Perl Medic
Transforming Legacy Code

Maintain, update and improve existing Perl code
Relentlessly practical and informative
Extensive use of code provides tangible solutions
Master writing the Disciplined Perl Program

Peter Scott
This page intentionally left blank
To my mother, for teaching me to read at a tender age after much begging. One thing led to another...
This page intentionally left blank
# Table of Contents

Preface ............................................................................................................................. xi
  Perl or perl? ...................................................................................................................... xv
  Obtaining Perl ............................................................................................................... xvi
  Historical Perl ................................................................................................................ xvii
  Who This Book Is For .................................................................................................... xviii
  Typographical Conventions ......................................................................................... xix
  For Further Reference ................................................................................................. xx
  Perl Versions ................................................................................................................... xxi
  Perl 6 ............................................................................................................................. xxi
  Acknowledgments .......................................................................................................... xxii

Chapter 1  Introduction (First Response) ................................................................. 1
  1.1 First Things First .................................................................................................... 2
  1.2 Reasons for Inheritance ...................................................................................... 3
  1.3 What Next? ......................................................................................................... 5
  1.4 Observe the Program in Its Natural Habitat ....................................................... 5
  1.5 Get Personal ....................................................................................................... 6
  1.6 Strictness ............................................................................................................. 6
  1.7 Warnings ............................................................................................................. 7

Chapter 2  Surveying the Scene ............................................................................ 13
  2.1 Versions .............................................................................................................. 14
  2.2 Part or Whole? ................................................................................................... 15
  2.3 Find the Dependencies ....................................................................................... 18

Chapter 3  Test Now, Test Forever (Diagnosis) ............................................... 25
  3.1 Testing Your Patience ......................................................................................... 26
# Table of Contents

3.2 Extreme Testing ................................................................. 27
3.3 An Example Using Test::Modules ........................................... 40
3.4 Testing Legacy Code .......................................................... 56
3.5 A Final Encouragement ....................................................... 59

Chapter 4 Rewriting (Transplants) ................................. 61
4.1 Strategizing ................................................................. 62
4.2 Why Are You Doing This? ............................................... 63
4.3 Style .............................................................................. 68
4.4 Comments ........................................................................ 72
4.5 Restyling .......................................................................... 74
4.6 Variable Renaming .......................................................... 75
4.7 Editing ............................................................................. 79
4.8 Line Editing ....................................................................... 80
4.9 Antipatterns ...................................................................... 84
4.10 Evolution .......................................................................... 94

Chapter 5 The Disciplined Perl Program ......................... 101
5.1 Package Variables vs. Lexical Variables ......................... 102
5.2 Warnings and Strictness ................................................. 107
5.3 use strict in Detail ....................................................... 110
5.4 use warnings in Detail .................................................... 117
5.5 Selective Disabling ........................................................ 119
5.6 Caveat Programmer ........................................................ 128
5.7 Perl Poetry ....................................................................... 129

Chapter 6 Restructuring (The Operating Table) ........... 131
6.1 Keep It Brief ................................................................. 132
6.2 Cargo Cult Perl .............................................................. 133
6.3 Escaping the Global Variable Trap ................................. 156
6.4 Debugging Strategies ..................................................... 157
Table of Contents

Chapter 7  Upgrading (Plastic Surgery)................................. 161
    7.1 Strategies ................................................................................................ 162
    7.2 Perl 4 ...................................................................................................... 163
    7.3 Perl 5.000 ............................................................................................... 164
    7.4 Perl 5.001 ............................................................................................... 165
    7.5 Perl 5.002 ............................................................................................... 165
    7.6 Perl 5.003 ............................................................................................... 166
    7.7 Perl 5.004 ............................................................................................... 166
    7.8 Perl 5.005 ............................................................................................... 167
    7.9 Perl 5.6.0 ................................................................................................ 169
    7.10 Perl 5.6.1 ................................................................................................ 170
    7.11 Perl 5.8.0 ............................................................................................... 170
    7.12 Perl 5.8.1 ............................................................................................... 171
    7.13 Perl 5.8.2 ............................................................................................... 172
    7.14 Perl 5.8.3 ............................................................................................... 172

Chapter 8  Using Modules (Genetic Enhancement) ............. 173
    8.1 The Case for CPAN ................................................................................ 174
    8.2 Using CPAN ........................................................................................... 182
    8.3 Improving Code with Modules .............................................................. 188
    8.4 Custom Perls .......................................................................................... 196

Chapter 9  Analysis (Forensic Pathology) ......................... 201
    9.1 Static Analysis ........................................................................................ 202
    9.2 Eliminating Superfluous Code ............................................................... 210
    9.3 Finding Inefficient Code ........................................................................ 212
    9.4 Debugging .............................................................................................. 216

Chapter 10  Increasing Maintainability (Prophylaxis)........... 225
    10.1 Making It Robust .................................................................................. 226
    10.2 Advanced Brevity ................................................................................. 235
    10.3 Documentation ..................................................................................... 239
    10.4 Custom Warnings ................................................................................ 242
    10.5 Version Control System Integration ...................................................... 244
### Table of Contents

**Chapter 11  A Case Study .................................................................247**

11.1  The Setup .................................................................248
11.2  Triage .................................................................251
11.3  Desperately Seeking Sanity ......................................255
11.4  Coming into the 21st Century ....................................259
11.5  Incorporating Modules Effectively, Part 1 ....................262
11.6  Incorporating Modules Effectively, Part 2 ....................265
11.7  Making It Mature, Part 1  ..........................................268
11.8  Making It Mature, Part 2 ..........................................272
11.9  Making It Mature, Part 3 ..........................................276
11.10 Advanced Modification ...........................................277

**Chapter 12  Conclusion (Prognosis) ........................................283**

12.1  In Conclusion ..........................................................284
12.2  Perl People ............................................................288
12.3  A Final Thought ........................................................291

**Appendix: Source Code ..........................................................293**

- Tie::Array::Bounded .................................................293
- Benchmark::TimeTick ................................................295
- smallprofpp ...............................................................300

**Bibliography .................................................................303**

**Index .................................................................307**

**About the Author ............................................................312**
“Worldwide, there are well over 200 billion lines of software that are fragmented, redundantly defined, hard to decipher, and highly inflexible . . . organizations run the risk of being mired down by a mountain of legacy code.”

Congratulations! Let’s say you just graduated with a computer science degree and now, bucking the economic trend, you’ve landed a job at a prestigious company with a large information technology department. You’re going to be replacing Bill, a programmer who won the lottery and was not seen or heard from again, save for a postcard from Puerto Vallarta two weeks later. Your coworkers warn you not to mention the postcard to your supervisor. You sit in Bill’s cubicle throwing out pieces of vendor advertising left in the center desk drawer, thinking about how you’re going to apply the elegant principles and sublime paradigms that professors inculcated in you at college. Just then, your supervisor arrives and, leaning over your shoulder, taps at your keyboard, bringing up a file.

“This is the last program Bill was working on. We think it’s almost finished. We’re behind schedule, so see if you can get it done by Thursday at the latest.”

As he leaves, you look at the program’s tangle of misindented lines and cryptic variable names, searching for comments, but the only ones you can find read, “XXX–Must change” and “Kludge!–But should work.” You wonder whether this is a corporate hazing ritual, but your instinct tells you otherwise.

Welcome to the real world.

In the real world, you’re lucky if you get to spend all your time developing one new program after another. Much of the time you’ll have to deal with someone else’s. In the real world, programmers take over responsibility for programs written by people they might not know, like, or agree with. Even if you’re fortunate enough to avoid this situation, you’ll still have to maintain your own code; and one day you’re going to look at something you wrote two years ago and ask, “What idiot wrote this?” Thereby arriving at more or less the same situation as the less fortunate programmers.

This book is about taking over Perl code, whether written by someone else or by yourself at a time when you were less wise about maintainability. Many problems of code inheritance are common to all languages, but I have noticed them especially in Perl.
Why does Perl tend to foster maintenance issues? The answer to this is the dark side of Perl’s strength and motto: “There’s More Than One Way To Do It” (enshrined in the acronym TMTOWTDI). Perl provides so many ways to do it that someone else quite possibly picked one that wasn’t your way, or might have used several different ways all in the same program.

The medical metaphor for this book stems from the rather drastic nature of the work we do as maintenance programmers. Often we must perform triage, deciding what code is worth saving and what is beyond redemption. Frequently we only have time for first aid, applying a field dressing to a ruptured program. We also have a hard time explaining our bills to the client. There may not be a Hippocratic Oath for programming, but it wouldn’t hurt to come up with one.

I wrote this book because I kept finding myself telling my students, “I’m going to teach you how to program Perl well, but I’d have to teach you a lot more before you could take over a program that wasn’t written well, and you wouldn’t appreciate taking that much time away from learning how to write good programs of your own.” So I’ve written a book to fill that need.

Perl is to computer languages as English is to human languages: bursting with irregular verbs, consistent only when it’s convenient, borrowing terms from other languages to form a great melting pot of syntax. Most computer languages are described in terms of some kind of functional niche (Pascal: teaching computer languages; FORTRAN: numeric analysis; Prolog: rule-driven expert systems; etc.). Perl’s makers simply describe it as “a language for getting your job done.” (See the preface to [WALL00].) Perl hosts a fantastic conglomeration of syntactic devices that allow a programmer from virtually any background to find a familiar foothold for learning the language.

The full picture isn’t as chaotic as this might imply: Larry Wall and others have done a brilliant job of tying together these eclectic devices into a framework that has an essential beauty. Therefore, just as the British speak of a “BBC English,” while many people program Perl with, say, a LISP or C accent, there is something approaching an “accentless Perl” style that leverages the language’s features to their best advantages. I will show how you can
“speak Perl like a native” in order to optimize the maintainability of your programs.

