Most programmers would have trouble explaining what they do. Programming is a job filled with details, and keeping track of those details requires focus. Hours drift by and the code appears. You look up and there are all of those statements. If you don't think carefully, you might think that programming is just typing statements in a programming language. You would be wrong, of course, but you wouldn't be able to tell by looking around the programming section of the bookstore.

In *The Pragmatic Programmer* Dave and Andy tell us how to program in a way that we can follow. How did they get so smart? Aren't they just as focused on details as other programmers? The answer is that they paid attention to what they were doing while they were doing it—and then they tried to do it better.

Imagine that you are sitting in a meeting. Maybe you are thinking that the meeting could go on forever and that you would rather be programming. Dave and Andy would be thinking about why they were

having the meeting, and wondering if there is something else they could do that would take the place of the meeting, and deciding if that something could be automated so that the work of the meeting just happens in the future. Then they would do it.

That is just the way Dave and Andy think. That meeting wasn't something keeping them from programming. It *was* programming. And it was programming that could be improved. I know they think this way because it is tip number two: Think About Your Work.

So imagine that these guys are thinking this way for a few years. Pretty soon they would have a collection of solutions. Now imagine them using their solutions in their work for a few more years, and discarding the ones that are too hard or don't always produce results. Well, that approach just about defines *pragmatic*. Now imagine them taking a year or two more to write their solutions down. You might think, *That information would be a gold mine*. And you would be right.

The authors tell us how they program. And they tell us in a way that we can follow. But there is more to this second statement than you might think. Let me explain.

The authors have been careful to avoid proposing a theory of software development. This is fortunate, because if they had they would be obliged to warp each chapter to defend their theory. Such warping is the tradition in, say, the physical sciences, where theories eventually become laws or are quietly discarded. Programming on the other hand has few (if any) laws. So programming advice shaped around wanna-be laws may sound good in writing, but it fails to satisfy in practice. This is what goes wrong with so many methodology books.

I've studied this problem for a dozen years and found the most promise in a device called a *pattern language*. In short, a *pattern* is a solution, and a pattern language is a system of solutions that reinforce each other. A whole community has formed around the search for these systems.

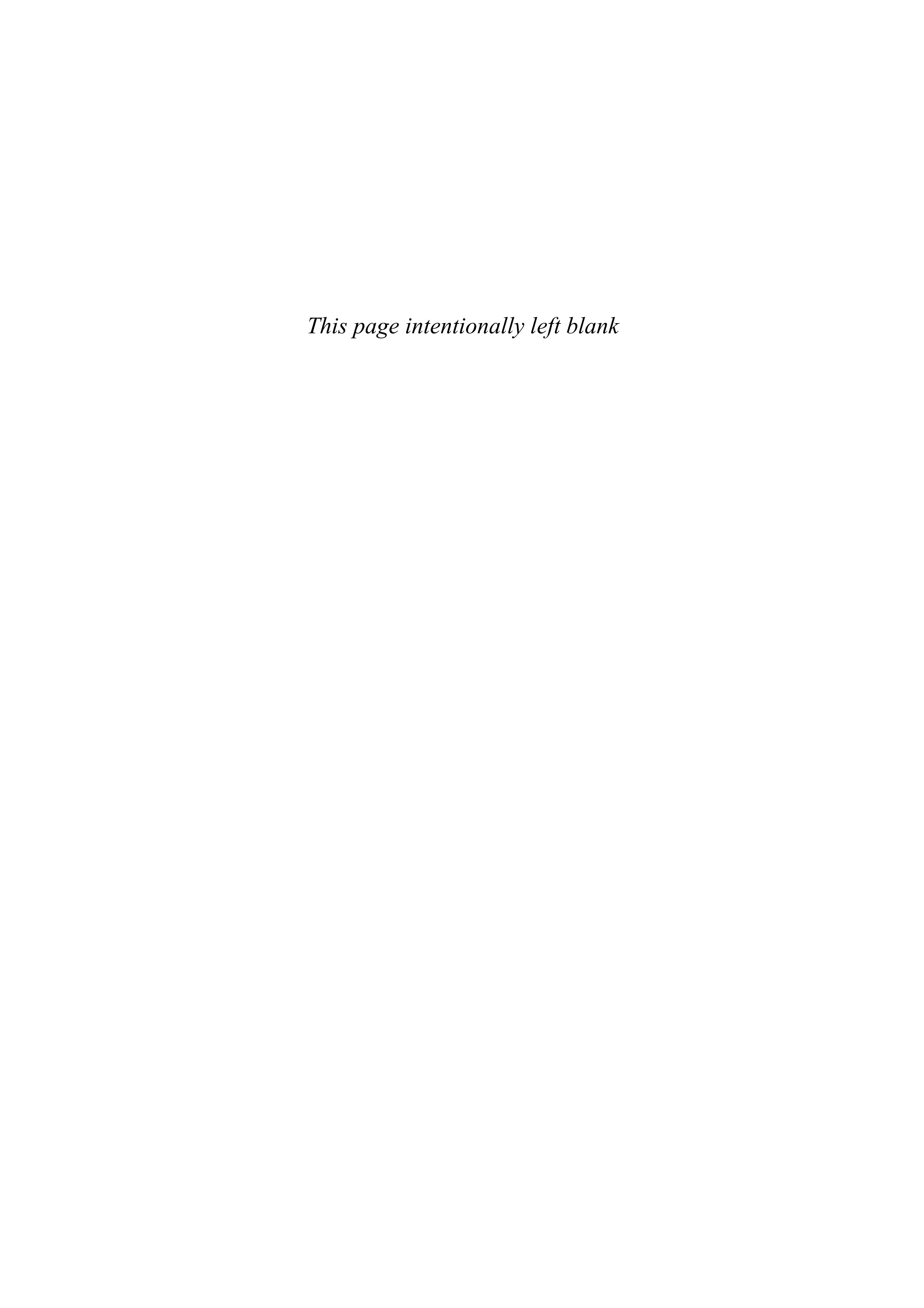
This book is more than a collection of tips. It is a pattern language in sheep's clothing. I say that because each tip is drawn from experience, told as concrete advice, and related to others to form a system. These are the characteristics that allow us to learn and follow a pattern language. They work the same way here.

You can follow the advice in this book because it is concrete. You won't find vague abstractions. Dave and Andy write directly for you, as if each tip was a vital strategy for energizing your programming career. They make it simple, they tell a story, they use a light touch, and then they follow that up with answers to questions that will come up when you try.

And there is more. After you read ten or fifteen tips you will begin to see an extra dimension to the work. We sometimes call it *QWAN*, short for the *quality without a name*. The book has a philosophy that will ooze into your consciousness and mix with your own. It doesn't preach. It just tells what works. But in the telling more comes through. That's the beauty of the book: It embodies its philosophy, and it does so unpretentiously.

So here it is: an easy to read—and use—book about the whole practice of programming. I've gone on and on about why it works. You probably only care that it does work. It does. You will see.

—Ward Cunningham


This page intentionally left blank

Preface

This book will help you become a better programmer.

It doesn't matter whether you are a lone developer, a member of a large project team, or a consultant working with many clients at once. This book will help you, as an individual, to do better work. This book isn't theoretical—we concentrate on practical topics, on using your experience to make more informed decisions. The word *pragmatic* comes from the Latin *pragmaticus*—"skilled in business"—which itself is derived from the Greek πραττειν, meaning "to do." This is a book about doing.

Programming is a craft. At its simplest, it comes down to getting a computer to do what you want it to do (or what your user wants it to do). As a programmer, you are part listener, part advisor, part interpreter, and part dictator. You try to capture elusive requirements and find a way of expressing them so that a mere machine can do them justice. You try to document your work so that others can understand it, and you try to engineer your work so that others can build on it. What's more, you try to do all this against the relentless ticking of the project clock. You work small miracles every day.

It's a difficult job.

There are many people offering you help. Tool vendors tout the miracles their products perform. Methodology gurus promise that their techniques guarantee results. Everyone claims that their programming language is the best, and every operating system is the answer to all conceivable ills.

Of course, none of this is true. There are no easy answers. There is no such thing as a *best* solution, be it a tool, a language, or an operating system. There can only be systems that are more appropriate in a particular set of circumstances.

This is where pragmatism comes in. You shouldn't be wedded to any particular technology, but have a broad enough background and experience base to allow you to choose good solutions in particular situations. Your background stems from an understanding of the basic principles of computer science, and your experience comes from a wide range of practical projects. Theory and practice combine to make you strong.

You adjust your approach to suit the current circumstances and environment. You judge the relative importance of all the factors affecting a project and use your experience to produce appropriate solutions. And you do this continuously as the work progresses. Pragmatic Programmers get the job done, and do it well.

Who Should Read This Book?

This book is aimed at people who want to become more effective and more productive programmers. Perhaps you feel frustrated that you don't seem to be achieving your potential. Perhaps you look at colleagues who seem to be using tools to make themselves more productive than you. Maybe your current job uses older technologies, and you want to know how newer ideas can be applied to what you do.

We don't pretend to have all (or even most) of the answers, nor are all of our ideas applicable in all situations. All we can say is that if you follow our approach, you'll gain experience rapidly, your productivity will increase, and you'll have a better understanding of the entire development process. And you'll write better software.

What Makes a Pragmatic Programmer?

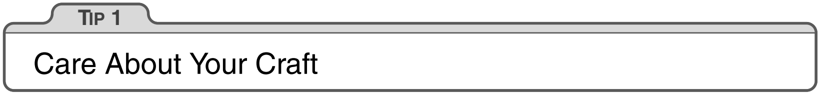
Each developer is unique, with individual strengths and weaknesses, preferences and dislikes. Over time, each will craft his or her own personal environment. That environment will reflect the programmer's individuality just as forcefully as his or her hobbies, clothing, or haircut. However, if you're a Pragmatic Programmer, you'll share many of the following characteristics:

- **Early adopter/fast adapter.** You have an instinct for technologies and techniques, and you love trying things out. When given some-

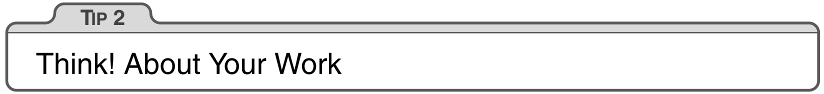
thing new, you can grasp it quickly and integrate it with the rest of your knowledge. Your confidence is born of experience.

- **Inquisitive.** You tend to ask questions. *That's neat—how did you do that? Did you have problems with that library? What's this BeOS I've heard about? How are symbolic links implemented?* You are a pack rat for little facts, each of which may affect some decision years from now.
- **Critical thinker.** You rarely take things as given without first getting the facts. When colleagues say "because that's the way it's done," or a vendor promises the solution to all your problems, you smell a challenge.
- **Realistic.** You try to understand the underlying nature of each problem you face. This realism gives you a good feel for how difficult things are, and how long things will take. Understanding for yourself that a process *should* be difficult or *will* take a while to complete gives you the stamina to keep at it.
- **Jack of all trades.** You try hard to be familiar with a broad range of technologies and environments, and you work to keep abreast of new developments. Although your current job may require you to be a specialist, you will always be able to move on to new areas and new challenges.

We've left the most basic characteristics until last. All Pragmatic Programmers share them. They're basic enough to state as tips:



We feel that there is no point in developing software unless you care about doing it well.



In order to be a Pragmatic Programmer, we're challenging you to think about what you're doing while you're doing it. This isn't a one-time audit of current practices—it's an ongoing critical appraisal of every

decision you make, every day, and on every development. Never run on auto-pilot. Constantly be thinking, critiquing your work in real time. The old IBM corporate motto, THINK!, is the Pragmatic Programmer's mantra.

If this sounds like hard work to you, then you're exhibiting the *realistic* characteristic. This is going to take up some of your valuable time—time that is probably already under tremendous pressure. The reward is a more active involvement with a job you love, a feeling of mastery over an increasing range of subjects, and pleasure in a feeling of continuous improvement. Over the long term, your time investment will be repaid as you and your team become more efficient, write code that's easier to maintain, and spend less time in meetings.

Individual Pragmatists, Large Teams

Some people feel that there is no room for individuality on large teams or complex projects. "Software construction is an engineering discipline," they say, "that breaks down if individual team members make decisions for themselves."

We disagree.

The construction of software *should* be an engineering discipline. However, this doesn't preclude individual craftsmanship. Think about the large cathedrals built in Europe during the Middle Ages. Each took thousands of person-years of effort, spread over many decades. Lessons learned were passed down to the next set of builders, who advanced the state of structural engineering with their accomplishments. But the carpenters, stonecutters, carvers, and glass workers were all craftspeople, interpreting the engineering requirements to produce a whole that transcended the purely mechanical side of the construction. It was their belief in their individual contributions that sustained the projects:

> *We who cut mere stones must always be envisioning cathedrals.*
>
> **— Quarry worker's creed**

Within the overall structure of a project there is always room for individuality and craftsmanship. This is particularly true given the current state of software engineering. One hundred years from now, our engineering may seem as archaic as the techniques used by medieval

cathedral builders seem to today's civil engineers, while our craftsmanship will still be honored.

It's a Continuous Process

> *A tourist visiting England's Eton College asked the gardener how he got the lawns so perfect. "That's easy," he replied, "You just brush off the dew every morning, mow them every other day, and roll them once a week."*
>
> *"Is that all?" asked the tourist.*
>
> *"Absolutely," replied the gardener. "Do that for 500 years and you'll have a nice lawn, too."*

Great lawns need small amounts of daily care, and so do great programmers. Management consultants like to drop the word *kaizen* in conversations. "Kaizen" is a Japanese term that captures the concept of continuously making many small improvements. It was considered to be one of the main reasons for the dramatic gains in productivity and quality in Japanese manufacturing and was widely copied throughout the world. Kaizen applies to individuals, too. Every day, work to refine the skills you have and to add new tools to your repertoire. Unlike the Eton lawns, you'll start seeing results in a matter of days. Over the years, you'll be amazed at how your experience has blossomed and your skills have grown.

