Extracted from:

Modern Perl, Fourth Edition

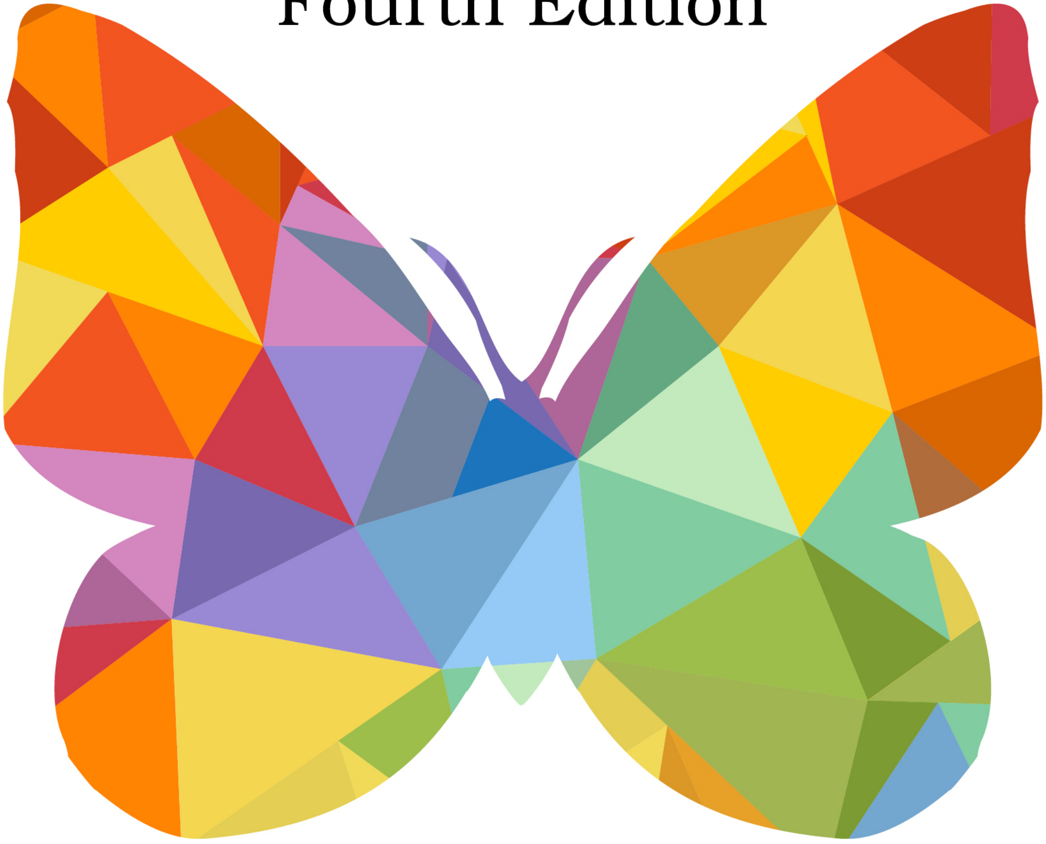
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Modern Perl

Fourth Edition



chromatic

The Classic Reference, Updated for Perl 5.22

Modern Perl, Fourth Edition

chromatic

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (index)
Linda Recktenwald (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-088-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2015

CHAPTER 8

Style and Efficacy

To program well, we must find the balance between getting the job done on time and doing the job right. We must balance time, resources, and quality. Programs have bugs. Programs need maintenance and expansion. Programs have multiple programmers. A beautiful program that never delivers value is worthless, but an awful program that cannot be maintained is a risk waiting to happen.

Skilled programmers understand their constraints and write the right code.

To write Perl well, you must understand the language. You must also cultivate a sense of good taste for the language and the design of programs. The only way to do so is to practice—not just writing code but maintaining and reading good code.

This path has no shortcuts, but it does have guideposts.

Writing Maintainable Perl

Maintainability is the nebulous measurement of how easy it is to understand and modify a program. Write some code. Come back to it in six months (or six days). How long does it take you to find and fix a bug or add a feature? That's maintainability.

Maintainability doesn't measure whether you have to look up the syntax for a built-in or a library function. It doesn't measure how someone who has never programmed before will or won't read your code. Assume you're talking to a competent programmer who understands the problem you're trying to solve. How much work does she have to put in to understand your code? What problems will she face in doing so?