It’s true that you’re officially allowed to program Perl in “baby talk” and Perl gurus have promised “not to laugh.” (See the same preface.) But by the same token, while aviators call any landing you can walk away from a good one, what I’m doing in this book is helping you avoid having your pilot’s license revoked.

Perl is like those people behind the travelers’ help desk in airports; it’s very good at understanding you no matter how poor your command of their language is. Because there are so many ways to write a Perl program that is not only syntactically correct (Perl makes no objection to running it) but also semantically correct (the program does what it’s supposed to—at least in the situations it’s been tried in), there is a wide variety of Perl programming styles that you might encounter, ranging from beautiful to what can charitably be described as incomprehensible.

The savvy among you will take that information and ask, “Where do my programs fit on that scale?” Because someone else may end up inheriting your code, and you’d prefer that they not end up sending it to authors like me as bad examples to go in books like this. See Chapter 5 for more advice on avoiding scorn.

If your experience or image of Perl is limited to short, mundane scripts, this book will appear to be overkill. I want you to know that Perl can quite easily accommodate large—as in tens of thousands of lines of code, multiple modules, and multiple programmers—projects. Projects of the size that demand rigorous requirements, documentation, and testing. If you’re used to Perl programs escaping that sort of attention, I believe that is partly the result of a misperception of the role and power of Perl.

For example, if a C program is written to fulfill some requirement and turns out to be 1,000 lines long, then the common reaction is, “This must be serious . . . we’d better have code walkthroughs, acceptance testing, operational readi-
ness reviews, and static code analyses. Oh, and don’t forget the Help Desk training and documentation."

But if a Perl program that fulfills exactly the same requirements weighs in at 100 lines (and 10:1 is a typical compression ratio for C code to Perl), the reaction is more likely to be, “Ah, a simple utility . . . and in a plebeian scripting language to boot. Just plunk it in the delivery directory and get on with the next task.”

When a Perl program reaches the 1,000-line mark, however, the honeymoon is probably over. Much of what I have to say addresses large programs. Chapter 3 in particular will show you how to get the respect of development teams who are used to putting everything through regression testing.

Please also see my earlier book with Ed Wright, *Perl Debugged* (Addison-Wesley, 2001) for more advice on good practices for developing and debugging Perl programs.

**Perl or perl?**

When you read this book and other works about Perl, you’ll see an apparent inconsistency in capitalization: sometimes it’s written as “Perl”, and others as “perl”. There’s really no inconsistency; the authors are referring to two different things. Perl is the language itself; perl is the program that runs Perl programs. There is only one Perl, but there are many perls (one or more for each type of computer).

Sometimes this distinction gets a bit blurred: For instance, most people will write, “Perl objects to mismatched parentheses” when it is arguably the program that’s doing the objecting and not the language. Just don’t write “PERL”; Perl isn’t an acronym, it doesn’t stand for anything.¹ (Well, aside from standing

---

¹. “Practical Extraction and Reporting Language” is an ex post facto moniker of convenience. It’s a term of endearment, not a formal title. Ditto for “Pathological Eclectic Rubbish Lister.”
for diversity of expression, freedom from artificial constraints, and the right to have fun in your work. But we’ll get to those later.)

**Obtaining Perl**

It would be remiss of me to tell you so much about Perl without telling you how to get it, although these days it’s hard to avoid; you probably already have it, especially if you have any flavor of UNIX or Linux. The easiest way to find out whether you have it is to get a command prompt and type:

```
perl -v
```

and see if you get a response. If not, try:

```
whereis perl
```

which on a UNIX system or similar will look around for Perl. If that doesn’t work for you either, here are brief instructions on how to download and install Perl:

- For Microsoft Windows machines, get the free ActivePerl distribution:  [http://www.activeState.com/ActivePerl/download.htm](http://www.activeState.com/ActivePerl/download.htm)
- For Macintosh:  
  (That URL is for pre-X versions of the OS; Perl comes with Mac OS X and builds fine on it, too.)
- For binary distributions for all other machines:  
- For the source of perl itself:  
The source file you want is called stable.tar.gz. The file devel.tar.gz is for Perl developers or testers only, and the file latest.tar.gz is the same as stable.tar.gz for complex historical reasons. Anything mentioning “Ponie” will be for developers only through 2004 at least, and any perl with a three-number component version with an odd middle number is likewise a development version.

Building Perl from source on a supported UNIX architecture requires just these commands after you download and unpack the right file:

```
./Configure
make
make test
make install
```

The Configure step asks you zillions of questions, and most people won’t have a clue what many of those questions are talking about; but the default answers Configure recommends are usually correct.²

www.cpan.org is the master Comprehensive Perl Archive Network (CPAN) site, mirrored around the world. CPAN is also the official repository of contributed modules (see Section 8.1).

**Historical Perl**

You’re probably not used to seeing instructions on how to obtain an out-of-date version of Perl. But you just might have to do that under some circumstances that will be explored later in this book. (Note: The older the version of Perl, the less likely the references I am about to give will enjoy substantial longevity.) The timeline for all releases of Perl is in the *perlhist* documentation page.

2. If you want to let Configure use those recommendations and go on without asking you, you can give it the options `–des` and it’ll churn away happily without your intervention. If you need more platform-specific help, look in the distribution for the README.platform file corresponding to your system.
You can get all major versions starting with 5.004_05 from ftp://ftp.cpan.org/pub/CPAN/src/5.0/. Earlier versions of Perl 5 (with the exception of 5.004_04, which was widely used) exhibited significant bugs, memory leaks, and security holes and are harder to find. However, you can get what looks like every version of Perl ever released at http://retroperl.cpan.org/. Using a perl before version 5.003 is not an activity to be undertaken lightly. You will not receive bug fixes or any other support beyond a terse admonition to leave the Stone Age and upgrade to a real version. I am revealing this source only for cases in which you must use an old perl to verify operation of a legacy program that does not work on a modern perl. If you must get a perl 4 from there, the last and best version of Perl 4 is version 4.0.36.

Retroperl even includes versions 1.0, 1.010, 2.0, 2.001, 3.01. To call these of historical interest only would be an understatement. If you think you need to get one of these perls for any serious work you may be more in need of an archaeologist or a therapist. As an example of nonserious work, however, in 2002 Michael Schwern and others released an upgrade to Perl 1 (bringing it to version 1.0_15) as a birthday present to Perl and Larry Wall, to show that it could still work on modern machines. See http://dev.perl.org/perl1/.

Historical versions of modules are under http://backpan.cpan.org/modules/, but you’ll have to go down the authors branch to find what you want.

Who This Book Is For

If you’ve been working with Perl long enough to have heard terms like scalar, array, and hash, then you’re in the right place. Parts of this book are aimed at early beginners and some parts require more experience to comprehend. Feel free to skip past anything that’s over your head and come back to it at a later date.
**Typographical Conventions**

I use the following conventions in this book:

- **Standard text**: Times New Roman
- **Code examples and URLs**: Courier
- **User input**: Bold Courier

Sometimes my code examples have line breaks where none existed in the original. In places where these line breaks would cause problems or aren’t obvious, I’ve put a backslash (\) at the end of the line to indicate that the line should be joined with the next one.

When I want to show that I typed an end-of-file character in a terminal session, I show it as a Control-D on its own line:

\^D

even though those characters will normally be overwritten by the next thing to be printed. Substitute whatever the terminal driver on your operating system uses if not a Control-D (e.g., Control-Z, Return on Windows).

When referring to Perl modules, I will often add “.pm” to the end of one-word module names to follow common practice and avoid confusion, but leave it out of module names with multiple components because that is the convention. For example, CGI.pm, but IO::Socket.

Citations are referenced by a tag in square brackets, for example, [SCOTT01], and the details are given in the Bibliography near the end of the book.

I reference many pages that are part of the standard documentation that comes with every Perl; you can type perldoc followed by the page name and it will display. I show these in italics: for instance, perlsub is the page containing information about subroutines in Perl.
For Further Reference


Get introductions to Perl programming from the following:

- [SCHWARTZ01]
- [WALL00]
- [CHAPMAN98]
- [JOHNSON99]
- [HALL98]

Perl Versions

In this book, I refer to the latest “stable” version of Perl, which is 5.8.3 as of this writing. The vast majority of what I say works unaltered on older versions of Perl 5, but not Perl 4. If you use any version of Perl older than 5.004_04, you should upgrade for reasons unconnected with features: 5.003 had issues such as security problems and memory leaks. You can find out the version number of your perl by passing it the -v flag:

% perl -v
This is perl, v5.8.3 built for i586-linux
Copyright 1987-2003, Larry Wall
[...]

Perl won’t execute a script named on the command line if the -v flag is present. A more detailed description of your perl’s configuration can be obtained with the -V flag; if you use the perlbug program that comes with perl to issue a bug report, it automatically includes this information in your report.
A separate development track exists for Perl; you will know if you have one of those versions because the release number either contains an underscore followed by a number of 50 or larger or contains an odd number between two dots. Nothing is guaranteed to work in such a distribution; it’s intended for testing. If you find you have one and you didn’t want it, the person who downloaded your perl probably visited the wrong FTP link. This happens more often than most people would think; take a moment to check your perl if you’re not sure.

**Perl 6**

If you’ve spent much time in the Perl universe, you’ve heard about Perl 6. I won’t be covering how to port programs to Perl 6 for a very logical reason: It doesn’t exist yet.³

A stable version of Perl 6 is still a few years away. When it emerges however, it will bear approximately the resemblance to Perl 5 that a Lamborghini Countach does to a Volkswagen Beetle (well, the new one, anyway). (Which is to say, backward compatibility has been prioritized beneath new capability for the first time in Perl development.)