How the Book Is Organized

This book is written as a collection of short sections. Each section is self-contained, and addresses a particular topic. You'll find numerous cross references, which help put each topic in context. Feel free to read the sections in any order—this isn't a book you need to read front-to-back.

Occasionally you'll come across a box labeled *Tip nn* (such as Tip 1, "Care About Your Craft" on page xix). As well as emphasizing points in the text, we feel the tips have a life of their own—we live by them daily. You'll find a summary of all the tips on a pull-out card inside the back cover.

Appendix A contains a set of resources: the book's bibliography, a list of URLs to Web resources, and a list of recommended periodicals, books, and professional organizations. Throughout the book you'll find references to the bibliography and to the list of URLs—such as [KP99] and [URL 18], respectively.

We've included exercises and challenges where appropriate. Exercises normally have relatively straightforward answers, while the challenges are more open-ended. To give you an idea of our thinking, we've included our answers to the exercises in Appendix B, but very few have a single *correct* solution. The challenges might form the basis of group discussions or essay work in advanced programming courses.

What's in a Name?

"When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean—neither more nor less."

► **Lewis Carroll, *Through the Looking-Glass***

Scattered throughout the book you'll find various bits of jargon—either perfectly good English words that have been corrupted to mean something technical, or horrendous made-up words that have been assigned meanings by computer scientists with a grudge against the language. The first time we use each of these jargon words, we try to define it, or at least give a hint to its meaning. However, we're sure that some have fallen through the cracks, and others, such as *object* and *relational database*, are in common enough usage that adding a definition would be boring. If you *do* come across a term you haven't seen before, please don't just skip over it. Take time to look it up, perhaps on the Web, or maybe in a computer science textbook. And, if you get a chance, drop us an e-mail and complain, so we can add a definition to the next edition.

Having said all this, we decided to get revenge against the computer scientists. Sometimes, there are perfectly good jargon words for concepts, words that we've decided to ignore. Why? Because the existing jargon is normally restricted to a particular problem domain, or to a particular phase of development. However, one of the basic philosophies of this book is that most of the techniques we're recommending are universal: modularity applies to code, designs, documentation, and team

organization, for instance. When we wanted to use the conventional jargon word in a broader context, it got confusing—we couldn't seem to overcome the baggage the original term brought with it. When this happened, we contributed to the decline of the language by inventing our own terms.

Source Code and Other Resources

Most of the code shown in this book is extracted from compilable source files, available for download from our Web site:

```
www.pragmaticprogrammer.com
```

There you'll also find links to resources we find useful, along with updates to the book and news of other Pragmatic Programmer developments.

Send Us Feedback

We'd appreciate hearing from you. Comments, suggestions, errors in the text, and problems in the examples are all welcome. E-mail us at

```
ppbook@pragmaticprogrammer.com
```

Acknowledgments

When we started writing this book, we had no idea how much of a team effort it would end up being.

Addison-Wesley has been brilliant, taking a couple of wet-behind-the-ears hackers and walking us through the whole book-production process, from idea to camera-ready copy. Many thanks to John Wait and Meera Ravindiran for their initial support, Mike Hendrickson, our enthusiastic editor (and a mean cover designer!), Lorraine Ferrier and John Fuller for their help with production, and the indefatigable Julie DeBaggis for keeping us all together.

Then there were the reviewers: Greg Andress, Mark Cheers, Chris Cleeland, Alistair Cockburn, Ward Cunningham, Martin Fowler, Thanh T. Giang, Robert L. Glass, Scott Henninger, Michael Hunter, Brian

Kirby, John Lakos, Pete McBreen, Carey P. Morris, Jared Richardson, Kevin Ruland, Eric Starr, Eric Vought, Chris Van Wyk, and Deborra Zukowski. Without their careful comments and valuable insights, this book would be less readable, less accurate, and twice as long. Thank you all for your time and wisdom.

The second printing of this book benefited greatly from the eagle eyes of our readers. Many thanks to Brian Blank, Paul Boal, Tom Ekberg, Brent Fulgham, Louis Paul Hebert, Henk-Jan Olde Loohuis, Alan Lund, Gareth McCaughan, Yoshiki Shibata, and Volker Wurst, both for finding the mistakes and for having the grace to point them out gently.

Over the years, we have worked with a large number of progressive clients, where we gained and refined the experience we write about here. Recently, we've been fortunate to work with Peter Gehrke on several large projects. His support and enthusiasm for our techniques are much appreciated.

This book was produced using LaTeX, pic, Perl, dvips, ghostview, ispell, GNU make, CVS, Emacs, XEmacs, EGCS, GCC, Java, iContract, and SmallEiffel, using the Bash and zsh shells under Linux. The staggering thing is that all of this tremendous software is freely available. We owe a huge "thank you" to the thousands of Pragmatic Programmers worldwide who have contributed these and other works to us all. We'd particularly like to thank Reto Kramer for his help with iContract.

Last, but in no way least, we owe a huge debt to our families. Not only have they put up with late night typing, huge telephone bills, and our permanent air of distraction, but they've had the grace to read what we've written, time after time. Thank you for letting us dream.

Andy Hunt
Dave Thomas

Chapter 3

The Basic Tools

Every craftsman starts his or her journey with a basic set of good-quality tools. A woodworker might need rules, gauges, a couple of saws, some good planes, fine chisels, drills and braces, mallets, and clamps. These tools will be lovingly chosen, will be built to last, will perform specific jobs with little overlap with other tools, and, perhaps most importantly, will feel right in the budding woodworker's hands.

Then begins a process of learning and adaptation. Each tool will have its own personality and quirks, and will need its own special handling. Each must be sharpened in a unique way, or held just so. Over time, each will wear according to use, until the grip looks like a mold of the woodworker's hands and the cutting surface aligns perfectly with the angle at which the tool is held. At this point, the tools become conduits from the craftsman's brain to the finished product—they have become extensions of his or her hands. Over time, the woodworker will add new tools, such as biscuit cutters, laser-guided miter saws, dovetail jigs—all wonderful pieces of technology. But you can bet that he or she will be happiest with one of those original tools in hand, feeling the plane sing as it slides through the wood.

Tools amplify your talent. The better your tools, and the better you know how to use them, the more productive you can be. Start with a basic set of generally applicable tools. As you gain experience, and as you come across special requirements, you'll add to this basic set. Like the craftsman, expect to add to your toolbox regularly. Always be on the lookout for better ways of doing things. If you come across a situation where you feel your current tools can't cut it, make a note to look for

something different or more powerful that would have helped. Let need drive your acquisitions.

Many new programmers make the mistake of adopting a single power tool, such as a particular integrated development environment (IDE), and never leave its cozy interface. This really is a mistake. We need to be comfortable beyond the limits imposed by an IDE. The only way to do this is to keep the basic tool set sharp and ready to use.

In this chapter we'll talk about investing in your own basic toolbox. As with any good discussion on tools, we'll start (in *The Power of Plain Text*) by looking at your raw materials, the stuff you'll be shaping. From there we'll move to the workbench, or in our case the computer. How can you use your computer to get the most out of the tools you use? We'll discuss this in *Shell Games*. Now that we have material and a bench to work on, we'll turn to the tool you'll probably use more than any other, your editor. In *Power Editing*, we'll suggest ways of making you more efficient.

To ensure that we never lose any of our precious work, we should always use a *Source Code Control* system—even for things such as our personal address book! And, since Mr. Murphy was really an optimist after all, you can't be a great programmer until you become highly skilled at *Debugging*.

You'll need some glue to bind much of the magic together. We discuss some possibilities, such as awk, Perl, and Python, in *Text Manipulation*.

Just as woodworkers sometimes build jigs to guide the construction of complex pieces, programmers can write code that itself writes code. We discuss this in *Code Generators*.

Spend time learning to use these tools, and at some point you'll be surprised to discover your fingers moving over the keyboard, manipulating text without conscious thought. The tools will have become extensions of your hands.

14

The Power of Plain Text

As Pragmatic Programmers, our base material isn't wood or iron, it's knowledge. We gather requirements as knowledge, and then express that knowledge in our designs, implementations, tests, and documents. And we believe that the best format for storing knowledge persistently is *plain text*. With plain text, we give ourselves the ability to manipulate knowledge, both manually and programmatically, using virtually every tool at our disposal.

What Is Plain Text?

Plain text is made up of printable characters in a form that can be read and understood directly by people. For example, although the following snippet is made up of printable characters, it is meaningless.

```
Field19=467abe
```

The reader has no idea what the significance of `467abe` may be. A better choice would be to make it *understandable* to humans.

```
DrawingType=UMLActivityDrawing
```

Plain text doesn't mean that the text is unstructured; XML, SGML, and HTML are great examples of plain text that has a well-defined structure. You can do everything with plain text that you could do with some binary format, including versioning.

Plain text tends to be at a higher level than a straight binary encoding, which is usually derived directly from the implementation. Suppose you wanted to store a property called `uses_menus` that can be either `TRUE` or `FALSE`. Using text, you might write this as

```
myprop.uses_menus=FALSE
```

Contrast this with `0010010101110101`.

The problem with most binary formats is that the context necessary to understand the data is separate from the data itself. You are artificially divorcing the data from its meaning. The data may as well be encrypted; it is absolutely meaningless without the application logic to parse it. With plain text, however, you can achieve a self-describing data stream that is independent of the application that created it.

TIP 20

Keep Knowledge in Plain Text

Drawbacks

There are two major drawbacks to using plain text: (1) It may take more space to store than a compressed binary format, and (2) it may be computationally more expensive to interpret and process a plain text file.

Depending on your application, either or both of these situations may be unacceptable—for example, when storing satellite telemetry data, or as the internal format of a relational database.

But even in these situations, it may be acceptable to store *metadata* about the raw data in plain text (see *Metaprogramming*, page 144).

Some developers may worry that by putting metadata in plain text, they're exposing it to the system's users. This fear is misplaced. Binary data may be more obscure than plain text, but it is no more secure. If you worry about users seeing passwords, encrypt them. If you don't want them changing configuration parameters, include a *secure hash*[1] of all the parameter values in the file as a checksum.

The Power of Text

Since *larger* and *slower* aren't the most frequently requested features from users, why bother with plain text? What *are* the benefits?

- Insurance against obsolescence
- Leverage
- Easier testing

Insurance Against Obsolescence

Human-readable forms of data, and self-describing data, will outlive all other forms of data and the applications that created them. Period.

1. MD5 is often used for this purpose. For an excellent introduction to the wonderful world of cryptography, see [Sch95].

As long as the data survives, you will have a chance to be able to use it—potentially long after the original application that wrote it is defunct.

You can parse such a file with only partial knowledge of its format; with most binary files, you must know all the details of the entire format in order to parse it successfully.

Consider a data file from some legacy system[2] that you are given. You know little about the original application; all that's important to you is that it maintained a list of clients' Social Security numbers, which you need to find and extract. Among the data, you see

```
<FIELD10>123-45-6789</FIELD10>
...
<FIELD10>567-89-0123</FIELD10>
...
<FIELD10>901-23-4567</FIELD10>
```

Recognizing the format of a Social Security number, you can quickly write a small program to extract that data—even if you have no information on anything else in the file.

But imagine if the file had been formatted this way instead:

```
AC27123456789B11P
...
XY43567890123QTYL
...
6T2190123456788AM
```

You may not have recognized the significance of the numbers quite as easily. This is the difference between *human readable* and *human understandable*.

While we're at it, `FIELD10` doesn't help much either. Something like

```
<SSNO>123-45-6789</SSNO>
```

makes the exercise a no-brainer—and ensures that the data will outlive any project that created it.

Leverage

Virtually every tool in the computing universe, from source code management systems to compiler environments to editors and stand-alone filters, can operate on plain text.

---

2. All software becomes legacy as soon as it's written.

The Unix Philosophy

Unix is famous for being designed around the philosophy of small, sharp tools, each intended to do one thing well. This philosophy is enabled by using a common underlying format—the line-oriented, plain text file. Databases used for system administration (users and passwords, networking configuration, and so on) are all kept as plain text files. (Some systems, such as Solaris, also maintain a binary form of certain databases as a performance optimization. The plain text version is kept as an interface to the binary version.)

When a system crashes, you may be faced with only a minimal environment to restore it (you may not be able to access graphics drivers, for instance). Situations such as this can really make you appreciate the simplicity of plain text.

For instance, suppose you have a production deployment of a large application with a complex site-specific configuration file (`sendmail` comes to mind). If this file is in plain text, you could place it under a source code control system (see *Source Code Control*, page 86), so that you automatically keep a history of all changes. File comparison tools such as `diff` and `fc` allow you to see at a glance what changes have been made, while `sum` allows you to generate a checksum to monitor the file for accidental (or malicious) modification.

Easier Testing

If you use plain text to create synthetic data to drive system tests, then it is a simple matter to add, update, or modify the test data *without having to create any special tools to do so.* Similarly, plain text output from regression tests can be trivially analyzed (with `diff`, for instance) or subjected to more thorough scrutiny with Perl, Python, or some other scripting tool.

Lowest Common Denominator

Even in the future of XML-based intelligent agents that travel the wild and dangerous Internet autonomously, negotiating data interchange among themselves, the ubiquitous text file will still be there. In fact, in

heterogeneous environments the advantages of plain text can outweigh all of the drawbacks. You need to ensure that all parties can communicate using a common standard. Plain text is that standard.

Related sections include:

- *Source Code Control*, page 86
- *Code Generators*, page 102
- *Metaprogramming*, page 144
- *Blackboards*, page 165
- *Ubiquitous Automation*, page 230
- *It's All Writing*, page 248

Challenges

- Design a small address book database (name, phone number, and so on) using a straightforward binary representation in your language of choice. Do this before reading the rest of this challenge.

    1. Translate that format into a plain text format using XML.

    2. For each version, add a new, variable-length field called *directions* in which you might enter directions to each person's house.

    What issues come up regarding versioning and extensibility? Which form was easier to modify? What about converting existing data?

15 Shell Games

Every woodworker needs a good, solid, reliable workbench, somewhere to hold work pieces at a convenient height while he or she works them. The workbench becomes the center of the wood shop, the craftsman returning to it time and time again as a piece takes shape.