To write maintainable software, you need experience solving real problems, an understanding of the idioms and techniques and style of your programming language, and good taste. You can develop all of these by concentrating on a few principles:

- *Remove duplication.* Bugs lurk in sections of repeated and similar code—when you fixed a bug in one piece of code, did you fix it in others? When you updated one section, did you update the others?

  Well-designed systems have little duplication. They use functions, modules, objects, and roles to extract duplicate code into distinct components that accurately model the domain of the problem. The best designs sometimes allow you to add features by *removing* code.

- *Name entities well.* Your code tells a story. Every name you choose for a variable, function, module, class, and role allows you to clarify or obfuscate your intent. Choose your names carefully. If you're having trouble choosing good names, you may need to rethink your design or study your problem in more detail.

- *Avoid unnecessary cleverness.* Concise code is good when it reveals the intention of the code. Clever code hides your intent behind flashy tricks. Perl allows you to write the right code at the right time. Choose the most obvious solution when possible. Experience and good taste will guide you.

  Some problems require clever solutions. When this happens, encapsulate this code behind a simple interface and document your cleverness.

- *Embrace simplicity.* If everything else is equal, a simpler program is easier to maintain than a complex program. Simplicity means knowing what's most important and doing just that.

Sometimes you need powerful, robust code. Sometimes you need a one-liner. Simplicity means building only what you need. This is no excuse to avoid error checking or modularity or validation or security. Simple code can use advanced features. Simple code can use CPAN modules—and many of them. Simple code may require work to understand. Yet simple code solves problems effectively, without *unnecessary* work.

Writing Idiomatic Perl

Perl borrows liberally from other languages. Perl lets you write the code you want to write. C programmers often write C-style Perl, just as Java programmers write Java-style Perl and Lisp programmers write Lispy Perl. Effective Perl programmers write Perlish Perl by embracing the language's idioms:

- *Understand community wisdom.* Perl programmers often debate techniques and idioms fiercely. Perl programmers also often share their work, and not just on the CPAN. Pay attention; there's not always one and only one best way to do things. The interesting discussions happen about the trade-offs between various ideals and styles.
- *Follow community norms.* Perl is a community of toolsmiths who solve broad problems, including static code analysis (Perl::Critic), reformatting (Perl::Tidy), and private distribution systems (CPAN::Mini, Carton, Pinto). Take advantage of the CPAN infrastructure; follow the CPAN model of writing, documenting, packaging, testing, and distributing your code.
- *Read code.* Join a mailing list such as Perl Beginners (http://learn.perl.org/faq/beginners.html) and otherwise immerse yourself in the community.[1] Read code and try to answer questions—even if you never post your answers, writing code to solve one problem every workday will teach you an enormous amount very quickly.

CPAN developers, Perl mongers, and mailing list participants have hard-won experience solving problems in myriad ways. Talk to them. Read their code. Ask questions. Learn from them and let them guide—and learn from—you.

Writing Effective Perl

Writing maintainable code means designing maintainable code. Good design comes from good habits:

- *Write testable code.* Writing an effective test suite (Testing on page ?) exercises the same design skills as writing effective code. Code is code. Good tests also give you the confidence to modify a program while keeping it running correctly.
- *Modularize.* Enforce encapsulation and abstraction boundaries. Find the right interfaces between components. Name things well and put them where they belong. Modularity forces you to think about similarities and differences and points of communication where your design fits together. Find the pieces that don't fit well. Revise your design until they do fit.
- *Follow sensible coding standards.* Effective guidelines discuss error handling, security, encapsulation, API design, project layout, and other facets of maintainable code. Excellent guidelines help developers communicate with each other with code. If you look at a new project and find nothing

1. http://www.perl.org/community.html

surprises you, that's great! Your job is to solve problems with code. Let your code—and the infrastructure around it—speak clearly.

- *Exploit the CPAN.* Perl programmers solve problems and then share those solutions. The CPAN is a force multiplier; search it first for a solution or partial solution to your problem. Invest time in research to find full or partial solutions you can reuse. It will pay off.

If you find a bug, report it. Patch it, if possible. Submit a failing test case. Fix a typo. Ask for a feature. Say "Thank you!" Then, when you're ready—when you create something new or fix something old in a reusable way—share your code.