Don’t panic. Perl 4 hung around for an indecent time after Perl 5 came out and Perl 5 will be ported, maintained, and improved for many years after Perl 6 emerges. The recently started Ponie⁴ project (http://www.poniecode.org) will enable Perl 5 to run on top of the Parrot engine underlying Perl 6. Your Perl 5 programs will quite likely keep working as long as you do.

This book will be applicable in large measure to Perl 6 in any case; most of the Perl 6 magic involves not removing existing features, but adding cool new ones. If you want to find out more about Perl 6 anyway, see [RANDAL03].

---

3. Except for development versions that test some of the features that have been decided on recently.
4. Ponie stands for Perl On New Implementation Engine (even though it isn’t capitalized).
Acknowledgments

Elaine Ashton and Jarkko Hietaniemi provided many valuable comments and help that I was happy to incorporate. Thanks also to Allen Wyke, Dan Livingston, and Adam Turoff for thoughtful and comprehensive reviewing. I am extremely grateful to Uri Guttman for the most detailed feedback I have ever seen, and to Uri and his wife Linda for some fun diversions at Perl conferences. Any remaining errors are in no way attributable to any of these reviewers. Ed Wright wrote the FrameMaker styles for typesetting and Carl Miyatake provided a Japanese translation. Ann Palmer did the cover illustration and interior artwork.

I would like to thank Karen McLean at Prentice Hall for her saintly patience, and Mike Henderson at Addison-Wesley for taking on this project in the first place. Anne Garcia and Heather Mullane performed critical production duties; Vanessa Moore spent ages ensuring the excellence of the finished product. I also thank my many students for the valuable lessons they have provided me. Kudos also to my friends at the Jet Propulsion Laboratory who regularly accomplish the spectacular against overwhelming odds. The model question in Chapter 12 appears by permission of its author, Mary Byrne.

Lastly, I would like to thank my wife Grace, for her support, love, and constant help throughout a period much longer than either of us anticipated. Most spouses would have stopped there, but she also toiled tirelessly on the index, proofreading, and copy editing. Greater support an author could not expect.
“A crash is when your competitor’s program dies. When your program dies, it is an ‘idiosyncrasy’. Frequently, crashes are followed with a message like ‘ID 02’. ‘ID’ is an abbreviation for idiosyncrasy and the number that follows indicates how many more months of testing the product should have had.”

— Guy Kawasaki
This chapter might appear at first blush to be out of sequence. We’re steadily getting more specific in the details of taking over code, and here near the beginning is a chapter on how to do testing. Shouldn’t it be near the end?

In a word, no. I am going to describe a philosophy of testing that will revolutionize your development practices if you have not already encountered it. I will show you how to implement it for Perl code and give a detailed example. It is so pivotal to the development process that I want to make sure you see it as soon as possible. So yes, it really should come before everything else.

### 3.1 Testing Your Patience

Here’s the hard part. Creating tests while you’re writing the code for the first time is far, far easier than adding them later on. I know it looks like it should be exactly the same amount of work, but the issue is motivation. When robots are invented that can create code, they won’t have this problem, and the rest of us can mull over this injustice while we’re collecting unemployment pay (except for the guy who invented the robot, who’ll be sipping margaritas on a beach somewhere, counting his royalties and hoping that none of the other programmers recognize him).

But we humans don’t like creating tests because it’s not in our nature; we became programmers to exercise our creativity, but “testing” conjures up images of slack-jawed drones looking for defects in bolts passing by them on a conveyor belt.

The good news is that the Test:: modules make it easy enough to overcome this natural aversion to writing tests at the time you’re developing code. The point at which you’ve just finished a new function is when your antitesting hormones are at their lowest ebb because you want to know whether or not it works. Instead of running a test that gets thrown away, or just staring at the code long enough to convince yourself that it must work, you can instead write a real test for it, because it may not require much more effort than typing:
is(some_func("some", "inputs"), qr/some outputs/, "some_func works");

The bad news is that retrofitting tests onto an already complete application requires much more discipline. And if anything could be worse than that, it would be retrofitting tests onto an already complete application that you didn’t write.

There’s no magic bullet that’ll make this problem disappear. The only course of action that’ll take more time in the long run than writing tests for your inherited code is not writing them. If you’ve already discovered the benefits of creating automated tests while writing an application from scratch then at least you’re aware of how much they can benefit you. I’ll explore one way to make the test writing more palatable in the next chapter.

3.2 Extreme Testing

This testing philosophy is best articulated by the Extreme Programming (XP) methodology, wherein it is fundamental (see [BECK00]). On the subject of testing, XP says:

- Development of tests should precede development of code.
- All requirements should be turned into tests.
- All tests should be automated.
- The software should pass all its tests at the end of every day.
- All bugs should get turned into tests.

If you’ve not yet applied these principles to the development of a new project, you’re in for a life-altering experience when you first give them an honest try. Because your development speed will take off like a termite in a lumberyard.
Perl wholeheartedly embraces this philosophy, thanks largely to the efforts in recent years of a group of people including Michael Schwern, chromatic, and others. Because of their enthusiasm and commitment to the testing process, the number of tests that are run when you build Perl from the source and type “make test” has increased from 5,000 in Perl 5.004_04 (1997) to 70,000 in the current development version of Perl 5.9.0 (2004). That’s right, a 14-fold increase.

True to the Perl philosophy, these developers exercised extreme laziness in adding those thousands of tests (see sidebar). To make it easier to create tests for Perl, they created a number of modules that can in fact be used to test anything. We’ll take a look at them shortly.

What is it about this technology that brings such joy to the developer’s heart? It provides a safety net, that’s what. Instead of perennially wondering whether you’ve accidentally broken some code while working on an unrelated piece, you can make certain at any time. If you want to make a radical change to some interface, you can be sure that you’ve fixed all the dependencies because every scenario that you care about will have been captured in a test case, and running all the tests is as simple as typing “make test”. One month into creating a new system that comprised more than a dozen modules and as many programs, I had built up a test suite that ran nearly 600 tests with that one command, all by adding the tests as I created the code they tested. When I made a radical change to convert one interface from functional to object-oriented, it took only a couple of hours because the tests told me when I was done.

This technique has been around for many years, but under the label *regression testing*, which sounds boring to anyone who can even figure out what it means.\(^1\) However, using that label can be your entrance ticket to respectability when trying to convince managers of large projects that you know what you’re talking about.

---

1. It’s called regression testing because its purpose is to ensure that no change has caused any part of the program to *regress* back to an earlier, buggier stage of development.
What’s this about laziness? Isn’t that a pejorative way to describe luminaries of the Perl universe?

Actually, no; they’d take it as a compliment. Larry Wall enumerated three principal virtues of Perl programmers:

1. **Laziness:** “Hard work” sounds, well, hard. If you’re faced with a mindless, repetitive task—such as running for public office—then laziness will make you balk at doing the same thing over and over again. Instead of stifling your creative spirit, you’ll cultivate it by inventing a process that automates the repetitive task. If the Karate Kid had been a Perl programmer, he’d have abstracted the common factor from “wax on” and “wax off” shortly before fetching an orbital buffer. (Only to get, er, waxed, in the tournament from being out of shape. But I digress.)

2. **Impatience:** There’s more than enough work to do in this business. By being impatient to get to the next thing quickly, you’ll not spend unnecessary time on the task you’re doing; you’ll find ways to make it as efficient as possible.

3. **Hubris:** It’s not good enough to be lazy and impatient if you’re going to take them as an excuse to do lousy work. You need an unreasonable amount of pride in your abilities to carry you past the many causes for discouragement. If you didn’t, and you thought about all the things that could go wrong with your code, you’d either never get out of bed in the morning, or just quit and take up potato farming.

So what are these magic modules that facilitate testing?

### 3.2.1 The Test Module

Test.pm was added in version 5.004 of Perl. By the time Perl 5.6.1 was released it was superseded by the Test::Simple module, which was published to CPAN and included in the Perl 5.8.0 core. Use Test::Simple instead.
If you inherit regression tests written to use Test.pm, it is still included in
the Perl core for backward compatibility. You should be able to replace its use
with Test::Simple if you want to start modernizing the tests.

3.2.2 The Test::Simple Module

When I say “simple,” I mean *simple*. Test::Simple exports precisely one func-
tion, \texttt{ok()}. It takes one mandatory argument, and one optional argument. If its
first argument evaluates to true, it prints “ok”; otherwise it prints “not ok”. In
each case it adds a number that starts at one and increases by one for each call
to \texttt{ok()}. If a second argument is given, \texttt{ok()} then prints a dash and that argu-
ment, which is just a way of annotating a test.