For a programmer manipulating files of text, that workbench is the command shell. From the shell prompt, you can invoke your full repertoire of tools, using pipes to combine them in ways never dreamt of by their original developers. From the shell, you can launch applications, debuggers, browsers, editors, and utilities. You can search for files,

query the status of the system, and filter output. And by programming the shell, you can build complex macro commands for activities you perform often.

For programmers raised on GUI interfaces and integrated development environments (IDEs), this might seem an extreme position. After all, can't you do everything equally well by pointing and clicking?

The simple answer is "no." GUI interfaces are wonderful, and they can be faster and more convenient for some simple operations. Moving files, reading MIME-encoded e-mail, and typing letters are all things that you might want to do in a graphical environment. But if you do all your work using GUIs, you are missing out on the full capabilities of your environment. You won't be able to automate common tasks, or use the full power of the tools available to you. And you won't be able to combine your tools to create customized *macro tools*. A benefit of GUIs is WYSIWYG—what you see is what you get. The disadvantage is WYSIAYG—what you see is *all* you get.

GUI environments are normally limited to the capabilities that their designers intended. If you need to go beyond the model the designer provided, you are usually out of luck—and more often than not, you *do* need to go beyond the model. Pragmatic Programmers don't just cut code, or develop object models, or write documentation, or automate the build process—we do *all* of these things. The scope of any one tool is usually limited to the tasks that the tool is expected to perform. For instance, suppose you need to integrate a code preprocessor (to implement design-by-contract, or multi-processing pragmas, or some such) into your IDE. Unless the designer of the IDE explicitly provided hooks for this capability, you can't do it.

You may already be comfortable working from the command prompt, in which case you can safely skip this section. Otherwise, you may need to be convinced that the shell is your friend.

As a Pragmatic Programmer, you will constantly want to perform ad hoc operations—things that the GUI may not support. The command line is better suited when you want to quickly combine a couple of commands to perform a query or some other task. Here are a few examples.

Find all .c files modified more recently than your Makefile.

Shell ... `find . -name '*.c' -newer Makefile -print`

GUI ..... Open the Explorer, navigate to the correct directory, click on the Makefile, and note the modification time. Then bring up Tools/Find, and enter `*.c` for the file specification. Select the date tab, and enter the date you noted for the Makefile in the first date field. Then hit OK.

Construct a zip/tar archive of my source.

Shell ... `zip archive.zip *.h *.c` – *or* –
`tar cvf archive.tar *.h *.c`

GUI ..... Bring up a ZIP utility (such as the shareware WinZip [URL 41]), select "Create New Archive," enter its name, select the source directory in the add dialog, set the filter to "`*.c`", click "Add," set the filter to "`*.h`", click "Add," then close the archive.

Which Java files have not been changed in the last week?

Shell ... `find . -name '*.java' -mtime +7 -print`

GUI ..... Click and navigate to "Find files," click the "Named" field and type in "*.java", select the "Date Modified" tab. Then select "Between." Click on the starting date and type in the starting date of the beginning of the project. Click on the ending date and type in the date of a week ago today (be sure to have a calendar handy). Click on "Find Now."

Of those files, which use the `awt` libraries?

Shell ...
```
find . -name '*.java' -mtime +7 -print |
       xargs grep 'java.awt'
```

GUI ..... Load each file in the list from the previous example into an editor and search for the string "java.awt". Write down the name of each file containing a match.

Clearly the list could go on. The shell commands may be obscure or terse, but they are powerful and concise. And, because shell commands can be combined into script files (or command files under Windows

systems), you can build sequences of commands to automate things you do often.

Tip 21

Use the Power of Command Shells

Gain familiarity with the shell, and you'll find your productivity soaring. Need to create a list of all the unique package names explicitly imported by your Java code? The following stores it in a file called "list."

```
grep '^import ' *.java |
  sed -e's/.*import  *//' -e's/;.*$//' |
  sort -u >list
```

If you haven't spent much time exploring the capabilities of the command shell on the systems you use, this might appear daunting. However, invest some energy in becoming familiar with your shell and things will soon start falling into place. Play around with your command shell, and you'll be surprised at how much more productive it makes you.

Shell Utilities and Windows Systems

Although the command shells provided with Windows systems are improving gradually, Windows command-line utilities are still inferior to their Unix counterparts. However, all is not lost.

Cygnus Solutions has a package called Cygwin [URL 31]. As well as providing a Unix compatibility layer for Windows, Cygwin comes with a collection of more than 120 Unix utilities, including such favorites as `ls`, `grep`, and `find`. The utilities and libraries may be downloaded and used for free, but be sure to read their license.[3] The Cygwin distribution comes with the Bash shell.

---

3. The GNU General Public License [URL 57] is a kind of legal virus that Open Source developers use to protect their (and your) rights. You should spend some time reading it. In essence, it says that you can use and modify GPL'd software, but if you distribute any modifications they must be licensed according to the GPL (and marked as such), and you must make source available. That's the virus part—whenever you derive a work from a GPL'd work, your derived work must also be GPL'd. However, it does not limit you in any way when simply using the tools—the ownership and licensing of software developed using the tools are up to you.

Using Unix Tools Under Windows

We love the availability of high-quality Unix tools under Windows, and use them daily. However, be aware that there are integration issues. Unlike their MS-DOS counterparts, these utilities are sensitive to the case of filenames, so `ls a*.bat` won't find `AUTOEXEC.BAT`. You may also come across problems with filenames containing spaces, and with differences in path separators. Finally, there are interesting problems when running MS-DOS programs that expect MS-DOS–style arguments under the Unix shells. For example, the Java utilities from JavaSoft use a colon as their CLASSPATH separator under Unix, but use a semicolon under MS-DOS. As a result, a Bash or ksh script that runs on a Unix box will run identically under Windows, but the command line it passes to Java will be interpreted incorrectly.

Alternatively, David Korn (of Korn shell fame) has put together a package called UWIN. This has the same aims as the Cygwin distribution—it is a Unix development environment under Windows. UWIN comes with a version of the Korn shell. Commercial versions are available from Global Technologies, Ltd. [URL 30]. In addition, AT&T allows free downloading of the package for evaluation and academic use. Again, read their license before using.

Finally, Tom Christiansen is (at the time of writing) putting together *Perl Power Tools*, an attempt to implement all the familiar Unix utilities portably, in Perl [URL 32].

Related sections include:

- *Ubiquitous Automation*, page 230

Challenges

- Are there things that you're currently doing manually in a GUI? Do you ever pass instructions to colleagues that involve a number of individual "click this button," "select this item" steps? Could these be automated?
- Whenever you move to a new environment, make a point of finding out what shells are available. See if you can bring your current shell with you.
- Investigate alternatives to your current shell. If you come across a problem your shell can't address, see if an alternative shell would cope better.

16

Power Editing

We've talked before about tools being an extension of your hand. Well, this applies to editors more than to any other software tool. You need to be able to manipulate text as effortlessly as possible, because text is the basic raw material of programming. Let's look at some common features and functions that help you get the most from your editing environment.

One Editor

We think it is better to know one editor very well, and use it for all editing tasks: code, documentation, memos, system administration, and so on. Without a single editor, you face a potential modern day Babel of confusion. You may have to use the built-in editor in each language's IDE for coding, and an all-in-one office product for documentation, and maybe a different built-in editor for sending e-mail. Even the keystrokes you use to edit command lines in the shell may be different.[4] It is difficult to be proficient in any of these environments if you have a different set of editing conventions and commands in each.

You need to be proficient. Simply typing linearly and using a mouse to cut and paste is not enough. You just can't be as effective that way as you can with a powerful editor under your fingers. Typing ← or BACKSPACE ten times to move the cursor left to the beginning of a line isn't as efficient as typing a single key such as ^A, Home, or 0.

TIP 22

Use a Single Editor Well

Choose an editor, know it thoroughly, and use it for all editing tasks. If you use a single editor (or set of keybindings) across all text editing activities, you don't have to stop and think to accomplish text manipulation: the necessary keystrokes will be a reflex. The editor will be

4. Ideally, the shell you use should have keybindings that match the ones used by your editor. Bash, for instance, supports both `vi` and `emacs` keybindings.

an extension of your hand; the keys will sing as they slice their way through text and thought. That's our goal.

Make sure that the editor you choose is available on all platforms you use. Emacs, vi, CRiSP, Brief, and others are available across multiple platforms, often in both GUI and non-GUI (text screen) versions.

Editor Features

Beyond whatever features you find particularly useful and comfortable, here are some basic abilities that we think every decent editor should have. If your editor falls short in any of these areas, then this may be the time to consider moving on to a more advanced one.

- **Configurable.** All aspects of the editor should be configurable to your preferences, including fonts, colors, window sizes, and keystroke bindings (which keys perform what commands). Using only keystrokes for common editing operations is more efficient than mouse or menu-driven commands, because your hands never leave the keyboard.
- **Extensible.** An editor shouldn't be obsolete just because a new programming language comes out. It should be able to integrate with whatever compiler environment you are using. You should be able to "teach" it the nuances of any new language or text format (XML, HTML version 9, and so on).
- **Programmable.** You should be able to program the editor to perform complex, multistep tasks. This can be done with macros or with a built-in scripting programming language (Emacs uses a variant of Lisp, for instance).

In addition, many editors support features that are specific to a particular programming language, such as:

- Syntax highlighting
- Auto-completion
- Auto-indentation
- Initial code or document boilerplate
- Tie-in to help systems
- IDE-like features (compile, debug, and so on)

Figure 3.1. Sorting lines in an editor

```
import java.util.Vector;      emacs: M-x sort-lines      import java.awt.*;
import java.util.Stack;       ------------------->       import java.net.URL;
import java.net.URL;          ------------------->       import java.util.Stack;
import java.awt.*;            vi: :.,+3!sort             import java.util.Vector;
```

A feature such as syntax highlighting may sound like a frivolous extra, but in reality it can be very useful and enhance your productivity. Once you get used to seeing keywords appear in a different color or font, a mistyped keyword that *doesn't* appear that way jumps out at you long before you fire up the compiler.

Having the ability to compile and navigate directly to errors within the editor environment is very handy on big projects. Emacs in particular is adept at this style of interaction.

Productivity

A surprising number of people we've met use the Windows `notepad` utility to edit their source code. This is like using a teaspoon as a shovel—simply typing and using basic mouse-based cut and paste is not enough.

What sort of things will you need to do that *can't* be done in this way?

Well, there's cursor movement, to start with. Single keystrokes that move you in units of words, lines, blocks, or functions are far more efficient than repeatedly typing a keystroke that moves you character by character or line by line.

Or suppose you are writing Java code. You like to keep your `import` statements in alphabetical order, and someone else has checked in a few files that don't adhere to this standard (this may sound extreme, but on a large project it can save you a lot of time scanning through a long list of `import` statements). You'd like to go quickly through a few files and sort a small section of them. In editors such as vi and Emacs you can do this easily (see Figure 3.1). Try *that* in `notepad`.

Some editors can help streamline common operations. For instance, when you create a new file in a particular language, the editor can supply a template for you. It might include:

- Name of the class or module filled in (derived from the filename)
- Your name and/or copyright statements
- Skeletons for constructs in that language (constructor and destructor declarations, for example)

Another useful feature is auto-indenting. Rather than having to indent manually (by using space or tab), the editor automatically indents for you at the appropriate time (after typing an open brace, for example). The nice part about this feature is that you can use the editor to provide a consistent indentation style for your project.[5]

Where to Go from Here

This sort of advice is particularly hard to write because virtually every reader is at a different level of comfort and expertise with the editor(s) they are currently using. So, to summarize, and to provide some guidance on where to go next, find yourself in the left-hand column of the chart, and look at the right-hand column to see what we think you should do.

If this sounds like you...	**Then think about...**
I use only basic features of many different editors.	Pick a powerful editor and learn it well.
I have a favorite editor, but I don't use all of its features.	Learn them. Cut down the number of keystrokes you need to type.
I have a favorite editor and use it where possible.	Try to expand and use it for more tasks than you do already.
I think you are nuts. Notepad is the best editor ever made.	As long as you are happy and productive, go for it! But if you find yourself subject to "editor envy," you may need to reevaluate your position.

5. The Linux kernel is developed this way. Here you have geographically dispersed developers, many working on the same pieces of code. There is a published list of settings (in this case, for Emacs) that describes the required indentation style.

What Editors Are Available?

Having recommended that you master a decent editor, which one do we recommend? Well, we're going to duck that question; your choice of editor is a personal one (some would even say a religious one!). However, in Appendix A, page 266, we list a number of popular editors and where to get them.

Challenges

- Some editors use full-blown languages for customization and scripting. Emacs, for example, uses Lisp. As one of the new languages you are going to learn this year, learn the language your editor uses. For anything you find yourself doing repeatedly, develop a set of macros (or equivalent) to handle it.
- Do you know everything your editor is capable of doing? Try to stump your colleagues who use the same editor. Try to accomplish any given editing task in as few keystrokes as possible.

Source Code Control

Progress, far from consisting in change, depends on retentiveness. Those who cannot remember the past are condemned to repeat it.

► **George Santayana, *Life of Reason***

One of the important things we look for in a user interface is the UNDO key—a single button that forgives us our mistakes. It's even better if the environment supports multiple levels of undo and redo, so you can go back and recover from something that happened a couple of minutes ago. But what if the mistake happened last week, and you've turned your computer on and off ten times since then? Well, that's one of the many benefits of using a source code control system: it's a giant UNDO key—a project-wide time machine that can return you to those halcyon days of last week, when the code actually compiled and ran.

Source code control systems, or the more widely scoped *configuration management* systems, keep track of every change you make in your source code and documentation. The better ones can keep track of

compiler and OS versions as well. With a properly configured source code control system, *you can always go back to a previous version of your software.*

But a source code control system (SCCS[6]) does far more than undo mistakes. A good SCCS will let you track changes, answering questions such as: Who made changes in this line of code? What's the difference between the current version and last week's? How many lines of code did we change in this release? Which files get changed most often? This kind of information is invaluable for bug-tracking, audit, performance, and quality purposes.

An SCCS will also let you identify releases of your software. Once identified, you will always be able to go back and regenerate the release, independent of changes that may have occurred later.