Doesn’t exactly sound like rocket science, does it? But on such a humble
foundation is the entire Perl regression test suite built. The only other require-
ment is that we know how many tests we expected to run so we can tell if
something caused them to terminate prematurely. That is done by an argument
to the \texttt{use} statement:

\begin{verbatim}
use Test::Simple tests => 5;
\end{verbatim}

The output from a test run therefore looks like:

\begin{verbatim}
1..5
ok 1  - Can make a frobnitz
ok 2  - Can fliggle the frobnitz
not ok 3 - Can grikkle the frobnitz
ok 4  - Can delete the frobnitz
ok 5  - Can't use a deleted frobnitz
\end{verbatim}

Note that the first line says how many tests are expected to follow. That makes
life easier for code like Test::Harness (see Section 3.2.9) that reads this output
in order to summarize it.
3.2.3 The Test::More Module

You knew there couldn’t be a module called Test::Simple unless there was something more complicated, right? Here it is. This is the module you’ll use for virtually all your testing. It exports many useful functions aside from the same \texttt{ok()} as Test::Simple. Some of the most useful ones are:

\begin{verbatim}
is($expression, $value, $description)
    Same as \texttt{ok($expression eq $value, $description)}. So why bother? Because \texttt{is()} can give you better diagnostics when it fails.

like($attribute, qr/regex/, $description)
    Tests whether $attribute matches the given regular expression.

is_deeply($struct1, $struct2, $description)
    Tests whether data structures match. Follows references in each and prints out the first discrepancy it finds, if any. Note that it does not compare the packages that any components may be blessed into.

isa_ok($object, $class)
    Tests whether an object is a member of, or inherits from, a particular class.

can_ok($object_or_class, @methods)
    Tests whether an object or a class can perform each of the methods listed.

use_ok($module, @imports)
    Tests whether a module can be loaded (if it contains a syntax error, for instance, this will fail). Wrap this test in a \texttt{BEGIN} block to ensure it is run at compile time, viz: \texttt{BEGIN \{use_ok("My::Module")\}}
\end{verbatim}

There’s much more. See the Test::More documentation. I won’t be using any other functions in this chapter, though.

Caveat: I don’t know why you might do this, but if you \texttt{fork()} inside the test script, don’t run tests from child processes. They won’t be recognized by the parent process where the test analyzer is running.
3.2.4 The Test::Exception Module

No, there’s no module called Test::EvenMore. But there is a module you’ll have to get from CPAN that can test for whether code lives or dies: Test::Exception. It exports these handy functions:

lives_ok()
  Passes if code does not die. The first argument is the block of code, the second is an optional tag string. Note there is no comma between those arguments (this is a feature of Perl’s prototyping mechanism when a code block is the first argument to a subroutine). For example:

  lives_ok { risky_function() } "risky_function lives!";

dies_ok()
  Passes if the code does die. Use this to check that error-checking code is operating properly. For example:

  dies_ok { $] / 0 } "division by zero dies!";

throws_ok()
  For when you want to check the actual text of the exception. For example:

  throws_ok { some_web_function() } qr/URL not found/,
  "Nonexistent page get fails";

The second argument is a regular expression that the exception thrown by the code block in the first argument is tested against. If the match succeeds, so does the test. The optional third argument is the comment tag for the test. Note that there is a comma between the second and third arguments.

---

2. Yet. I once promised Mike Schwern a beer if he could come up with an excuse to combine the UNIVERSAL class and an export functionality into UNIVERSAL::exports as a covert tribute to James Bond. He did it. Schwern, I still owe you that beer . . .
3.2.5 The Test::Builder Module

Did you spot that all these modules have a lot in common? Did you wonder how you’d add a Test:: module of your own, if you wanted to write one?

Then you’re already thinking lazily, and the testing guys are ahead of you. That common functionality lives in a superclass module called Test::Builder, seldom seen, but used to take the drudgery out of creating new test modules.

Suppose we want to write a module that checks whether mail messages conform to RFC 822 syntax. We’ll call it Test::MailMessage, and it will export a basic function, msg_ok(), that determines whether a message consists of an optional set of header lines, optionally followed by a blank line and any number of lines of text. (Yes, an empty message is legal according to this syntax. Unfortunately, too few people who have nothing to say avail themselves of this option.) Here’s the module:

**Example 3.1 Using Test::Builder to Create Test::MailMessage**

```perl
package Test::MailMessage;
use strict;
use warnings;
use Carp;
use Test::Builder;
use base qw(Exporter);
our @EXPORT = qw(msg_ok);

my $test = Test::Builder->new;

sub import {
    my $self = shift;
    my $pack = caller;
    $test->exported_to($pack);
    $test->plan(@_);
}
```

$self->export_to_level(1, $self, 'msg_ok');
}

sub msg_ok {
    my $arg = shift;
    my $tester = _new();
    eval {
        if (defined(fileno($arg))) {
            while (<$arg>) {
                $tester->_validate($_);
            }
        } elsif (ref $arg) {
            $tester->_validate($_) for @$arg;
        } else {
            for ($arg =~ /(.*
)/g) {
                $tester->_validate($_);
            }
        }
    }
    $test->ok(!$@, shift);
    $test->diag($@) if $@;
}

sub _new {
    return bless { expect => "header" };}

sub _validate {
    my ($self, $line) = @_;  
    return if $self->{expect} eq "body";
In line 1 we put this module into its own package, and in lines 2 and 3 we set warnings and strictness to help development go smoothly. In lines 4 and 5 we load the Carp module so we can call \texttt{croak()}, and the Test::Builder module so we can create an instance of it. In lines 6 and 7 we declare this to be a subclass of the Exporter module, exporting to the caller the subroutine \texttt{msg\_ok()}. (Note that this is \textit{not} a subclass of Test::Builder.)

In line 9 we create a Test::Builder object that will do the boring part of testing for us. Lines 11 through 20 are copied right out of the Test::Builder documentation; the \texttt{import()} routine is what allows us to say how many tests we’re going to run when we \texttt{use} the module.

Lines 22 through 49 define the \texttt{msg\_ok()} function itself. Its single argument specifies the mail message, either via a scalar containing the message, a reference to an array of lines in the message, or a filehandle from which the message can be read. Rather than read all of the lines from that filehandle into memory, we’re going to operate on them one at a time because it’s not necessary to have the whole message in memory. That’s why we create the object \texttt{$\text{tester}$} in line 25 to handle each line: it will contain a memory of its current state.

Then we call the \texttt{_validate()} method of \texttt{$\text{tester}$} with each line of the message. Because that method will \texttt{croak()} if the message is in error, we wrap those loops in an \texttt{eval} block. This allows us easily to skip superfluous scanning of a message after detecting an error.
Finally, we see whether an error occurred; if an exception was thrown by `croak()` inside the `eval` block, `$@` will contain its text; otherwise `$@` will be empty. The `ok()` method of the `Test::Builder` object we created is the same function we’re used to using in `Test::Simple`; it takes a true or false value, and an optional tag string, which we pass from our caller. If we had an exception, we pass its text to `Test::Builder`’s `diag()` method, which causes it to be output as a comment during testing.

The `_new()` method in lines 50–53 is not called `new()` because it’s not really a proper constructor; it’s really just creating a state object, which is why we didn’t bother to make it inheritable. It starts out in life expecting to see a mail header.

Lines 56–70 validate a line of a message. Because anything goes in a message body, if that’s what we’re expecting we have nothing to do. Otherwise, if we’re expecting a header or header continuation line, then first we check for a continuation line (which starts with white space; this is how a long message header “overflows”). If we have a blank line (line 67), that separates the header from the body, so we switch to expecting body text.

Finally, we must at this point be expecting a header line, and one of those starts with non-white-space characters followed by a colon. If we don’t have that, the message is bogus; but if we do, the next line could be either a header line or a continuation of the current header (or the blank line separating headers from the body).

Here’s a simple test of the `Test::MailMessage` module:

```
1  #!/usr/bin/perl
2  use strict;
3  use warnings;
4
5  use lib qw(..);
6  use Test::MailMessage tests => 2;
7
8  msg_ok(<<EOM, "okay");
9  from: ok
```
3.2 Extreme Testing

```perl
subject: whatever

body

EOM

msg_ok(\*DATA, "bogus");

__END__

bogus mail

message

The result of running this is:

1..2
ok 1 - okay
not ok 2 - bogus

# Failed test (.test at line 14)
# Invalid header at ./test line 14
# Looks like you failed 1 tests of 2.

Although we only used one Test:: module, we could have used others, for example:

use Test::MailMessage tests z=> 2;
use Test::Exception;
use Test::More;

Only one of the use statements for Test::Modules should give the number of tests to be run. Do not think that each use statement is supposed to number the tests run by functions of that module; instead, one use statement gives the total number of tests to be run.

brian d foy⁴ used Test::Builder to create Test::Pod⁵, which is also worth covering.

---

⁴ That’s not a typo; he likes his name to be spelled, er, rendered that way, thus going one step farther than bell hooks.
⁵ Now maintained by Andy Lester.
3.2.6 The Test::Pod Module

Documentation in Perl need not be entirely unstructured. The Plain Old Documentation (POD) format for storing documentation in the Perl source code (see the perlpod manual page) is a markup language and therefore it is possible to commit syntax errors. So rather than wait until your users try to look at your documentation (okay, play along with me here—imagine that you have users who want to read your documentation), and get errors from their POD viewer, you can make sure in advance that the POD is good.

Test::Pod exports a single function, pod_ok(), which checks the POD in the file named by its argument. I’ll show an example of its use later in this chapter.

3.2.7 Test::Inline

If you’re thinking that tests deserve to be inside the code they’re testing just as much as documentation does, then you want Test::Inline. This module by Michael Schwern enables you to embed tests in code just like POD, because, in fact, it uses POD for that embedding.

3.2.8 Test::NoWarnings

Fergal Daly’s Test::NoWarnings (formerly Test::Warn::None) lets you verify that your code is free of warnings. In its simplest usage, you just use the module, and increment the number of tests you’re running, because Test::NoWarnings adds one more. So if your test starts:

use Test::More tests => 17;

then change it to:

use Test::NoWarnings;
use Test::More tests => 18;
and the final test will be that no warnings were generated in the running of the other tests.

### 3.2.9 The Test::Harness Module

Test::Harness is how you combine multiple tests. It predates every other Test::module, and you’ll find it in every version of Perl 5. Test::Harness exports a function, `runtests()`, which runs all the test files whose names are passed to it as arguments and summarizes their results. You won’t see one line printed per test; `runtests()` intercepts those lines of output. Rather you’ll see one line printed per test file, followed by a summary of the results of the tests in that file. Then it prints a global summary line. Here’s an example of the output:

```
t/01load....ok
 t/02tie.....ok
 t/03use.....ok
 t/04pod.....ok
 All tests successful.
 Files=4, Tests=24,  2 wallclock secs ( 1.51 cusr +  0.31 csys = 1.82 CPU)
```

As it runs, before printing “ok” on each line, you’ll see a count of the tests being run updating in place, finally to be overwritten by “ok”. If any fail, you’ll see something appropriate instead of “ok”.

You can use Test::Harness quite easily, for instance:

```%
perl -MTest::Harness -e 'runtests(glob "*.t")'
```

but it’s seldom necessary even to do that, because a standard Perl module makefile will do it for you. I’ll show you how shortly.

Test::Harness turns your regression tests into a full-fledged deliverable. Managers just love to watch the numbers whizzing around.
3.3 An Example Using Test:: Modules

Let’s put what we’ve learned to use in developing an actual application. Say that we want to create a module that can limit the possible indices of an array, a bounds checker if you will. Perl’s arrays won’t normally do that, so we need a mechanism that intercepts the day-to-day activities of an array and checks the indices being used, throwing an exception if they’re outside a specified range. Fortunately, such a mechanism exists in Perl; it’s called tieing, and pretty powerful it is too.

Because our module will work by letting us tie an array to it, we’ll call it Tie::Array::Bounded. We start by letting h2xs do the rote work of creating a new module:

```
% h2xs -AXn Tie::Array::Bounded
Writing Tie/Array/Bounded/Bounded.pm
Writing Tie/Array/Bounded/Makefile.PL
Writing Tie/Array/Bounded/README
Writing Tie/Array/Bounded/test.pl
Writing Tie/Array/Bounded/Changes
Writing Tie/Array/Bounded/MANIFEST
```

That saved a lot of time! h2xs comes with perl, so you already have it. Don’t be put off by the name: h2xs was originally intended for creating perl extensions from C header files, a more or less obsolete purpose now, but by dint of copious interface extension, h2xs now enjoys a new lease on life for creating modules. (In Section 8.2.4, I’ll look at a more modern alternative to h2xs.)

Don’t be confused by the fact that the file Bounded.pm is in the directory Tie/Array/Bounded. It may look like there’s an extra directory in there but the hierarchy that h2xs created is really just to help keep your sources straight. Everything you create will be in the bottom directory, so we could cd there. For instant gratification we can create a Makefile the way we would with any CPAN module:

6. If you’re smart enough to bring up $ ]; then you’re also smart enough to know that you shouldn’t be using it.
3.3 An Example Using Test::Modules

% cd Tie/Array/Bounded
% perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Tie::Array::Bounded

and now we can even run a test:

% make test

It even passes! This is courtesy of the file test.pl that h2xs created for us, which contains a basic test that the module skeleton created by h2xs passes. This is very good for building our confidence. Unfortunately, test.pl is not the best way to create tests. We’ll see why when we improve on it by moving test.pl into a subdirectory called “t” and rebuilding the Makefile before rerunning “make test”:

% mkdir t
% mv test.pl t/01load.t
% perl Makefile.PL
Writing Makefile for Tie::Array::Bounded
% make test

The big difference: “make test” knows that it should run Test::Harness over the .t files in the t subdirectory, thereby giving us a summary of the results.
There’s only one file in there at the moment, but we can create more if we want instead of having to pack every test into test.pl.

At this point you might want to update the MANIFEST file to remove the line for test.pl now that we have removed that file.

If you’re using Perl 5.8.0 or later, then your h2xs has been modernized to create the test in t/1.t; furthermore, it will use Test::More. But if you have a prior version of Perl, you’ll find the test.pl file we just moved uses the deprecated Test module, so let’s start from scratch and replace the contents of t/01load.t as follows:

```
#!/usr/bin/perl
use strict;
use warnings;

use Test::More tests => 1;
use blib;
BEGIN { use_ok("Tie::Array::Bounded") }
```

The `use blib` statement causes Perl to search in parent directories for a blib directory that contains the Tie/Array/Bounded.pm module created by make. Although we’ll usually run our tests by typing “make test” in the parent directory, this structure for a .t file allows us to run tests individually, which will be helpful when isolating failures.

Running this test either stand-alone (“./01load.t”) or with “make test” produces the same output as before (plus a note from `use blib` about where it found the blib directory), so let’s move on and add some code to Bounded.pm. First, delete some code that h2xs put there; we’re not going to export anything, and our code will work on earlier versions of Perl 5, so remove code until the executable part of Bounded.pm looks like this:

```

```

7. I name my tests with two leading digits so that they will sort properly; I want to run them in a predictable order, and if I have more than nine tests, test 10.t would be run before test 2.t, because of the lexicographic sorting used by the `glob()` function called by “make test”. Having done that, I can then add text after the digits so that I can also see what the tests are meant for, winding up with test names such as 01load.t.
package Tie::Array::Bounded;
use strict;
use warnings;
our $VERSION = '0.01';
1;

Now it’s time to add subroutines to implement tieing. tie is how to make possessed variables with Perl: Literally anything can happen behind the scenes when the user does the most innocuous thing. A simple expression like $world_peace++ could end up launching a wave of nuclear missiles, if $world_peace happens to be tied to Mutually::Assured::Destruction. (See, you can even use Perl to make covert political statements.)

We need a TIEARRAY subroutine; perltie tells us so. So let’s add an empty one to Bounded.pm:

sub TIEARRAY
{
}

and add a test to look for it in 01load.t:

use Test::More tests => 2;
use blib;
BEGIN { use_ok("Tie::Array::Bounded") }
can_ok("Tie::Array::Bounded", "TIEARRAY");

Running “make test” copies the new Bounded.pm into blib and produces:

% make test
cp Bounded.pm blib/lib/Tie/Array/Bounded.pm
PERL_DL_NONLAZY=1 /usr/local/bin/perl -Iblib/arch -Iblib/lib -I/usr/lib/perl5/5.6.1/i386-linux -I/usr/lib/perl5/5.6.1 -e 'use Test::Harness qw(&runtests $verbose); $verbose=0; runtests @ARGV;' t/*.t
t/01load....Using /home/peter/perl_Medic/Tie/Array/Bounded/blib/t/01load....ok
All tests successful.
Files=1, Tests=2, 0 wallclock secs (0.29 cusr + 0.02 csys = 0.31 CPU)

We have just doubled our number of regression tests!

It may seem as though we’re taking ridiculously small steps here. A subroutine that doesn’t do anything? What’s the point in testing for that? Actually, the first time I ran that test, it failed: I had inadvertently gone into overwrite mode in the editor and made a typo in the routine name. The point in testing every little thing is to build your confidence in the code and catch even the dumbest errors right away.

So let’s continue. We should decide on an interface for this module; let’s say that when we tie an array we must specify an upper bound for the array indices, and optionally a lower bound. If the user employs an index out of this range, the program will die. For the sake of having small test files, we’ll create a new one for this test and call it 02tie.t:

```perl
#!/usr/bin/perl
use strict;
use warnings;
use Test::More tests => 1;
use blib;
use Tie::Array::Bounded;

my $obj = tie my @array, "Tie::Array::Bounded";
isa_ok($obj, "Tie::Array::Bounded");
```

So far, this just tests that the underlying object from the `tie` is or inherits from `Tie::Array::Bounded`. Run this test before you even add any code to `TIE-ARRAY` to make sure that it does indeed fail:

```
% ./02tie.t
1..1
Using /home/peter/perl_Medic/Tie/Array/Bounded/t/../blib
not ok 1 - The object isa Tie::Array::Bounded
#     Failed test (.02tie.t at line 10)
```
3.3 An Example Using Test:: Modules

# The object isn't defined
# Looks like you failed 1 tests of 1.

We’re not checking that the module can be used or that it has a TIEARRAY method; we already did those things in 01load.t. Now we know that the test routine is working properly. Let’s make a near-minimal version of TIEARRAY that will satisfy this test:

```perl
sub TIEARRAY
{
    my $class = shift;
    my ($upper, $lower);
    return bless { upper => $upper,
                  lower => $lower,
                  array => []
                  }, $class;
}
```

Now the test passes. Should we test that the object is a hashref with keys upper, lower, and so on? No—that’s part of the private implementation of the object and users, including tests, have no right peeking in there.

Well, it doesn’t really do to have a bounded array type if the user doesn’t specify any bounds. A default lower bound of 0 is obvious because most bounded arrays will start from there anyway and be limited in how many elements they can contain. It doesn’t make sense to have a default upper bound because no guess could be better than any other. We want this module to die if the user doesn’t specify an upper bound (italicized code):

```perl
sub TIEARRAY
{
    my ($class, %arg) = @_;  
    my ($upper, $lower) = @arg{qw(upper lower)};
    $lower ||= 0;
    croak "No upper bound for array" unless $upper;
    return bless { upper => $upper,
                  lower => $lower,
                  array => []
                  }, $class;
}
```
Note that when we want to die in a module, the proper routine to use is `croak()`. This results in an error message that identifies the calling line of the code, and not the current line, as the source of the error. This allows the user to locate the place in their program where they made a mistake. `croak()` comes from the Carp Module, so we added a `use Carp` statement to Bounded.pm (not shown).

Note also that we set the lower bound to a default of 0. True, if the user didn’t specify a lower bound, `$lower` would be undefined and hence evaluate to 0 in a numeric context. But it’s wise to expose our defaults explicitly, and this also avoids warnings about using an uninitialized value. Modify 02tie.t to say:

```perl
use Test::More tests => 1;
use Test::Exception;
use blib;
use Tie::Array::Bounded;

dies_ok { tie my @array, "Tie::Array::Bounded" } "Croak with no bound specified";
```

If you’re running 02tie.t as a stand-alone test, remember to run `make` in the parent directory after modifying Bounded.pm so that Bounded.pm gets copied into the blib tree.

Great! Now let’s add back in the test that we can create a real object when we `tie` with the proper calling sequence:

```perl
my $obj;
lives_ok { $obj = tie my @array, "Tie::Array::Bounded",
        upper => 42
    } "Tied array okay";
isa_ok($obj, "Tie::Array::Bounded");
```

and increase the number of tests to 3. (Notice that there is no comma after the block of code that’s the first argument to `dies_ok` and `lives_ok`.)
All this testing has gotten us in a pedantic frame of mind. The user shouldn’t be allowed to specify an array bound that is negative or not an integer. Let’s add a statement to TIEARRAY (in italics):

```perl
sub TIEARRAY {
  my ($class, %arg) = @_; 
  my ($upper, $lower) = @arg{qw(upper lower)};
  $lower ||= 0;
  croak "No upper bound for array" unless $upper;
  \D/ and croak "Array bound must be integer"
    for ($upper, $lower);
  return bless { upper => $upper,
                  lower => $lower,
                  array => []
            }, $class;
}
```

and, of course, test it:

```perl
throws_ok { tie my @array, "Tie::Array::Bounded", upper => -1 } 
  qr/must be integer/, "Non-integral bound fails";
```

Now we’re not only checking that the code dies, but that it dies with a message matching a particular pattern.

We’re really on a roll here! Why don’t we batten down the hatches on this interface and let the user know if they gave us an argument we’re not expecting:

```perl
sub TIEARRAY {
  my ($class, %arg) = @_; 
  my ($upper, $lower) = delete @arg{qw(upper lower)};
  croak "Illegal arguments in tie" if %arg;
  croak "No upper bound for array" unless $upper;
  $lower ||= 0;
  \D/ and croak "Array bound must be integer"
    for ($upper, $lower);
}
```
The succinctness of our approach depends on the underappreciated hash slice and the delete() function. Hash slices [GUTTMAN98] are a way to get multiple elements from a hash with a single expression, and the delete() function removes those elements while returning their values. Therefore, anything left in the hash must be illegal.

We’re nearly done with the pickiness. There’s one final test we should apply. Have you guessed what it is? We should make sure that the user doesn’t enter a lower bound that’s higher than the upper one. Can you imagine what the implementation of bounded arrays would do if we didn’t check for this? I can’t, because I haven’t written it yet, but it might be ugly. Let’s head that off at the pass right now:

sub TIEARRAY
{
my ($class, %arg) = @_;
my ($upper, $lower) = delete @arg{qw(upper lower)};
croak "Illegal arguments in tie" if %arg;
$lower ||= 0;
croak "No upper bound for array" unless $upper;
/\D/ and croak "Array bound must be integer"
for ($upper, $lower);
croak "Upper bound < lower bound" if $upper < $lower;
return bless { upper => $upper,
               lower => $lower,
               array => []
               }, $class;
}
3.3 An Example Using Test:: Modules

and the new test goes at the end of 02tie.t (italicized):

**Example 3.2 Final Version of 02tie.t**

```perl
#!/usr/bin/perl
use strict;
use warnings;

use Test::More tests => 6;
use Test::Exception;
use blib;
use Tie::Array::Bounded;

dies_ok { tie my @array, "Tie::Array::Bounded" }
   "Croak with no bound specified";

my $obj;
lives_ok { $obj = tie my @array, "Tie::Array::Bounded",
   upper => 42 }
   "Tied array okay";

isa_ok($obj, "Tie::Array::Bounded");

throws_ok { tie my @array, "Tie::Array::Bounded", upper => -1 }
   qr/must be integer/, "Non-integral bound fails";

throws_ok { tie my @array, "Tie::Array::Bounded", frogs => 10 }
   qr/Illegal arguments/, "Illegal argument fails";

throws_ok { tie my @array, "Tie::Array::Bounded",
   lower => 2, upper => 1 }
   qr/Upper bound < lower/, "Wrong bound order fails";

Whoopee! We’re nearly there. Now we need to make the tied array behave properly, so let’s start a new test file for that, called 03use.t:

```perl
#!/usr/bin/perl
use strict;
use warnings;
```
use Test::More tests => 1;
use Test::Exception;
use blib;
use Tie::Array::Bounded;

my @array;
tie @array, "Tie::Array::Bounded", upper => 5;
lives_ok { $array[0] = 42 } "Store works";

As before, let’s ensure that the test fails before we add the code to implement it:

% t/03use.t
1..1
Using /home/peter/perl_Medic/Tie/Array/Bounded/blib
not ok 1 - Store works
# Failed test (t/03use.t at line 13)
# died: Can't locate object method "STORE" via package
"Tie::Array::Bounded" (perhaps you forgot to load
"Tie::Array::Bounded") at t/03use.t line 13.
# Looks like you failed 1 tests of 1.

How about that. The test even told us what routine we need to write. perltie tells us what it should do. So let’s add to Bounded.pm:

sub STORE
{
    my ($self, $index, $value) = @_;  
    $self->{_bound_check}($index);
    $self->{_array}[$index] = $value;
}

sub _bound_check
{
    my ($self, $index) = @_; 
    my ($upper, $lower) = @{$self}{qw(upper lower)};  
    croak "Index $index out of range [$lower, $upper]"
        if $index < $lower || $index > $upper;
}
We’ve abstracted the bounds checking into a method of its own in anticipation of needing it again. Now 03use.t passes, and we can add another test to make sure that the value we stored in the array can be retrieved:

```perl
is($array[0], 42, "Fetch works");
```

You might think this would fail for want of the `FETCH` method, but in fact:

```perl
ok 1 - Store works
Can't locate object method "FETCHSIZE" via package "Tie::Array::Bounded" (perhaps you forgot to load "Tie::Array::Bounded") at t/03use.t line 14.
# Looks like you planned 2 tests but only ran 1.
# Looks like your test died just after 1.
```

Back to `perltie` to find out what `FETCHSIZE` is supposed to do: return the size of the array. Easy enough:

```perl
sub FETCHSIZE
{
    my $self = shift;
    scalar @{$self->{array}};
}
```

Now the test does indeed fail for want of `FETCH`, so we’ll add that:

```perl
sub FETCH
{
    my ($self, $index) = @_;
    $self->_bound_check($index);
    $self->{array}[$index];
}
```

Finally we are back in the anodyne land of complete test success. Time to add more tests:
throws_ok { $array[6] = "dog" } qr/out of range/,          "Bounds exception";
is_deeply(\@array, [ 42 ], "Array contents correct");

These work immediately. But an ugly truth emerges when we try another simple array operation:

lives_ok { push @array, 17 } "Push works";

This results in:

not ok 5 - Push works
# Failed test (t/03use.t at line 19)
# died: Can't locate object method "PUSH" via package
"Tie::Array::Bounded" (perhaps you forgot to load
"Tie::Array::Bounded") at t/03use.t line 19.
# Looks like you failed 1 tests of 5.

Inspecting `perltie` reveals that `PUSH` is one of several methods it looks like we’re going to have to write. Do we really have to write them all? Can’t we be lazier than that?

Yes, we can. The Tie::Array core module defines `PUSH` and friends in terms of a handful of methods we have to write: `FETCH`, `STORE`, `FETCHSIZE`, and `STORESIZE`. The only one we haven’t done yet is `STORESIZE`:

```perl
sub STORESIZE
{
    my ($self, $size) = @_;
    $self->bound_check($size-1);
    $#{$self->{array}} = $size - 1;
}
```

We need to add near the top of Bounded.pm:

---

8. Remember, if you find yourself doing something too rote or boring, look for a way to get the computer to make it easier for you. Top of the list of those ways would be finding code someone else already wrote to solve the problem.
use base qw(Tie::Array);

to inherit all that array method goodness.

This is a big step to take, and if we didn’t have canned tests, we might wonder what sort of unknown havoc could be wrought upon our module by a new base class if we misused it. However, our test suite allows us to determine that, in fact, nothing has broken.

Now we can add to 01load.t the methods FETCH, STORE, FETCHSIZE, and STORESIZE in the can_ok test:

**Example 3.3 Final Version of 01load.t**

```perl
#!/usr/bin/perl
use strict;
use warnings;
use Test::More tests => 2;
use blib;
BEGIN { use_ok("Tie::Array::Bounded") }
can_ok("Tie::Array::Bounded", qw(TIEARRAY STORE FETCH FETCHSIZE STORESIZE));
```

Because our tests pass, let’s add as many more as we can to test all the boundary conditions we can think of, leaving us with a final 03use.t file of:

**Example 3.4 Final Version of 03use.t**

```perl
#!/usr/bin/perl
use strict;
use warnings;
use Test::More tests => 15;
use Test::Exception;
use blib;
use Tie::Array::Bounded;
```
my $RANGE_EXCEP = qr/out of range/;

my @array;
tie @array, "Tie::Array::Bounded", upper => 5;
lives_ok { $array[0] = 42 } "Store works";
is($array[0], 42, "Fetch works");

throws_ok { $array[6] = "dog" } $RANGE_EXCEP,
  "Bounds exception";
is_deeply(\@array, [ 42 ], "Array contents correct");

lives_ok { push @array, 17 } "Push works";
is($array[1], 17, "Second array element correct");

lives_ok { push @array, 2, 3 } "Push multiple elements works";
is_deeply(\@array, [ 42, 17, 2, 3 ], "Array contents correct");

lives_ok { splice(@array, 4, 0, qw(apple banana)) } 
  "Splice works";
is_deeply(\@array, [ 42, 17, 2, 3, 'apple', 'banana' ], 
  "Array contents correct");

throws_ok { push @array, "excessive" } $RANGE_EXCEP,
  "Push bounds exception";
is(scalar @array, 6, "Size of array correct");

tie @array, "Tie::Array::Bounded", lower => 3, upper => 6;

throws_ok { $array[1] = "too small" } $RANGE_EXCEP,
  "Lower bound check failure";

lives_ok { @array[3..6] = 3..6 } "Slice assignment works";
throws_ok { push @array, "too big" } $RANGE_EXCEP,
  "Push bounds exception";

Tests are real programs, too. Because we test for the same exception repeatedly, we put its recognition pattern in a variable to be lazy.