We often use an SCCS to manage branches in the development tree. For example, once you have released some software, you'll normally want to continue developing for the next release. At the same time, you'll need to deal with bugs in the current release, shipping fixed versions to clients. You'll want these bug fixes rolled into the next release (if appropriate), but you don't want to ship code under development to clients. With an SCCS you can generate branches in the development tree each time you generate a release. You apply bug fixes to code in the branch, and continue developing on the main trunk. Since the bug fixes may be relevant to the main trunk as well, some systems allow you to merge selected changes from the branch back into the main trunk automatically.

Source code control systems may keep the files they maintain in a central repository—a great candidate for archiving.

Finally, some products may allow two or more users to be working concurrently on the same set of files, even making concurrent changes in the same file. The system then manages the merging of these changes when the files are sent back to the repository. Although seemingly risky, such systems work well in practice on projects of all sizes.

---

6. We use the uppercase SCCS to refer to generic source code control systems. There is also a specific system called "sccs," originally released with AT&T System V Unix.

TIP 23

Always Use Source Code Control

Always. Even if you are a single-person team on a one-week project. Even if it's a "throw-away" prototype. Even if the stuff you're working on isn't source code. Make sure that *everything* is under source code control—documentation, phone number lists, memos to vendors, makefiles, build and release procedures, that little shell script that burns the CD master—everything. We routinely use source code control on just about everything we type (including the text of this book). Even if we're not working on a project, our day-to-day work is secured in a repository.

Source Code Control and Builds

There is a tremendous hidden benefit in having an entire project under the umbrella of a source code control system: you can have product builds that are *automatic* and *repeatable*.

The project build mechanism can pull the latest source out of the repository automatically. It can run in the middle of the night after everyone's (hopefully) gone home. You can run automatic regression tests to ensure that the day's coding didn't break anything. The automation of the build ensures consistency—there are no manual procedures, and you won't need developers remembering to copy code into some special build area.

The build is repeatable because you can always rebuild the source as it existed on a given date.

But My Team Isn't Using Source Code Control

Shame on them! Sounds like an opportunity to do some evangelizing! However, while you wait for them to see the light, perhaps you should implement your own private source control. Use one of the freely available tools we list in Appendix A, and make a point of keeping your personal work safely tucked into a repository (as well as doing whatever your project requires). Although this may seem to be duplication of effort, we can pretty much guarantee it will save you grief (and save your project money) the first time you need to answer questions such

as "What did you do to the *xyz* module?" and "What broke the build?" This approach may also help convince your management that source code control really works.

Don't forget that an SCCS is equally applicable to the things you do outside of work.

Source Code Control Products

Appendix A, page 271, gives URLs for representative source code control systems, some commercial and others freely available. And many more products are available—look for pointers to the configuration management FAQ. For an introduction to the freely-available CVS version control system, see our book *Pragmatic Version Control* [TH03].

Related sections include:

- *Orthogonality*, page 34
- *The Power of Plain Text*, page 73
- *It's All Writing*, page 248

Challenges

- Even if you are not able to use an SCCS at work, install RCS or CVS on a personal system. Use it to manage your pet projects, documents you write, and (possibly) configuration changes applied to the computer system itself.

- Take a look at some of the Open Source projects for which publicly accessible archives are available on the Web (such as Mozilla [URL 51], KDE [URL 54], and the Gimp [URL 55]). How do you get updates of the source? How do you make changes—does the project regulate access or arbitrate the inclusion of changes?

18

Debugging

It is a painful thing
To look at your own trouble and know
That you yourself and no one else has made it
► **Sophocles, *Ajax***

The word *bug* has been used to describe an "object of terror" ever since the fourteenth century. Rear Admiral Dr. Grace Hopper, the inventor of COBOL, is credited with observing the first *computer bug*—literally, a moth caught in a relay in an early computer system. When asked to explain why the machine wasn't behaving as intended, a technician reported that there was "a bug in the system," and dutifully taped it—wings and all—into the log book.

Regrettably, we still have "bugs" in the system, albeit not the flying kind. But the fourteenth century meaning—a bogeyman—is perhaps even more applicable now than it was then. Software defects manifest themselves in a variety of ways, from misunderstood requirements to coding errors. Unfortunately, modern computer systems are still limited to doing what you *tell* them to do, not necessarily what you *want* them to do.

No one writes perfect software, so it's a given that debugging will take up a major portion of your day. Let's look at some of the issues involved in debugging and some general strategies for finding elusive bugs.

Psychology of Debugging

Debugging itself is a sensitive, emotional subject for many developers. Instead of attacking it as a puzzle to be solved, you may encounter denial, finger pointing, lame excuses, or just plain apathy.

Embrace the fact that debugging is just *problem solving*, and attack it as such.

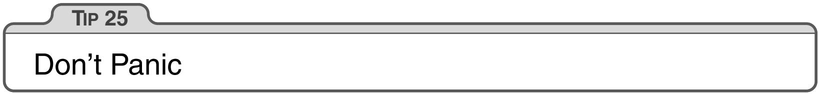
Having found someone else's bug, you can spend time and energy laying blame on the filthy culprit who created it. In some workplaces this is part of the culture, and may be cathartic. However, in the technical arena, you want to concentrate on fixing the *problem*, not the blame.

It doesn't really matter whether the bug is your fault or someone else's. It is still your problem.

A Debugging Mindset

The easiest person to deceive is one's self.
► **Edward Bulwer-Lytton, *The Disowned***

Before you start debugging, it's important to adopt the right mindset. You need to turn off many of the defenses you use each day to protect your ego, tune out any project pressures you may be under, and get yourself comfortable. Above all, remember the first rule of debugging:



It's easy to get into a panic, especially if you are facing a deadline, or have a nervous boss or client breathing down your neck while you are trying to find the cause of the bug. But it is very important to step back a pace, and actually *think* about what could be causing the symptoms that you believe indicate a bug.

If your first reaction on witnessing a bug or seeing a bug report is "that's impossible," you are plainly wrong. Don't waste a single neuron on the train of thought that begins "but that can't happen" because quite clearly it *can*, and has.

Beware of myopia when debugging. Resist the urge to fix just the symptoms you see: it is more likely that the actual fault may be several steps removed from what you are observing, and may involve a number of other related things. Always try to discover the root cause of a problem, not just this particular appearance of it.

Where to Start

Before you *start* to look at the bug, make sure that you are working on code that compiled cleanly—without warnings. We routinely set

compiler warning levels as high as possible. It doesn't make sense to waste time trying to find a problem that the compiler could find for you! We need to concentrate on the harder problems at hand.

When trying to solve any problem, you need to gather all the relevant data. Unfortunately, bug reporting isn't an exact science. It's easy to be misled by coincidences, and you can't afford to waste time debugging coincidences. You first need to be accurate in your observations.

Accuracy in bug reports is further diminished when they come through a third party—you may actually need to *watch* the user who reported the bug in action to get a sufficient level of detail.

Andy once worked on a large graphics application. Nearing release, the testers reported that the application crashed every time they painted a stroke with a particular brush. The programmer responsible argued that there was nothing wrong with it; he had tried painting with it, and it worked just fine. This dialog went back and forth for several days, with tempers rapidly rising.

Finally, we got them together in the same room. The tester selected the brush tool and painted a stroke from the upper right corner to the lower left corner. The application exploded. "Oh," said the programmer, in a small voice, who then sheepishly admitted that he had made test strokes only from the lower left to the upper right, which did not expose the bug.

There are two points to this story:

- You may need to interview the user who reported the bug in order to gather more data than you were initially given.
- Artificial tests (such as the programmer's single brush stroke from bottom to top) don't exercise enough of an application. You must brutally test both boundary conditions and realistic end-user usage patterns. You need to do this systematically (see *Ruthless Testing*, page 237).

Debugging Strategies

Once *you* think you know what is going on, it's time to find out what the *program* thinks is going on.

Bug Reproduction

No, our bugs aren't really multiplying (although some of them are probably old enough to do it legally). We're talking about a different kind of reproduction.

The best way to start fixing a bug is to make it reproducible. After all, if you can't reproduce it, how will you know if it is ever fixed?

But we want more than a bug that can be reproduced by following some long series of steps; we want a bug that can be reproduced with a *single command*. It's a lot harder to fix a bug if you have to go through 15 steps to get to the point where the bug shows up. Sometimes by forcing yourself to isolate the circumstances that display the bug, you'll even gain an insight on how to fix it.

See *Ubiquitous Automation*, page 230, for other ideas along these lines.

Visualize Your Data

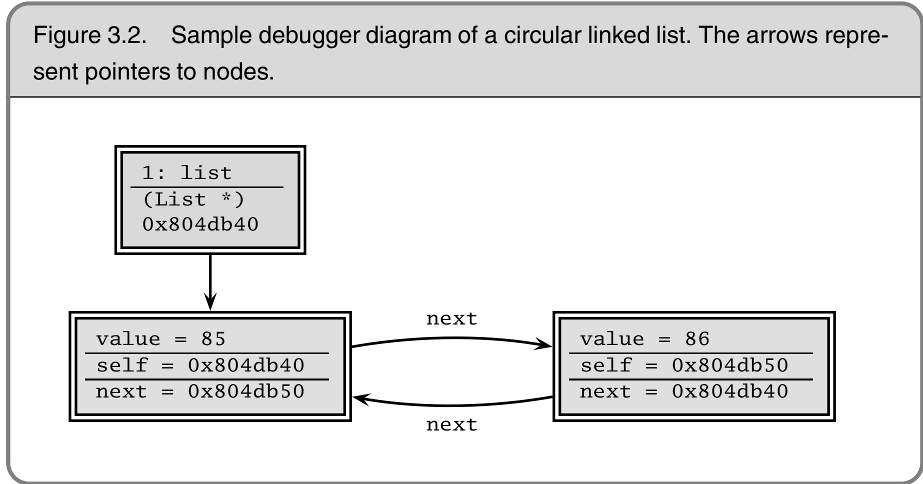
Often, the easiest way to discern what a program is doing—or what it is going to do—is to get a good look at the data it is operating on. The simplest example of this is a straightforward "`variable name = data value`" approach, which may be implemented as printed text, or as fields in a GUI dialog box or list.

But you can gain a much deeper insight into your data by using a debugger that allows you to *visualize* your data and all of the interrelationships that exist. There are debuggers that can represent your data as a 3D fly-over through a virtual reality landscape, or as a 3D waveform plot, or just as simple structural diagrams, as shown in Figure 3.2 on the next page. As you single-step through your program, pictures like these can be worth much more than a thousand words, as the bug you've been hunting suddenly jumps out at you.

Even if your debugger has limited support for visualizing data, you can still do it yourself—either by hand, with paper and pencil, or with external plotting programs.

The DDD debugger has some visualization capabilities, and is freely available (see [URL 19]). It is interesting to note that DDD works with

Figure 3.2. Sample debugger diagram of a circular linked list. The arrows represent pointers to nodes.



multiple languages, including Ada, C, C++, Fortran, Java, Modula, Pascal, Perl, and Python (clearly an orthogonal design).

Tracing

Debuggers generally focus on the state of the program *now*. Sometimes you need more—you need to watch the state of a program or a data structure over time. Seeing a stack trace can only tell you how you got here directly. It can't tell you what you were doing prior to this call chain, especially in event-based systems.

Tracing statements are those little diagnostic messages you print to the screen or to a file that say things such as "got here" and "value of x = 2." It's a primitive technique compared with IDE-style debuggers, but it is peculiarly effective at diagnosing several classes of errors that debuggers can't. Tracing is invaluable in any system where time itself is a factor: concurrent processes, real-time systems, and event-based applications.

You can use tracing statements to "drill down" into the code. That is, you can add tracing statements as you descend the call tree.

Trace messages should be in a regular, consistent format; you may want to parse them automatically. For instance, if you needed to track down a resource leak (such as unbalanced file opens/closes), you could trace each `open` and each `close` in a log file. By processing the log

Corrupt Variables? Check Their Neighborhood

Sometimes you'll examine a variable, expecting to see a small integer value, and instead get something like `0x6e69614d`. Before you roll up your sleeves for some serious debugging, have a quick look at the memory around this corrupted variable. Often it will give you a clue. In our case, examining the surrounding memory as characters shows us

```
20333231 6e69614d 2c745320 746f4e0a
 1 2 3    M a i n    S t , \n N o t
2c6e776f 2058580a 31323433 00000a33
 o w n , \n X X    3 4 2 1  3\n\0\0
```

Looks like someone sprayed a street address over our counter. Now we know where to look.

file with Perl, you could easily identify where the offending `open` was occurring.

Rubber Ducking

A very simple but particularly useful technique for finding the cause of a problem is simply to explain it to someone else. The other person should look over your shoulder at the screen, and nod his or her head constantly (like a rubber duck bobbing up and down in a bathtub). They do not need to say a word; the simple act of explaining, step by step, what the code is supposed to do often causes the problem to leap off the screen and announce itself.[7]

It sounds simple, but in explaining the problem to another person you must explicitly state things that you may take for granted when going through the code yourself. By having to verbalize some of these assumptions, you may suddenly gain new insight into the problem.

---

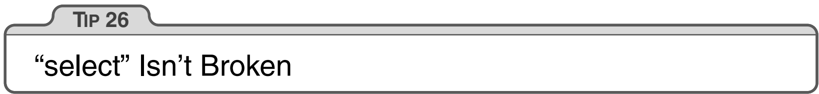
7. Why "rubber ducking"? While an undergraduate at Imperial College in London, Dave did a lot of work with a research assistant named Greg Pugh, one of the best developers Dave has known. For several months Greg carried around a small yellow rubber duck, which he'd place on his terminal while coding. It was a while before Dave had the courage to ask. . . .

Process of Elimination

In most projects, the code you are debugging may be a mixture of application code written by you and others on your project team, third-party products (database, connectivity, graphical libraries, specialized communications or algorithms, and so on) and the platform environment (operating system, system libraries, and compilers).