Bounded.pm, although not exactly a model of efficiency (our internal array contains unnecessary space allocated to the first $lower elements that will never be used), is now due for documenting, and h2xs filled out some POD
3.3 An Example Using Test:: Modules

stubs already. We’ll flesh it out to the final version you can see in the Appendix. I’ll go into documentation more in Chapter 10.

Now we create 04pod.t to test that the POD is formatted correctly:

Example 3.5 Final Version of 04pod.t

```perl
#!/usr/bin/perl
use strict;
use warnings;

use Test::Pod tests => 1;
use blib;
use Tie::Array::Bounded;
pod_ok($INC{"Tie/Array/Bounded.pm"});
```

There’s just a little trick there to allow us to run this test from any directory, since all the others can be run with the current working directory set to either the parent directory or the t directory. We load the module itself and then get Perl to tell us where it found the file by looking it up in the `%INC` hash, which tracks such things (see its entry in `perlvar`).

With a final “make test”, we’re done:

```
Files=4, Tests=24, 2 wallclock secs ( 1.58 cusr +  0.24 csys = 1.82 CPU)
```

We have a whole 24 tests at our fingertips ready to be repeated any time we want.

You can get more help on how to use these modules from the module Test::Tutorial, which despite the module appellation contains no code, only documentation.

With only a bit more work, this module could have been submitted to CPAN. See [TREGAR02] for full instructions.
3.4 Testing Legacy Code

“This is all well and good,” I can hear you say, “but I just inherited a swamp of 27 programs and 14 modules and they have no tests. What do I do?”

By now you’ve learned that it is far more appealing to write tests as you write the code they test, so if you can possibly rewrite this application, do so. But if you’re stuck with having to tweak an existing application, then adopt a top-down approach. Start by testing that the application meets its requirements . . . assuming you were given requirements or can figure out what they were. See what a successful run of the program outputs and how it may have changed its environment, then write tests that look for those effects.

3.4.1 A Simple Example

You have an inventory control program for an aquarium, and it produces output files called cetaceans.txt, crustaceans.txt, molluscs.txt, pinnipeds.txt, and so on. Capture the output files from a successful run and put them in a subdirectory called success. Then run this test:

Example 3.6 Demonstration of Testing Program Output

```perl
my @Success_files;
BEGIN {
    @Success_files = glob "success/*.txt";
}
use Test::More tests => 1 + 2 * @Success_files;

is(system("aquarium"), 0, "Program succeeded");

for my $success (@Success_files) {
    (my $output = $success) =~ s#.*/##;
    ok(-e $output, "$output present");
```

56
First, we capture the names of the output files in the success subdirectory. We do that in a BEGIN block so that the number of names is available in line 6. In line 8 we run the program and check that it has a successful return code. Then for each of the required output files, in line 14 we test that it is present, and in line 16 we use the UNIX cmp utility to check that it matches the saved version. If you don’t have a cmp program, you can write a Perl subroutine to perform the same test: Just read each file and compare chunks of input until finding a mismatch or hitting the ends of file.

3.4.2 Testing Web Applications

A Common Gateway Interface (CGI) program that hasn’t been developed with a view toward automated testing may be a solid block of congealed code with pieces of web interface functionality sprinkled throughout it like raisins in a fruit cake. But you don’t need to rip it apart to write a test for it; you can verify that it meets its requirements with an end-to-end test. All you need is a program that pretends to be a user at a web browser and checks that the response to input is correct. It doesn’t matter how the CGI program is written because all the testing takes place on a different machine from the one the CGI program is stored on.

The WWW::Mechanize module by Andy Lester comes to your rescue here. It allows you to automate web site interaction by pretending to be a web browser, a function ably pulled off by Gisle Aas’ LWP::UserAgent module. WWW::Mechanize goes several steps farther, however (in fact, it is a subclass of LWP::UserAgent), enabling cookie handling by default and providing methods for following hyperlinks and submitting forms easily, including transparent handling of hidden fields.9

---

9. If you’re thinking, “Hey! I could use this to write an agent that will stuff the ballot box on surveys I want to fix,” forget it; it’s been done before. Chris Nandor used Perl to cast thousands of votes for his choice for American League All-Star shortstop [GLOBE99]. And this was before WWW::Mechanize was even invented.
Suppose we have an application that provides a login screen. For the usual obscure reasons, the login form, login.html, contains one or more hidden fields in addition to the user-visible input fields, like this:

```html
<FORM ACTION="login.cgi" METHOD="POST">
  <INPUT NAME="username" TYPE="text">
  <INPUT NAME="password" TYPE="text">
  <INPUT NAME="fruglido" TYPE="hidden" VALUE="grilkku">
  <INPUT TYPE="Submit">
</FORM>
```

On successful login, the response page greets the user with “Welcome, ” followed by the user’s first name. We can write this test for this login function:

**Example 3.7 Using WWW::Mechanize to Test a Web Application**

```perl
#!/usr/bin/perl
use strict;
use warnings;

use WWW::Mechanize;
use Test::More tests => 3;

my $URL = 'http://localhost/login.html';
my $USERNAME = 'peter';
my $PASSWORD = 'secret';

my $ua = WWW::Mechanize->new;
ok($ua->get($URL)->is_success, "Got first page")
or die $ua->res->message;

$ua->set_fields(username => $USERNAME,

password => $PASSWORD);
ok($ua->submit->is_success, "Submitted form")
or die $ua->res->message;

like($ua->content, qr/Welcome, Peter/, "Logged in okay");
```
In line 12 we create a new WWW::Mechanize user agent to act as a pretend
browser, and in line 13 we test to see if it was able to get the login page; the
get() method returns a HTTP::Response object that has an is_success() method. If something went wrong with fetching the page the false value will be passed through the ok() function; there’s no point in going further so we might as well die() (line 14). We can get at the HTTP::Response object again via the res() method of the user agent to call its message() method, which returns the text of the reason for failure.

In lines 16 and 17 we provide the form inputs by name, and in line 18 the submit() method of the user agent submits the form and reads the response, again returning an HTTP::Response object allowing us to verify success as before. Once we have a response page we check to see whether it looks like what we wanted.

Note that WWW::Mechanize can be used to test interaction with any web application, regardless of where that application is running or what it is written in.

3.4.3 What Next?

The kind of end-to-end testing we have been doing is useful and necessary; it is also a lot easier than the next step. To construct comprehensive tests for a large package, we must include unit tests; that means testing each function and method. However, unless we have descriptions of what each subroutine does, we won’t know how to test them without investigative work to find out what they are supposed to do. I’ll go into those kinds of techniques later.

3.5 A Final Encouragement

In addition to the more obvious benefits, constructing tests before or contemporaneously with code development encourages good program interface design.
Take, for example, a web-based system incorporating CGI programs and an extensive back end. When writing tests for such a beast it will become rapidly apparent that testing it via the CGI interface is tedious at best. You have to wait for the whole server round trip to happen, and most of that time is occupied by the operation of software and networks you may not be responsible for and don’t want to test. By cutting out the fat and calling the back end directly you’ll eliminate the tedium. However, you don’t want to leave out interface code that should be tested. So you make the CGI programs as small as possible: Gather user inputs, pass them on to interface-independent code, take the outputs of that code and format them for the user. That way you have so little code in the CGI programs that testing them will be a snap. A single call to a CGI program itself will accomplish that, and every other test can concentrate on going directly to the back end.

Congratulations. You’ve just reinvented the Model-View-Controller pattern (see [GAMMA95]), a device generally recognized to be a pretty good thing.