It is possible that a bug exists in the OS, the compiler, or a third-party product—but this should not be your first thought. It is much more likely that the bug exists in the application code under development. It is generally more profitable to assume that the application code is incorrectly calling into a library than to assume that the library itself is broken. Even if the problem *does* lie with a third party, you'll still have to eliminate your code before submitting the bug report.

We worked on a project where a senior engineer was convinced that the `select` system call was broken on Solaris. No amount of persuasion or logic could change his mind (the fact that every other networking application on the box worked fine was irrelevant). He spent weeks writing work-arounds, which, for some odd reason, didn't seem to fix the problem. When finally forced to sit down and read the documentation on `select`, he discovered the problem and corrected it in a matter of minutes. We now use the phrase "select is broken" as a gentle reminder whenever one of us starts blaming the system for a fault that is likely to be our own.



Remember, if you see hoof prints, think horses—not zebras. The OS is probably not broken. And the database is probably just fine.

If you "changed only one thing" and the system stopped working, that one thing was likely to be responsible, directly or indirectly, no matter how farfetched it seems. Sometimes the thing that changed is outside of your control: new versions of the OS, compiler, database, or other third-party software can wreak havoc with previously correct code. New bugs might show up. Bugs for which you had a work-around get fixed, breaking the work-around. APIs change, functionality changes; in short, it's a whole new ball game, and you must retest the system under these

new conditions. So keep a close eye on the schedule when considering an upgrade; you may want to wait until *after* the next release.

If, however, you have no obvious place to start looking, you can always rely on a good old-fashioned binary search. See if the symptoms are present at either of two far away spots in the code. Then look in the middle. If the problem is present, then the bug lies between the start and the middle point; otherwise, it is between the middle point and the end. You can continue in this fashion until you narrow down the spot sufficiently to identify the problem.

The Element of Surprise

When you find yourself surprised by a bug (perhaps even muttering "that's impossible" under your breath where we can't hear you), you must reevaluate truths you hold dear. In that linked list routine—the one you knew was bulletproof and couldn't possibly be the cause of this bug—did you test *all* the boundary conditions? That other piece of code you've been using for years—it couldn't possibly still have a bug in it. Could it?

Of course it can. The amount of surprise you feel when something goes wrong is directly proportional to the amount of trust and faith you have in the code being run. That's why, when faced with a "surprising" failure, you must realize that one or more of your assumptions is wrong. Don't gloss over a routine or piece of code involved in the bug because you "know" it works. Prove it. Prove it in *this* context, with *this* data, with *these* boundary conditions.

When you come across a surprise bug, beyond merely fixing it, you need to determine why this failure wasn't caught earlier. Consider whether you need to amend the unit or other tests so that they would have caught it.

Also, if the bug is the result of bad data that was propagated through a couple of levels before causing the explosion, see if better parameter checking in those routines would have isolated it earlier (see the

discussions on crashing early and assertions on pages 120 and 122, respectively).

While you're at it, are there any other places in the code that may be susceptible to this same bug? Now is the time to find and fix them. Make sure that *whatever* happened, you'll know if it happens again.

If it took a long time to fix this bug, ask yourself why. Is there anything you can do to make fixing this bug easier the next time around? Perhaps you could build in better testing hooks, or write a log file analyzer.

Finally, if the bug is the result of someone's wrong assumption, discuss the problem with the whole team: if one person misunderstands, then it's possible many people do.

Do all this, and hopefully you won't be surprised next time.

Debugging Checklist

- Is the problem being reported a direct result of the underlying bug, or merely a symptom?
- Is the bug *really* in the compiler? Is it in the OS? Or is it in your code?
- If you explained this problem in detail to a coworker, what would you say?
- If the suspect code passes its unit tests, are the tests complete enough? What happens if you run the unit test with *this* data?
- Do the conditions that caused this bug exist anywhere else in the system?

Related sections include:

- *Assertive Programming*, page 122
- *Programming by Coincidence*, page 172
- *Ubiquitous Automation*, page 230
- *Ruthless Testing*, page 237

Challenges

- Debugging is challenge enough.

19 Text Manipulation

Pragmatic Programmers manipulate text the same way woodworkers shape wood. In previous sections we discussed some specific tools—shells, editors, debuggers—that we use. These are similar to a woodworker's chisels, saws, and planes—tools specialized to do one or two jobs well. However, every now and then we need to perform some transformation not readily handled by the basic tool set. We need a general-purpose text manipulation tool.

Text manipulation languages are to programming what routers[8] are to woodworking. They are noisy, messy, and somewhat brute force. Make mistakes with them, and entire pieces can be ruined. Some people swear they have no place in the toolbox. But in the right hands, both routers and text manipulation languages can be incredibly powerful and versatile. You can quickly trim something into shape, make joints, and carve. Used properly, these tools have surprising finesse and subtlety. But they take time to master.

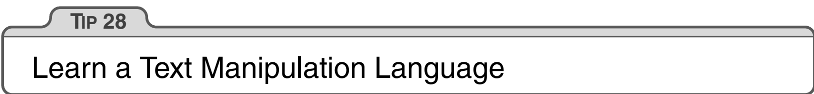
There is a growing number of good text manipulation languages. Unix developers often like to use the power of their command shells, augmented with tools such as `awk` and `sed`. People who prefer a more structured tool like the object-oriented nature of Python [URL 9]. Some people use Tcl [URL 23] as their tool of choice. We happen to prefer Ruby [TFH04] and Perl [URL 8] for hacking out short scripts.

These languages are important enabling technologies. Using them, you can quickly hack up utilities and prototype ideas—jobs that might take five or ten times as long using conventional languages. And that multiplying factor is crucially important to the kind of experimenting that we do. Spending 30 minutes trying out a crazy idea is a whole lot better than spending five hours. Spending a day automating important components of a project is acceptable; spending a week might not be. In their book *The Practice of Programming* [KP99], Kernighan and Pike built the same program in five different languages. The Perl version was the shortest (17 lines, compared with C's 150). With Perl you can

---

8. Here *router* means the tool that spins cutting blades very, very fast, not a device for interconnecting networks.

manipulate text, interact with programs, talk over networks, drive Web pages, perform arbitrary precision arithmetic, and write programs that look like Snoopy swearing.



To show the wide-ranging applicability of text manipulation languages, here's a sample of some applications we've developed over the last few years.

- **Database schema maintenance.** A set of Perl scripts took a plain text file containing a database schema definition and from it generated:
  - The SQL statements to create the database
  - Flat data files to populate a data dictionary
  - C code libraries to access the database
  - Scripts to check database integrity
  - Web pages containing schema descriptions and diagrams
  - An XML version of the schema
- **Java property access.** It is good OO programming style to restrict access to an object's properties, forcing external classes to get and set them via methods. However, in the common case where a property is represented inside the class by a simple member variable, creating a `get` and `set` method for each variable is tedious and mechanical. We have a Perl script that modifies the source files and inserts the correct method definitions for all appropriately flagged variables.
- **Test data generation.** We had tens of thousands of records of test data, spread over several different files and formats, that needed to be knitted together and converted into a form suitable for loading into a relational database. Perl did it in a couple of hours (and in the process found a couple of consistency errors in the original data).
- **Book writing.** We think it is important that any code presented in a book should have been tested first. Most of the code in this

book has been. However, using the *DRY* principle (see *The Evils of Duplication*, page 26) we didn't want to copy and paste lines of code from the tested programs into the book. That would have meant that the code was duplicated, virtually guaranteeing that we'd forget to update an example when the corresponding program was changed. For some examples, we also didn't want to bore you with all the framework code needed to make our example compile and run. We turned to Perl. A relatively simple script is invoked when we format the book—it extracts a named segment of a source file, does syntax highlighting, and converts the result into the typesetting language we use.

- **C to Object Pascal interface.** A client had a team of developers writing Object Pascal on PCs. Their code needed to interface to a body of code written in C. We developed a short Perl script that parsed the C header files, extracting the definitions of all exported functions and the data structures they used. We then generated Object Pascal units with Pascal records for all the C structures, and imported procedure definitions for all the C functions. This generation process became part of the build, so that whenever the C header changed, a new Object Pascal unit would be constructed automatically.

- **Generating Web documentation.** Many project teams are publishing their documentation to internal Web sites. We have written many Perl programs that analyze database schemas, C or C++ source files, makefiles, and other project sources to produce the required HTML documentation. We also use Perl to wrap the documents with standard headers and footers, and to transfer them to the Web site.

We use text manipulation languages almost every day. Many of the ideas in this book can be implemented more simply in them than in any other language of which we're aware. These languages make it easy to write code generators, which we'll look at next.

Related sections include:

- *The Evils of Duplication*, page 26

Exercises

Answer on p. 285

11. Your C program uses an enumerated type to represent one of 100 states. You'd like to be able to print out the state as a string (as opposed to a number) for debugging purposes. Write a script that reads from standard input a file containing

```
name
state_a
state_b
 :    :
```

Produce the file *name.h*, which contains

```
extern const char* NAME_names[];
typedef enum {
   state_a,
   state_b,
    :    :
 } NAME;
```

and the file *name.c*, which contains

```
const char* NAME_names[] = {
   "state_a",
   "state_b",
     :    :
 };
```

Answer on p. 286

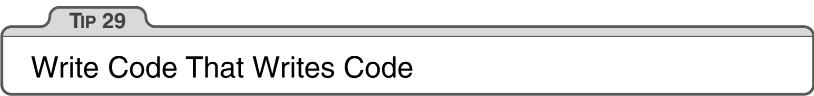
12. Halfway through writing this book, we realized that we hadn't put the `use strict` directive into many of our Perl examples. Write a script that goes through the `.pl` files in a directory and adds a `use strict` at the end of the initial comment block to all files that don't already have one. Remember to keep a backup of all files you change.

Code Generators

When woodworkers are faced with the task of producing the same thing over and over, they cheat. They build themselves a jig or a template. If they get the jig right once, they can reproduce a piece of work time after time. The jig takes away complexity and reduces the chances of making mistakes, leaving the craftsman free to concentrate on quality.

As programmers, we often find ourselves in a similar position. We need to achieve the same functionality, but in different contexts. We need to repeat information in different places. Sometimes we just need to protect ourselves from carpal tunnel syndrome by cutting down on repetitive typing.

In the same way a woodworker invests the time in a jig, a programmer can build a code generator. Once built, it can be used throughout the life of the project at virtually no cost.



There are two main types of code generators:

1. *Passive code generators* are run once to produce a result. From that point forward, the result becomes freestanding—it is divorced from the code generator. The wizards discussed in *Evil Wizards*, page 198, along with some CASE tools, are examples of passive code generators.

2. *Active code generators* are used each time their results are required. The result is a throw-away—it can always be reproduced by the code generator. Often, active code generators read some form of script or control file to produce their results.

Passive Code Generators

Passive code generators save typing. They are basically parameterized templates, generating a given output from a set of inputs. Once the result is produced, it becomes a full-fledged source file in the project; it will be edited, compiled, and placed under source control just like any other file. Its origins will be forgotten.

Passive code generators have many uses:

- *Creating new source files.* A passive code generator can produce templates, source code control directives, copyright notices, and standard comment blocks for each new file in a project. We have our editors set up to do this whenever we create a new file: edit a new Java program, and the new editor buffer will automatically contain a comment block, package directive, and the outline class declaration, already filled in.

- *Performing one-off conversions* among programming languages. We started writing this book using the troff system, but we switched to LaTeX after 15 sections had been completed. We wrote a code generator that read the troff source and converted it to LaTeX. It was

about 90% accurate; the rest we did by hand. This is an interesting feature of passive code generators: they don't have to be totally accurate. You get to choose how much effort you put into the generator, compared with the energy you spend fixing up its output.

- *Producing lookup tables and other resources* that are expensive to compute at runtime. Instead of calculating trigonometric functions, many early graphics systems used precomputed tables of sine and cosine values. Typically, these tables were produced by a passive code generator and then copied into the source.

Active Code Generators

While passive code generators are simply a convenience, their active cousins are a necessity if you want to follow the *DRY* principle. With an active code generator, you can take a single representation of some piece of knowledge and convert it into all the forms your application needs. This *is not* duplication, because the derived forms are disposable, and are generated as needed by the code generator (hence the word *active*).

Whenever you find yourself trying to get two disparate environments to work together, you should consider using active code generators.

Perhaps you're developing a database application. Here, you're dealing with two environments—the database and the programming language you are using to access it. You have a schema, and you need to define low-level structures mirroring the layout of certain database tables. You could just code these directly, but this violates the *DRY* principle: knowledge of the schema would then be expressed in two places. When the schema changes, you need to remember to change the corresponding code. If a column is removed from a table, but the code base is not changed, you might not even get a compilation error. The first you'll know about it is when your tests start failing (or when the user calls).

An alternative is to use an active code generator—take the schema and use it to generate the source code for the structures, as shown in Figure 3.3. Now, whenever the schema changes, the code used to access it also changes, automatically. If a column is removed, then its corresponding field in the structure will disappear, and any higher-level code that uses that column will fail to compile. You've caught the error at compile time,

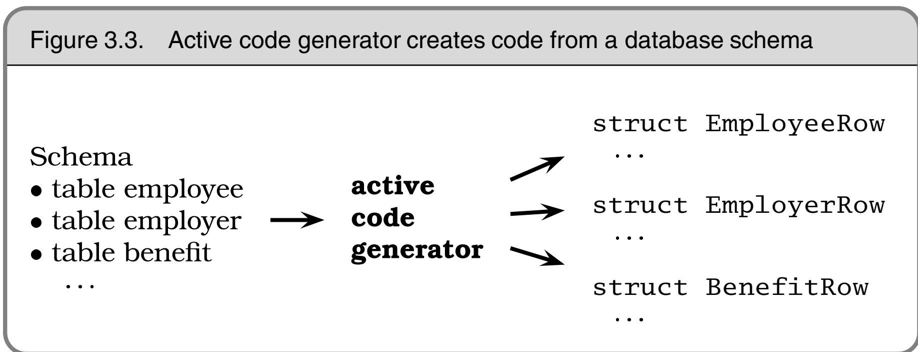


Figure 3.3. Active code generator creates code from a database schema

not in production. Of course, this scheme works only if you make the code generation part of the build process itself.[9]

Another example of melding environments using code generators happens when different programming languages are used in the same application. In order to communicate, each code base will need some information in common—data structures, message formats, and field names, for example. Rather than duplicate this information, use a code generator. Sometimes you can parse the information out of the source files of one language and use it to generate code in a second language. Often, though, it is simpler to express it in a simpler, language-neutral representation and generate the code for both languages, as shown in Figure 3.4 on the following page. Also see the answer to Exercise 13 on page 286 for an example of how to separate the parsing of the flat file representation from code generation.

Code Generators Needn't Be Complex

All this talk of *active* this and *passive* that may leave you with the impression that code generators are complex beasts. They needn't be. Normally the most complex part is the parser, which analyzes the input file. Keep the input format simple, and the code generator becomes

---

9. Just *how* do you go about building code from a database schema? There are several ways. If the schema is held in a flat file (for example, as `create table` statements), then a relatively simple script can parse it and generate the source. Alternatively, if you use a tool to create the schema directly in the database, then you should be able to extract the information you need directly from the database's data dictionary. Perl provides libraries that give you access to most major databases.

Figure 3.4. Generating code from a language-neutral representation. In the input file, lines starting with 'M' flag the start of a message definition, 'F' lines define fields, and 'E' is the end of the message.

```
# Add a product
# to the 'on-order' list
M AddProduct
F id         int
F name       char[30]
F order_code int
E
```



```
/* Add a product */
/* to the 'on-order' list */
typedef struct {
  int   id;
  char  name[30];
  int   order_code;
} AddProductMsg;
```

```
{ Add a product }
{ to the 'on-order' list }
AddProductMsg = packed record
  id:         LongInt;
  name:       array[0..29] of char;
  order_code: LongInt;
end;
```

simple. Have a look at the answer to Exercise 13 (page 286): the actual code generation is basically `print` statements.

Code Generators Needn't Generate Code

Although many of the examples in this section show code generators that produce program source, this needn't always be the case. You can use code generators to write just about any output: HTML, XML, plain text—any text that might be an input somewhere else in your project.

Related sections include:

- *The Evils of Duplication*, page 26
- *The Power of Plain Text*, page 73
- *Evil Wizards*, page 198
- *Ubiquitous Automation*, page 230

Exercises

Answer on p. 286

13. Write a code generator that takes the input file in Figure 3.4, and generates output in two languages of your choice. Try to make it easy to add new languages.

Index

A

Accessor function, 31
ACM, *see* Association for Computing Machinery
Active code generator, 104
Activity diagram, 150
Advanced C++ Programming Styles and Idioms, 265
Advanced Programming in the Unix Environment, 264
Aegis transaction-based configuration management, 246, 271
Agent, 76, 117, 297
Algorithm
  binary chop, 180
  choosing, 182
  combinatoric, 180
  divide-and-conquer, 180
  estimating, 177, 178
  linear, 177
  O() notation, 178, 181
  quicksort, 180
  runtime, 181
  sublinear, 177
Allocations, nesting, 131
Analysis Patterns, 264
Anonymity, 258
AOP, *see* Aspect-Oriented Programming
Architecture
  deployment, 156
  flexibility, 46
  prototyping, 55
  temporal decoupling, 152
Art of Computer Programming, 183
Artificial intelligence, marauding, 26
Aspect-Oriented Programming (AOP), 39, 273
Assertion, 113, 122, 175
  side effects, 124
  turning off, 123
Association for Computing Machinery (ACM), 262
  Communications of the ACM, 263
  SIGPLAN, 263
Assumptions, testing, 175
"`at`" command, 231
Audience, 21
  needs, 19
`auto_ptr`, 134
Automation, 230
  approval procedures, 235
  build, 88, 233
  compiling, 232
  `cron`, 231
  documentation, 251
  scripts, 234
  team, 229
  testing, 29, 238
  Web site generation, 235
`awk`, 99

B

Backus-Naur Form (BNF), 59n
Base class, 112
`bash` shell, 80, 82n
Bean, *see* Enterprise Java Beans (EJB)
Beck, Kent, 194, 258
Beowulf project, 268
"Big *O*" notation, 177
"Big picture", 8
Binary chop, 97, 180
Binary format, 73
  problems parsing, 75
`bison`, 59, 269
BIST, *see* Built-In Self Test
Blackboard system, 165
  partitioning, 168
  workflow, 169

Blender example
- contract for, 119, 289
- regression test jig, 305
- workflow, 151

BNF, *see* Backus-Naur Form (BNF)
Boiled frog, 8, 175, 225
Boundary condition, 173, 243
Brain, Marshall, 265
Branding, 226
Brant, John, 268
"Broken Window Theory", 5
- *vs.* stone soup, 9

Brooks, Fred, 264
Browser, class, 187
Browser, refactoring, 187, 268
Bug, 90
- failed contract as, 111
- *see also* Debugging; Error

Build
- automation, 88, 233
- dependencies, 233
- final, 234
- nightly, 231
- refactoring, 187

Built-In Self Test (BIST), 189
Business logic, 146
Business policy, 203

C

C language
- assertions, 122
- DBC, 114
- duplication, 29
- error handling, 121
- error messages, 115
- macros, 121
- Object Pascal interface, 101

C++ language, 46
- assertions, 122
- `auto_ptr`, 134
- books, 265
- DBC, 114
- decoupling, 142
- DOC++, 251, 269
- duplication, 29
- error messages, 115
- exceptions, 132
- unit tests, 193

Caching, 31
Call, routine, 115, 173
Cascading Style Sheets (CSS), 253
Cat
- blaming, 3
- herding, 224
- Schrödinger's, 47

Catalyzing change, 8
Cathedrals, xx
Cetus links, 265
Change, catalyzing, 8
Christiansen, Tom, 81
Class
- assertions, 113
- base, 112
- coupling, 139, 142
- coupling ratios, 242
- encapsulating resource, 132
- invariant, 110, 113
- number of states, 245
- resource allocation, 132
- subclass, 112
- wrapper, 132, 133, 135, 141

Class browser, 187
ClearCase, 271
Cockburn, Alistair, xxiii, 205, 264, 272
Code generator, 28, 102
- active, 104
- makefiles, 232
- parsers, 105
- passive, 103

Code profiler, 182
Code reviews, 33, 236
Coding
- algorithm speed, 177
- comments, 29, 249
- coupled, 130
- coverage analysis, 245
- database schema, 104
- defensive, 107
- and documentation, 29, 248
- estimating, 68
- exceptions, 125
- implementation, 173
- iterative, 69
- "lazy", 111
- metrics, 242
- modules, 138
- multiple representations, 28
- orthogonality, 34, 36, 40

ownership, 258
prototypes, 55
server code, 196
"shy", 40, 138
specifications, 219
tracer bullets, 49–51
unit testing, 190, 192
see also Coupled code; Decoupled code; Metadata; Source code control system (SCCS)
Cohesion, 35
COM, *see* Component Object Model
Combinatorial explosion, 140, 167
Combinatoric algorithm, 180
Command shell, 77
`bash`, 80
Cygwin, 80
vs. GUI, 78
UWIN, 81
Windows, 80
Comment, 29, 249
avoiding duplication, 29
DBC, 113
parameters, 250
types of, 249
unnecessary, 250
see also Documentation
Common Object Request Broker (CORBA), 29, 39, 46
Event Service, 160
Communicating, 18
audience, 19, 21
duplication, 32
e-mail, 22
and formal methods, 221
presentation, 20
style, 20
teams, 225
users, 256
writing, 18
Communications of the ACM, 263
Comp.object FAQ, 272
Compiling, 232
compilers, 267
DBC, 113
warnings and debugging, 92
Component Object Model (COM), 55
Component-based systems, *see* Modular system
Concurrency, 150
design, 154
interfaces, 155
and Programming by Coincidence, 154
requirements analysis of, 150
workflow, 150
Concurrent Version System (CVS), 271
Configuration
cooperative, 148
dynamic, 144
metadata, 147
Configuration management, 86, 271
Constantine, Larry L., 35
Constraint management, 213
Constructor, 132
initialization, 155
Contact, authors' e-mail, xxiii
Context, use instead of globals, 40
Contract, 109, 174
see also Design by contract (DBC)
Controller (MVC), 162
Coplien, Jim, 265
CORBA, *see* Common Object Request Broker
Coupled code, 130
coupling ratios, 242
minimizing, 138, 158
performance, 142
temporal coupling, 150
see also Decoupled code
Coverage analysis, 245
Cox, Brad J., 189n
Crash, 120
Critical thinking, 16
`cron`, 231
CSS, *see* Cascading Style Sheets
CVS, *see* Concurrent Version System
Cygwin, 80, 270

D

Data
blackboard system, 169
caching, 31
dictionary, 144
dynamic data structures, 135
global, 40
language, 60
normalizing, 30

readable *vs.* understandable, 75
test, 100, 243
views, 160
visualizing, 93
see also Metadata
Data Display Debugger (DDD), 93, 268
Database
active code generator, 104
schema, 105f, 141, 144
schema maintenance, 100
DBC, *see* Design by contract
DDD, *see* Data Display Debugger
Deadline, 6, 246
Deadlock, 131
Debugging, 90
assertions, 123
binary search, 97
bug location, 96
bug reproduction, 93
checklist, 98
compiler warnings and, 92
corrupt variables, 95
"Heisenbug", 124
rubber ducking, 95
and source code branching, 87
surprise bug, 97
and testing, 92, 195
time bomb, 192
tracing, 94
view, 164
visualizing data, 93
Decision making, 46
Decoupled code, 38, 40
architecture, 152
blackboard system, 166
Law of Demeter, 140
metadata, 145
minimizing coupling, 138
modular testing, 244
physical decoupling, 142
temporal coupling, 150
workflow, 150
see also Coupled code
Defensive coding, 107
Delegation, 304
Delphi, 55
see also Object Pascal
Demeter project, 274
Demeter, Law of, 140
Dependency, reducing, *see* Modular system; Orthogonality
Deployment, 156
Deployment descriptor, 148
Design
accessor functions, 31
concurrency, 154
context, 174
deployment, 156
design/methodology testing, 242
metadata, 145
orthogonality, 34, 37
physical, 142
refactoring, 186
using services, 154
Design by contract (DBC), 109, 155
and agents, 117
assertions, 113
class invariant, 110
as comments, 113
dynamic contracts, 117
iContract, 268
language support, 114
list insertion example, 110
pre- and postcondition, 110, 113, 114
predicates, 110
unit testing, 190
Design Patterns, 264
observer, 158
singleton, 41
strategy, 41
Destructor, 132
Detectives, 165
Development tree, 87
Development, iterative, 69
Divide-and-conquer algorithm, 180
DOC++ documentation generator, 251, 269
DocBook, 254
Documentation
automatic updating, 251
and code, 29, 248
comments, 29, 113, 249, 251
executable, 251
formats, 253
HTML, 101
hypertext, 210
internal/external, 248

invariant, 117
mark-up languages, 254
orthogonality, 42
outline, 18
requirements, 204
technical writers, 252
word processors, 252, 254
writing specifications, 218
see also Comment; Web documentation
Dodo, 148
Domain, problem, 58, 66
Don't repeat yourself, *see* DRY principle
Downloading source code, *see* Example code
Dr. Dobbs Journal, 263
DRY principle, 27, 29, 42
see also Duplication
Duck, rubber, *see* Rubber duck
Dumpty, Humpty, xxii, 165
Duplication, 26
code generators avoid, 28
and code reviews, 33
design errors, 30
documentation and code, 29
DRY principle, 27, 29
interdeveloper, 32
in languages, 29
multiple representations, 28
teams, 226
under time pressure, 32
types of, 27
Dynamic configuration, 144
Dynamic data structure, 135
Dynamics of Software Development, 264

E

E-mail, 22
address for feedback, xxiii
Editor, 82
auto-indenting, 85
cursor movement, 84
features, 83
generating code, 103
how many to learn, 82
template, 84
types of, 266
Windows notepad, 84
Effective C++, 265
Eiffel, 109, 114, 267
EJB, *see* Enterprise Java Beans
`elvis` editor, 267
Emacs editor, 84, 266
`Viper vi` emulator, 267
Embedded mini-language, 62, 145
Embellishment, 11
Encapsulation, object, 127, 158
Eno, Brian, 205
Enterprise Java Beans (EJB), 39, 147
Entropy, 4
Error
DBC messages, 115
design, 30
domain-specific, 59
early crash, 120
log messages, 196
orthogonality, 41
testing, 240, 247
see also Exception
Error handler, 127
Estimating, 64
accuracy, 64
algorithms, 177, 178
iterative, 69
models, 66
problem domain, 66
project schedules, 68
records, 68
testing, 182
Eton College, xxi
Event, 157
Event channel, 160
Example code
add logging, 40
airline reservations, 164, 296
`assert` macro, 122
`auto_ptr` example, 134
bad resource balancing, 129, 130
downloading, xxiii
exception error handling, 125
good resource balancing, 131
JavaDoc example, 250
method chaining, 139
normalized class, 31
open password file, 126
open user file, 127

resources and exceptions, 132, 133
side effect, 124
spaghetti error handling, 125
square root, 190
string parsing with `StringTokenizer`, 156
string parsing with `strtok`, 155
unnormalized class, 30

Example code by name
- `AOP`, 40
- `Misc.java`, 156
- `assert`, 122
- `bad_balance.c`, 129, 130
- `balance.cc`, 134
- `balance.c`, 131–133
- `class Line`, 30, 31
- `exception`, 125
- `findPeak`, 250
- `interface Flight`, 164, 296
- `misc.c`, 155
- `openpasswd.java`, 126
- `openuserfile.java`, 127
- `plotDate`, 139
- `side_effect`, 124
- `spaghetti`, 125
- `sqrt`, 190

Exception, 121
- effects of, 127
- and error handlers, 127
- missing files, 126
- resource balancing, 132
- when to use, 125

Excuses, 3
Executable document, 251
`expect`, 269
Expert, *see* Guru
Expiring asset, 12
eXtensible Style Language (XSL), 253
Extinction, 148
eXtreme Programming, 238n, 258, 272