3.5.1 A Final Caveat

Tests can’t replace using your brain. They’re only as smart as their creator: If there’s a bug that isn’t tested for, the tests won’t find it. You still have to look at what you write and think about it, or it could harbor a bug that you didn’t think to test for. Testing just saves you from having to repeat the same train of thought over and over again.
Index

Symbols
#!. See shebang line
$] 111

A
ActivePerl, obtaining xvi
ActiveState 186, 202
AFS 103, 143
algorithms, importance of 216
Alzabo 98
anonymous hash reference 152
@ARGV 104
Attribute::Handlers 171, 238
attrs pragma 167
AUTOLOAD 121, 282
AutoLoader, using pragmas with 119
Avogadro’s Number 167
AxKit 98, 196

B
B::Deobfuscate 24
B::Deparse 111, 113
B::Xref 210
bareword 115
Barr, Graham 270
beautification 74
Benchmark::Timer 205
Benchmark::TimeTick 206, 295
block comments 73
brace style, BSD/Allman 69
brace style, Kernighan and Ritchie 70
btreeperl 197

C
-c command-line flag 109, 158
C to Perl, compression ratio xv
can_ok() 31
carp() 243
Carp::Assert 228, 229
CGI programs, testing 57
CGI::Kwiki 97, 286
cgi-lib.pl 189
CGI.pm 152, 166, 188, 261, 262
chromatic 28
cisamperl 197
Class::Contract 230
Class::Inspector 210
client/server, evolving 99
code unit 132
code, obfuscated 23
comments 72
Comprehensive Perl Archive Network 175, 177, 182
concatenation, pointless 135
conferences, Perl 287
confess() 126
Config::Auto 95
constant pragma 77, 166, 167
constants 78
constants, symbolic 77
Conway, Damian 230, 238
CPAN See Comprehensive Perl Archive Network
CPAN, reliability of modules 175
CPANPLUS 187
CPAN.pm 182, 183, 184
CPAN.pm, shell mode 183
croak() 126
cTREEperl 197
CVS 245

D
^D xix
Date::Calc 191
Date::Manip 190
DBI.pm 98
DBM files 97
DBM files, advantages of 194
Debug::FaultAutoBT 209
debugger commands, basic 217
debugger, Perl 216, 217, 218, 219, 220, 221
debugging 157, 159
deobfuscation 24
deparse module 113
dependencies, code for listing 19
dependencies, finding 18
Design by Contract 230
Devel::Cover 212
Devel::Coverage 210
Index

Devel::DProf 213
Devel::LeakTrace 209
Devel::SmallProf 214
dies_ok() 32, 46
Digest::MD5 171
duaperl 197
Dua.pm 255, 256, 259

E
$^E 165
-e command-line flag 109
EmailErrors module 126
en passant technique 64
error handling, evolving 95
Error.pm 96
event handling, evolving 98
Event.pm 98
Exception::Class 96
exception-handling block 132
@EXPORT 105
external data storage, evolving 97
external information representation, evolving 97
external programs 91, 92
external programs, excessive calling 91
Extreme Programming 3, 27
ExtUtils::MakeMaker 187, 244

F
FatalsToEmail module 126
FETCH() 51, 294
FETCHSIZE() 51, 294
fields pragma 167
File::Find 103, 146
File::Path 92
filter, LDAP 269
Filter::Simple 171
Filter::Util::Call 171
find() 17
FindBin 166
FindBin module 166
for, as a synonym for foreach 155
foreach 154
foreach, postfixed 167, 168
foy, brian d 37

G
\G 166, 167
G 167
Getopt::Std 112
getopts() 112

H
h2xs 40, 42
hash, multidimensional 151
hashes, not using 85
Hopkins, Sharon 129
HTML parsing 192
HTML::Template 98, 266
hubris 29

I
idiosyncrasy 25
impatience 29
indentation 69
indenting, continuation line 70
Ingerson, Brian 215
initialization, superfluous 148
Inline:: modules 165
Inline::C 215
in-place editing 16
input interfaces, evolving 95
interperl 198
interprocess communication, evolving 99
IO::Socket 191
is() 31
is_deeply() 31
@ISA 105
isa_ok() 31
isqlperl 198

L
-l command-line flag 109
Larry Wall 29
laziness 28, 29
legacy code, writing tests for 56
levels of perl mastery 65
libnet 171
like() 31
limerick, bad 108
line editing 80
List::Util 171
Index

lives_ok 32, 46
local 103, 104, 106
lock 168
Log::Dispatch::FileRotate 96
Log::Log4perl 96, 234
logging, evolving 96
lvalue subroutines 169, 170

M
-M command-line flag 109
MacPerl, obtaining xvi
magic numbers 86
Mail::Audit 195
Mail::Internet 195
Mail::Send 195, 272
Mail::SpamAssassin 195
makefile 40
Makefile.PL 41
map 143
Memoize module 171
MIME::Base64 171
MIME::Lite 195
Model-View-Controller pattern 60
modularity, evolving 95
module installation, manual 184
module, definition 15
Module::Build 187
Module::CoreList 163
Module::Info 209
msg_ok() 34
my() 66, 106, 110, 112, 147

N
Net::Cmd 99
Net::LDAP 259
Net::LDAP::Entry 268, 277
Net::LDAP::Filter 270
Net::LDAP::Search 259
newsgroups 284
no 119
no strict 'refs' 120
no strict 'subs' 120
no strict 'vars' 119
no warnings 123
no warnings 'redefine' 123
Nunavut 176

O
$^O 165
ok() 30
open() 169
operator overloading 238
operator, "diamond" 103
optimizing for speed 215
optimizing, different reasons 10
oraperl 198
our() 105, 110, 119, 169
overloading 238

P
parentheses, useless 147
periodicals, Perl 287
PERL xv
Perl 4 14, 163
Perl 4.018 162
Perl 5.000 164
Perl 5.001 165
Perl 5.002 165
Perl 5.003 166
Perl 5.004 166
Perl 5.005 167
Perl 5.6.0 169
Perl 5.6.1 170
Perl 5.8.0 162, 170
Perl 6 xxi
Perl golf 67
Perl Mongers 287
Perl Object Environment 98
Perl poetry 129
Perl source, obtaining xvi
Perl success stories 180
Perl, accentless xiii
Perl, maintainability xiii
Perl, obtaining xvi
Perl, old versions of xviii
Perl, version detection 14
perlhist xvi
perlrun 109
perlstyle 77
pgperl 198
.ph files 17
Phoenix, Tom 150
Plain Old Documentation 240, 241, 242
Index

pmtools 23  
POD formatters 242  
POD  See Plain Old Documentation  
pod_ok() 38  
pod2xml 242  
POE  See Perl Object Environment  
poetry mode 115  
poetry, bad 108  
portals, Perl 285  
PPM 186  
pragma 102, 107, 115  
précis 79  
pseudo-hashes 167

R  
rationale, design 11  
RCS 244  
Real World, the xii  
refactoring 253  
references 150, 151  
references, symbolic 115, 116, 149  
regression testing 28  
regular expressions, debugging 222  
regular expressions, ineffective 88  
Retroperl xviii  
return codes, checking 142  
return statement 153  
revision control 5  
RFC 822 33  
rocisperl 197  
Rolsky, Dave 227  
runtests() 39

S  
Safe.pm 235  
scalar() 136  
Scalar::Util 171  
SCCS 5, 245  
Schwartz, Randal 126, 150  
Schwern, Michael 28  
scope 105  
scoping, dynamic 105, 106  
scoping, lexical 105  
search.cpan.org 176, 177, 178  
sentinel comments 73  
shebang line 14, 15  
shebang line, modifying 16, 17  
$SIG{__DIE__} 165  
$SIG{__WARN__} 165  
slices, hash 141  
slices, useless 140  
smallprofpp 300  
SOAP 99  
Socket.pm 191  
sort pragma 171  
Source Code Control System 5  
source file typing 15  
sqlperl 198  
Storable module 171  
STORE() 50, 294  
STORESIZE() 294  
stream processing, line by line 144  
strict 102  
stringification, useless 134  
style 68  
subroutine prototypes 165  
subroutine, redefinition 124  
Subversion 245  
suidperl 166, 171  
Switch module 171  
sybperl 199

T  
-T command-line flag 171, 269  
-t command-line flag 170, 171  
Tangram 98  
tarballs 182  
Template Toolkit 98  
temporary variables 81  
Test:: modules 26  
Test::Builder 33, 35  
Test::Exception 32  
Test::Harness 39  
Test::Inline 38  
Test::MailMessage 33  
Test::More 31, 171  
Test::NoWarnings 38  
Test::Pod 38  
Test::Simple 30, 171  
Test::Tutorial 55  
testing, regression 28  
test.pl 42
Index

Test.pm 29
Text::Balanced 171
throws_ok() 32, 47
Thunderbirds 140
Tie::Array::Bounded 40, 44, 293
Tie::File 171
TIEARRAY 43, 293
tieing 40, 235
Time::HiRes 171
tkperl 199
TMTOWTDI xiii
typeglobs 121

U
undef 153
Unicode 170
uniperl 199
UNIVERSAL::exports 32
universe, fundamental structure of 79
URI Parsing 193
URL.pm 134
use bibl 42
use lib 22
use strict 6, 66, 107, 110, 128
use strict 'refs' 115
use strict 'subs' 113
use strict 'vars' 110
use vars 110, 119, 169
use warnings 6, 66, 107, 117, 128
use_ok() 31

V
variable renaming 75
variable, lexical 102, 103
variable, package 102, 106, 110
$VERSION 105, 244
virtues of Perl programmers, principal 29
Visual Perl 202

W
$^W 124
-w command-line flag 117
warnings in modules 118
warnings, custom 242
warnings, disabling 124
warnings, lexical 126
warnings, whether to leave in production code 8
Web server processing, evolving 98
WWW::Mechanize 20, 57, 59

X
XML-RPC 99
XP  See Extreme Programming

Y
YAML 97
YAPC 287
Yet Another Society 288
About the Author

PETER J. SCOTT runs Pacific Systems Design Technologies, providing Perl training, application development, and enterprise systems analysis. He was a speaker on the 2002 Perl Whirl cruise and at YAPC::Canada, and he founded his local Perl Mon- ger group. A software developer since 1981 and a Perl developer since 1992, he has also created pro- grams for NASA’s Jet Propulsion Laboratory. Scott graduated from Cambridge University, England, with a Master of Arts Degree in Computer Science and now lives in the Pacific Northwest with his wife Grace, a cat, and a parrot, at least one of which also uses Perl. He is the lead author of *Perl Debugged*. 