F

Feature creep, 10
Feedback, e-mail address, xxiii
File
- exception, 126
- header, 29
- implementation, 29
- log, 196
- makefile, 232
- source, 103

Final build, 234
Fish, dangers of, 34
Flexibility, 46
Formal methods, 220, 221
Four Posts Puzzle, 213
Fowler, Martin, xxiii, 186, 273
Free Software Foundation, *see* GNU Project
Frog, boiled, *see* Boiled frog
Function
- accessor, 31
- Law of Demeter for ∼s, 140
- similar, 41

G

Gamma, Erich, 194
Garbage collection, 134
Gardening metaphor, 184
Gehrke, Peter, xxiv
Glass, Robert, 221, 236
Global variables, 40, 130, 154
Glossary, project, 210
GNU Project, 274
- C/C++ compiler, 267
- General Public License (GPL), 80
- GNU Image Manipulation Program (GIMP), 274
- SmallEiffel, 267

"Good-enough software", *see* Software, quality
Gordian knot, 212
Goto, 127
GUI system
- *vs.* command shell, 78
- interface, 78
- testing, 244

Guru, 17, 198

H

Hash, secure, 74
Header file, 29
"Heisenbug", 124, 289
Helicopter, 34n
Hopper, Grace, 8n, 90
"Hot-key" sequence, 196

HTTP Web server, 196
Human factors, 241
Humpty Dumpty, xxii, 165
Hungarian notation, 249
Hungry consumer model, 153
Hypertext document, 210

I

iContract, 110, 114, 268
IDE, *see* Integrated Development Environment
IEEE Computer Society, 262
    IEEE Computer, 262
    IEEE Software, 263
Imperative language, 60
Implementation
    accidents, 173
    coding, 173
    specifications, 219
Imposed duplication, 28
Inadvertent duplication, 30
Indentation, automatic, 85
Independence, *see* Orthogonality
Infrastructure, 37
Inheritance, 111
    assertions, 113
    fan-in/fan-out, 242
Inner tennis, 215
Inspection, code, *see* Code reviews
Insure++, 136
Integrated circuit, 189n
Integrated Development Environment (IDE), 72, 232
Integration platform, 50
Integration testing, 239
Interface
    blackboard system, 168
    C/Object Pascal, 101
    concurrency, 155
    error handler, 128
    GUI, 78
    prototyping, 55
    user, 203
Invariant, 110, 113, 155
    loop, 116
    semantic, 116, 135
ISO9660 format, 233n
Iterative development, 69

J

Jacobson, Ivar, 204
Jargon, xxii, 210
Jargon file, 273
Java, 46, 267
    code generation, 232
    DBC, 114
    Enterprise Java Beans, 39, 147
    error messages, 115
    exceptions, 121
    iContract, 110, 114, 268
    `javaCC`, 59, 269
    JavaDoc, 248, 251
    JavaSpaces, 166, 273
    JUnit, 195
    multithreaded programming, 154
    property access, 100
    property files, 145
    resource balancing, 134
    RMI, 128
    string parser, 156
    tree view, 161
    unit tests, 193
    and Windows shells, 81
JavaDoc, *see* Java

K

K Desktop Environment, 273
Kaizen, xxi, 14
    see also Knowledge portfolio
Kernighan, Brian, 99
Keybinding, 82
Kirk, James T., 26
Knowledge
    producers and consumers, 166
Knowledge portfolio, 12
    building, 13
    critical thinking, 16
    learning and reading, 14
    researching, 15
Knuth, Donald, 183, 248
Korn, David, 81
Kramer, Reto, xxiv
Kruchten, Phillipe, 227n

L

Lakos, John, xxiv, 9, 142, 265
Lame excuses, 3

Language, programming
- conversions, 103, 105
- DBC, 114
- domain, 57
- duplication in, 29
- learning, 14
- prototypes, 55
- scripting, 55, 145
- specification, 58, 62
- text manipulation, 99
- *see also* Mini-language

Large-Scale C++ Software Design, 142, 265
LaTeX system, 103
Law of Demeter, 140
Lawns, care of, xxi
Layered design, 37
Layered system, *see* Modular system
"lazy" code, 111
Lex and Yacc, 59
Librarian, *see* Project librarian
Library code, 39
Linda model, 167
Linear algorithms, 177
Linux, 15, 254, 265
Liskov Substitution Principle, 111
Listening, 21
Literate programming, 248
Logging, 39, 196
- *see also* Tracing

Lookup table, 104
Loop
- nested, 180
- simple, 180

Loop invariant, 116

M

Macro, 78, 86
- assertions, 122
- documentation, 252
- error handling, 121

Maintenance, 26
- imperative languages, 61

Makefile, 232
- recursive, 233

Managing expectations, 256
Mark-up language, 254
Martin, Robert C., 273
McCabe Cyclomatic Complexity Metric, 242
Member variables, *see* Accessor functions
Memory allocation, 135
Metadata, 144, 203
- business logic, 146
- configuration, 147
- controlling transactions, 39
- decoupled code, 145
- and formal methods, 221
- in plain text, 74

Metric, 242
Meyer, Bertrand, 31n, 109, 184, 264
Meyer, Scott, 265
Microsoft Visual C++, 198
Microsoft Windows, 46
Mini-language, 59
- data language, 60
- embedded, 62
- imperative, 60
- parsing, 62
- stand-alone, 62

Mixing board, 205
MKS Source Integrity, 271
Model, 160
- calculations, 67
- components and parameters, 66
- and estimating, 66
- executable documents, 251
- view, 162

Model-view-controller (MVC), 38, 160
Modular system, 37
- coding, 138
- prototyping, 55
- resource allocation, 135
- reversibility, 45
- testing, 41, 190, 244

More Effective C++, 265
Mozilla, 273
Multithreaded programming, 154
MVC, *see* Model-view-controller
The Mythical Man Month, 264

N

Name, variable, 249
`Nana`, 114, 268
Nest allocations, 131
Nested loop, 180

Netscape, 145, 273
Newsgroup, 15, 17, 33
Nonorthogonal system, 34
Normalize, 30
Novobilski, Andrew J., 189n

O

O() notation, 178, 181
Object
  coupling, 140n
  destruction, 133, 134
  persistence, 39
  publish/subscribe protocol, 158
  singleton, 41
  valid/invalid state, 154
  viewer, 163
Object Management Group (OMG), 270
Object Pascal, 29
  C interface, 101
Object-Oriented Programming, 189n
Object-Oriented Software Construction, 264
Obsolescence, 74
OLTP, *see* On-Line Transaction Processing system
OMG, *see* Object Management Group
On-Line Transaction Processing system (OLTP), 152
Options, providing, 3
Ordering, *see* Workflow
Orthogonality, 34
  coding, 34, 36, 40
  design, 37
  documentation, 42
  DRY principle, 42
  nonorthogonal system, 34
  productivity, 35
  project teams, 36, 227
  testing, 41
  toolkits & libraries, 39
  see also Modular system
Over embellishment, 11

P

Pain management, 185
`paint()` method, 173
Painting, 11
Papua New Guinea, 16
Parallel programming, 150
Parrots, killer, *see* Branding
Parsing, 59
  code generators, 105
  log messages, 196
  mini-language, 62
  strings, 155
Partitioning, 168
Pascal, 29
Passive code generator, 103
Performance testing, 241
Perl, 55, 62, 99
  C/Object Pascal interface, 101
  database schema generation, 100
  home page, 267
  Java property access, 100
  power tools, 270
  test data generation, 100
  testing, 197
  and typesetting, 100
  Unix utilities in, 81
  web documentation, 101
Perl Journal, 263
Persistence, 39, 45
Petzold, Charles, 265
Pike, Rob, 99
Pilot
  landing, handling, etc., 217
  who ate fish, 34
Plain text, 73
  vs. binary format, 73
  drawbacks, 74
  executable documents, 251
  leverage, 75
  obsolescence, 74
  and easier testing, 76
  Unix, 76
Polymorphism, 111
Post-it note, 53, 55
Powerbuilder, 55
The Practice of Programming, 99
Pragmatic programmer
  characteristics, xviii
  e-mail address, xxiii
  Web site, xxiii
Pre- and postcondition, 110, 113, 114
Predicate logic, 110
Preprocessor, 114
Presentation, 20

Problem domain, 58, 66
  metadata, 146
Problem solving, 213
  checklist for, 214
Productivity, 10, 35
Programming by coincidence, 173
Programming staff
  expense of, 237
Programming Windows, 265
Project
  glossary, 210
  "heads", 228
  saboteur, 244
  schedules, 68
  see also Automation; Team, project
Project librarian, 33, 226
Prototyping, 53, 216
  architecture, 55
  disposable code, 56
  kinds of, 54
  and programming languages, 55
  and tracer code, 51
  using, 54
Publish/subscribe protocol, 158
Pugh, Greg, 95n
Purify, 136
PVCS Configuration Management, 271
Python, 55, 99, 267

Q

Quality
  control, 9
  requirements, 11
  teams, 225
Quarry worker's creed, xx
Quicksort algorithm, 180

R

Rational Unified Process, 227n
Raymond, Eric S., 273
RCS, *see* Revision Control System
Real-world data, 243
Refactoring, 5, 185
  automatic, 187
  and design, 186
  testing, 187
  time constraints, 185
Refactoring browser, 187, 268
Refinement, excessive, 11
Regression, 76, 197, 232, 242
Relationship
  has-a, 304
  kind-of, 111, 304
Releases, and SCCS, 87
Remote Method Invocation (RMI), 128
  exception handling, 39
Remote procedure call (RPC), 29, 39
Repository, 87
Requirement, 11, 202
  business problem, 203
  changing, 26
  creep, 209
  DBC, 110
  distribution, 211
  documenting, 204
  in domain language, 58
  expressing as invariant, 116
  formal methods, 220
  glossary, 210
  over specifying, 208
  and policy, 203
  usability testing, 241
  user interface, 203
Researching, 15
Resource balancing, 129
  C++ exceptions, 132
  checking, 135
  coupled code, 130
  dynamic data structures, 135
  encapsulation in class, 132
  Java, 134
  nest allocations, 131
Response set, 141, 242
Responsibility, 2, 250, 258
Reuse, 33, 36
Reversibility, 44
  flexible architecture, 46
Revision Control System (RCS), 250, 271
Risk management, 13
  orthogonality, 36
RMI, *see* Remote Method Invocation
Rock-n-roll, 47
RPC, *see* Remote procedure call
Rubber ducking, 3, 95
Rules engine, 169

S

Saboteur, 244
Samba, 272
Sample programs, *see* Example code
Sather, 114, 268
SCCS, *see* Source code control system
Schedule, project, 68
Schrödinger, Erwin (and his cat), 47
Scope, requirement, 209
Screen scraping, 61
Scripting language, 55, 145
Secure hash, 74
`sed`, 99
Sedgewick, Robert, 183
Self-contained components, *see* Orthogonality; Cohesion
Semantic invariant, 116, 135
`sendmail` program, 60
Sequence diagram, 158
Server code, 196
Services, design using, 154
Shell, command, 77
  vs. GUI, 78
  see also Command shell
"Shy code", 40
Side effect, 124
SIGPLAN, 263
Simple loop, 180
Singleton object, 41
Slashdot, 265
SmallEiffel, 267
Smalltalk, 46, 186, 187, 268, 272
Software
  development technologies, 221
  quality, 9
  requirements, 11
Software bus, 159
"Software Construction", 184
Software Development Magazine, 263
Software IC, 189n
"Software rot", 4
Solaris, 76
Source code
  cat eating, 3
  documentation, *see* Comments
  downloading, *see* Example code
  duplication in, 29
  generating, 103
  reviews, *see* Code reviews
Source code control system (SCCS), 86
  Aegis, 246
  builds using, 88
  CVS, 271
  development tree, 87
  plain text and, 76
  RCS, 250, 271
  repository, 87
  tools, 271
Specialization, 221
Specification, 58
  implementation, 219
  language, 62
  as security blanket, 219
  writing, 218
Spy cells, 138
Squeak, 268
Stand-alone mini-language, 62
"Start-up fatigue", 7
Starting a project
  problem solving, 212
  prototyping, 216
  specifications, 217
  see also Requirement
Stevens, W. Richard, 264
Stone soup, 7
  vs. broken windows, 9
Stone-cutter's creed, xx
String parser, 155
Stroop effect, 249
`strtok` routine, 155
Structured walkthroughs, *see* Code reviews
Style sheet, 20, 254
Style, communication, 20
Subclass, 112
Sublinear algorithm, 177
Supplier, *see* Vendor
Surviving Object-Oriented Projects: A Manager's Guide, 264
SWIG, 55, 270
Synchronization bar, 151
Syntax highlighting, 84
Synthetic data, 243

T

T Spaces, 166, 269
TAM, *see* Test Access Mechanism
Tcl, 55, 99, 269

Team, project, 36, 224
- automation, 229
- avoiding duplication, 32
- code review, 236
- communication, 225
- duplication, 226
- functionality, 227
- organization, 227
- pragmatism in, xx
- quality, 225
- tool builders, 229

Technical writer, 252
Template, use case, 205
Temporal coupling, 150
Test Access Mechanism (TAM), 189
Test harness, 194
Testing
- automated, 238
  - from specification, 29
- bug fixing, 247
- coverage analysis, 245
- and culture, 197
- debugging, 92, 196
- design/methodology, 242
- effectiveness, 244
- estimates, 182
- frequency, 246
- GUI systems, 244
- integration, 239
- orthogonality, 36, 41
- performance, 241
- role of plain text, 76
- refactoring, 187
- regression, 76, 197, 232, 242
- resource exhaustion, 240
- saboteur, 244
- test data, 100, 243
- usability, 241
- validation and verification, 239
- *see also* Unit testing

Text manipulation language, 99
TOM programming language, 268
Toolkits, 39
Tools, adaptable, 205
Tracer code, 49
- advantages of, 50
- and prototyping, 51

Tracing, 94
- *see also* Logging

Trade paper, 263
Trade-offs, 249
Transactions, EJB, 39
Tree widget, 161
troff system, 103
Tuple space, 167

U

UML, *see* Unified modeling language (UML)
UNDO key, 86
Unified modeling language (UML)
- activity diagram, 150
- sequence diagram, 158
- use case diagram, 208

Uniform Access Principle, 31n
Unit testing, 190
- DBC, 190
- modules, 239
- test harness, 194
- test window, 196
- writing tests, 193

Unix, 46, 76
- Application Default files, 145
- books, 264
- Cygwin, 270
- DOS tools, 270
- Samba, 272
- UWIN, 81, 270

Unix Network Programming, 264
Usability testing, 241
Use case, 204
- diagrams, 206

Usenet newsgroup, 15, 17, 33
User
- expectations, 256
- groups, 18
- interface, 203
- requirements, 10

UWIN, 81, 270

V

Variable
- corrupt, 95
- global, 130, 154
- name, 249

Vendor
- libraries, 39
- reducing reliance on, 36, 39, 46

vi editor, 266
View
  debugging, 164
  executable documents, 251
  Java tree view, 161
  model-view-controller, 160, 162
  model-viewer network, 162
vim editor, 266
Visual Basic, 55
Visual C++, 198
Visual SourceSafe, 271
VisualWorks, 268

W

Walkthoughs, *see* Code reviews
Warnings, compilation, 92
Web documentation, 101, 210, 253
  automatic generation, 235
  news and information, 265
Web server, 196
Web site, pragmatic programmer, xxiii
What You See Is What You Get (WYSIWYG), 78
WikiWikiWeb, 265
Win32 System Services, 265
Windows, 46
  "at" command, 231
  books, 265
  Cygwin, 80
  metadata, 145
  notepad, 84
  Unix utilities, 80, 81
  UWIN, 81
WinZip, 272
WISDOM acrostic, 20
Wizard, 198
Word processor, 252, 254
Workflow, 150
  blackboard system, 169
  content-driven, 234
Wrapper, 132, 133, 135, 141
Writing, 18
  see also Documentation
www.pragmaticprogrammer.com, xxiii
WYSIWYG, *see* What You See Is What You Get

X

XEmacs editor, 266
Xerox Parc, 39
XSL, *see* eXtensible Style Language
xUnit, 194, 269

Y

yacc, 59
Yourdon, Edward, 10, 35
Y2K problem, 32, 208

Z

Z shell, 272

informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE

InformIT is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

Addison-Wesley | Cisco Press | EXAM/CRAM | IBM Press | QUE | PRENTICE HALL | SAMS | Safari Books Online

LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters. Visit **informit.com/newsletters**.
- Access FREE podcasts from experts at **informit.com/podcasts**.
- Read the latest author articles and sample chapters at **informit.com/articles**.
- Access thousands of books and videos in the Safari Books Online digital library at **safari.informit.com**.
- Get tips from expert blogs at **informit.com/blogs**.

Visit **informit.com/learn** to discover all the ways you can access the hottest technology content.

Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit **informit.com/socialconnect**.

PEARSON

THIS PRODUCT

informit.com/register

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account. You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram
IBM Press | Que | Prentice Hall | Sams

SAFARI BOOKS ONLINE

The Pragmatic Programmer

This card summarizes the tips and checklists found in *The Pragmatic Programmer.*

For more information about THE PRAGMATIC PROGRAMMERS LLC, source code for the examples, up-to-date pointers to Web resources, and an online bibliography, visit us at `www.pragmaticprogrammer.com`.

Quick Reference Guide

TIPS 1 TO 22

1. Care About Your Craft xix
Why spend your life developing software unless you care about doing it well?

2. Think! About Your Work xix
Turn off the autopilot and take control. Constantly critique and appraise your work.

3. Provide Options, Don't Make Lame Excuses 3
Instead of excuses, provide options. Don't say it can't be done; explain what *can* be done.

4. Don't Live with Broken Windows 5
Fix bad designs, wrong decisions, and poor code when you see them.

5. Be a Catalyst for Change 8
You can't force change on people. Instead, show them how the future might be and help them participate in creating it.

6. Remember the Big Picture 8
Don't get so engrossed in the details that you forget to check what's happening around you.

7. Make Quality a Requirements Issue 11
Involve your users in determining the project's real quality requirements.

8. Invest Regularly in Your Knowledge Portfolio 14
Make learning a habit.

9. Critically Analyze What You Read and Hear 16
Don't be swayed by vendors, media hype, or dogma. Analyze information in terms of you and your project.

10. It's Both What You Say and the Way You Say It 21
There's no point in having great ideas if you don't communicate them effectively.

11. *DRY–Don't Repeat Yourself* 27
Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

12. Make It Easy to Reuse 33
If it's easy to reuse, people will. Create an environment that supports reuse.

13. Eliminate Effects Between Unrelated Things 35
Design components that are self-contained, independent, and have a single, well-defined purpose.

14. There Are No Final Decisions 46
No decision is cast in stone. Instead, consider each as being written in the sand at the beach, and plan for change.

15. Use Tracer Bullets to Find the Target 49
Tracer bullets let you home in on your target by trying things and seeing how close they land.

16. Prototype to Learn 54
Prototyping is a learning experience. Its value lies not in the code you produce, but in the lessons you learn.

17. Program Close to the Problem Domain 58
Design and code in your user's language.

18. Estimate to Avoid Surprises 64
Estimate before you start. You'll spot potential problems up front.

19. Iterate the Schedule with the Code 69
Use experience you gain as you implement to refine the project time scales.

20. Keep Knowledge in Plain Text 74
Plain text won't become obsolete. It helps leverage your work and simplifies debugging and testing.

21. Use the Power of Command Shells 80
Use the shell when graphical user interfaces don't cut it.

22. Use a Single Editor Well 82
The editor should be an extension of your hand; make sure your editor is configurable, extensible, and programmable.

Tips 23 to 46

23. Always Use Source Code Control 88
Source code control is a time machine for your work—you *can* go back.

24. Fix the Problem, Not the Blame 91
It doesn't really matter whether the bug is your fault or someone else's—it is still your problem, and it still needs to be fixed.

25. Don't Panic When Debugging 91
Take a deep breath and THINK! about what could be causing the bug.

26. "select" Isn't Broken 96
It is rare to find a bug in the OS or the compiler, or even a third-party product or library. The bug is most likely in the application.

27. Don't Assume It—Prove It 97
Prove your assumptions in the actual environment—with real data and boundary conditions.

28. Learn a Text Manipulation Language 100
You spend a large part of each day working with text. Why not have the computer do some of it for you?

29. Write Code That Writes Code 103
Code generators increase your productivity and help avoid duplication.

30. You Can't Write Perfect Software 107
Software can't be perfect. Protect your code and users from the inevitable errors.

31. Design with Contracts 111
Use contracts to document and verify that code does no more and no less than it claims to do.

32. Crash Early 120
A dead program normally does a lot less damage than a crippled one.

33. Use Assertions to Prevent the Impossible 122
Assertions validate your assumptions. Use them to protect your code from an uncertain world.

34. Use Exceptions for Exceptional Problems 127
Exceptions can suffer from all the readability and maintainability problems of classic spaghetti code. Reserve exceptions for exceptional things.

35. Finish What You Start 129
Where possible, the routine or object that allocates a resource should be responsible for deallocating it.

36. Minimize Coupling Between Modules 140
Avoid coupling by writing "shy" code and applying the Law of Demeter.

37. Configure, Don't Integrate 144
Implement technology choices for an application as configuration options, not through integration or engineering.

38. Put Abstractions in Code, Details in Metadata . 145
Program for the general case, and put the specifics outside the compiled code base.

39. Analyze Workflow to Improve Concurrency 151
Exploit concurrency in your user's workflow.

40. Design Using Services 154
Design in terms of *services*—independent, concurrent objects behind well-defined, consistent interfaces.

41. Always Design for Concurrency 156
Allow for concurrency, and you'll design cleaner interfaces with fewer assumptions.

42. Separate Views from Models 161
Gain flexibility at low cost by designing your application in terms of models and views.

43. Use Blackboards to Coordinate Workflow 169
Use blackboards to coordinate disparate facts and agents, while maintaining independence and isolation among participants.

44. Don't Program by Coincidence 175
Rely only on reliable things. Beware of accidental complexity, and don't confuse a happy coincidence with a purposeful plan.

45. Estimate the Order of Your Algorithms 181
Get a feel for how long things are likely to take *before* you write code.

46. Test Your Estimates 182
Mathematical analysis of algorithms doesn't tell you everything. Try timing your code in its target environment.

Tips 47 to 70

47. Refactor Early, Refactor Often 186
Just as you might weed and rearrange a garden, rewrite, rework, and re-architect code when it needs it. Fix the root of the problem.

48. Design to Test 192
Start thinking about testing before you write a line of code.

49. Test Your Software, or Your Users Will 197
Test ruthlessly. Don't make your users find bugs for you.

50. Don't Use Wizard Code You Don't Understand . 199
Wizards can generate reams of code. Make sure you understand *all* of it before you incorporate it into your project.

51. Don't Gather Requirements—Dig for Them 202
Requirements rarely lie on the surface. They're buried deep beneath layers of assumptions, misconceptions, and politics.

52. Work with a User to Think Like a User 204
It's the best way to gain insight into how the system will *really* be used.

53. Abstractions Live Longer than Details 209
Invest in the abstraction, not the implementation. Abstractions can survive the barrage of changes from different implementations and new technologies.

54. Use a Project Glossary 210
Create and maintain a single source of all the specific terms and vocabulary for a project.

55. Don't Think Outside the Box—*Find* the Box 213
When faced with an impossible problem, identify the *real* constraints. Ask yourself: "Does it have to be done this way? Does it have to be done at all?"

56. Start When You're Ready 215
You've been building experience all your life. Don't ignore niggling doubts.

57. Some Things Are Better Done than Described . 218
Don't fall into the specification spiral—at some point you need to start coding.

58. Don't Be a Slave to Formal Methods 220
Don't blindly adopt any technique without putting it into the context of your development practices and capabilities.

59. Costly Tools Don't Produce Better Designs 222
Beware of vendor hype, industry dogma, and the aura of the price tag. Judge tools on their merits.

60. Organize Teams Around Functionality 227
Don't separate designers from coders, testers from data modelers. Build teams the way you build code.

61. Don't Use Manual Procedures 231
A shell script or batch file will execute the same instructions, in the same order, time after time.

62. Test Early. Test Often. Test Automatically. 237
Tests that run with every build are much more effective than test plans that sit on a shelf.

63. Coding Ain't Done 'Til All the Tests Run 238
'Nuff said.

64. Use Saboteurs to Test Your Testing 244
Introduce bugs on purpose in a separate copy of the source to verify that testing will catch them.

65. Test State Coverage, Not Code Coverage 245
Identify and test significant program states. Just testing lines of code isn't enough.

66. Find Bugs Once 247
Once a human tester finds a bug, it should be the *last* time a human tester finds that bug. Automatic tests should check for it from then on.

67. English is Just a Programming Language 248
Write documents as you would write code: honor the *DRY* principle, use metadata, MVC, automatic generation, and so on.

68. Build Documentation In, Don't Bolt It On 248
Documentation created separately from code is less likely to be correct and up to date.

69. Gently Exceed Your Users' Expectations 255
Come to understand your users' expectations, then deliver just that little bit more.

70. Sign Your Work 258
Craftsmen of an earlier age were proud to sign their work. You should be, too.

Checklists

✔ **Languages to Learn** .... page 17

Tired of C, C++, and Java? Try CLOS, Dylan, Eiffel, Objective C, Prolog, Smalltalk, or TOM. Each of these languages has different capabilities and a different "flavor." Try a small project at home using one or more of them.

✔ **The WISDOM Acrostic** .... page 20

What do you want them to learn?
What is their **i**nterest in what you've got to say?
How **s**ophisticated are they?
How much **d**etail do they want?
Whom do you want to **o**wn the information?
How can you **m**otivate them to listen to you?

✔ **How to Maintain Orthogonality** ..... page 34

- Design independent, well-defined components.
- Keep your code decoupled.
- Avoid global data.
- Refactor similar functions.

✔ **Things to prototype** .... page 53

- Architecture
- New functionality in an existing system
- Structure or contents of external data
- Third-party tools or components
- Performance issues
- User interface design

✔ **Architectural Questions** .... page 55

- Are responsibilities well defined?
- Are the collaborations well defined?
- Is coupling minimized?
- Can you identify potential duplication?
- Are interface definitions and constraints acceptable?
- Can modules access needed data—*when* needed?

✔ **Debugging Checklist** .... page 98

- Is the problem being reported a direct result of the underlying bug, or merely a symptom?
- Is the bug *really* in the compiler? Is it in the OS? Or is it in your code?
- If you explained this problem in detail to a coworker, what would you say?
- If the suspect code passes its unit tests, are the tests complete enough? What happens if you run the unit test with *this* data?
- Do the conditions that caused this bug exist anywhere else in the system?

✔ **Law of Demeter for Functions** ..... page 141

An object's method should call only methods belonging to:

- Itself
- Any parameters passed in
- Objects it creates
- Component objects

✔ **How to Program Deliberately** ...... page 172

- Stay aware of what you're doing.
- Don't code blindfolded.
- Proceed from a plan.
- Rely only on reliable things.
- Document your assumptions.
- Test assumptions as well as code.
- Prioritize your effort.
- Don't be a slave to history.

✔ **When to Refactor** .... page 185

- You discover a violation of the *DRY* principle.
- You find things that could be more orthogonal.
- Your knowledge improves.
- The requirements evolve.
- You need to improve performance.

✔ **Cutting the Gordian Knot** .... page 212

When solving *impossible* problems, ask yourself:

- Is there an easier way?
- Am I solving the right problem?
- *Why* is this a problem?
- What makes it hard?
- Do I have to do it this way?
- Does it have to be done at all?

✔ **Aspects of Testing** .... page 237

- Unit testing
- Integration testing
- Validation and verification
- Resource exhaustion, errors, and recovery
- Performance testing
- Usability testing
- Testing the tests themselves

Checklists from *The Pragmatic Programmer*, by Andrew Hunt and David Thomas. Visit `www.pragmaticprogrammer.com`.

Copyright © 2000 by Addison Wesley Longman, Inc.