Stylish Perl

Programming and programming *well* are related, but distinct skills. If we only wrote programs once and never had to modify or maintain them, if our programs never had bugs, if we never had to choose between using more memory or taking more time, and if we never had to work with other people, we wouldn't have to worry about how well we program. To program well, you must understand the differences between potential solutions based on specific priorities of time, resources, and future plans.

Writing Perl well means understanding how Perl works. It also means developing a sense of good taste. To develop that skill, you must practice writing and maintaining code and reading good code. There are no shortcuts--but you can improve the effectiveness of your practice by following a few guidelines.

Writing Maintainable Perl

The easier your program is to understand and to modify, the better. This is *maintainability*. Set aside your current program for six months, then try to fix a bug or add a feature. The more maintainable the code, the less artificial complexity you will encounter making changes.

To write maintainable Perl you must:

* *Remove duplication.* Perl offers many opportunities to use abstraction to reduce and remove duplication. Functions, objects, roles, and modules, for example, allow you to define models of the problem and your solution.

  The more duplication in your system, the more work it is to make a necessary change, and the more likely you will forget to make a change in every place necessary. The less duplication in your system, the more likely you've found an effective design for your problem. The best designs allow you to add features while removing code overall.

* *Name entities well.* Everything you can name in your system--functions, classes, methods, variables, modules--can aid or hinder clarity. The ease with which you can name these entities reveals your understanding of the problem and the cohesion of your design. Your design tells a story, and every word you use effectively can help you remember that story when you must later maintain the code.

* *Avoid unnecessary cleverness.* Novices sometimes mistake cleverness for conciseness. Concise code avoids unnecessary structure or complexity. Clever code sometimes prefers its own cleverness to simplicity. Perl offers many approaches to solve similar problems. Often one form is more readable. Sometimes one form is faster, or simpler. Usually one is more obvious in context.

  You can't always avoid the dark corners of Perl, and some problems require cleverness to solve effectively. Only good taste and experience will help you evaluate the appropriate level of cleverness. As a rule of thumb, if you're prouder of explaining your solution to your coworkers than you are of solving a problem, your code may have unnecessary complexity.

  If you *do* need clever code, encapsulate it behind a simple interface and document your cleverness very well.

* *Embrace simplicity.* Given two programs which solve the same problem, the simplest is almost always easier to maintain. Simplicity doesn't require you to eschew advanced Perl knowledge, or to avoid using libraries, or to pound out hundreds of lines of procedural code.

  Simplicity means that you've solved the problem at hand effectively without adding anything you don't need. This is no excuse to avoid error checking or verification or validation or security. Instead it's a reminder to think about what's really important. Sometimes you don't need frameworks, or objects, or complex data structures. Sometimes you do. Simplicity means knowing the difference.

Writing Idiomatic Perl

Perl steals ideas from other languages as well as from the wild world outside of programming. Perl tends to claim these ideas by making them Perlish. To write Perl well, you must know how experienced Perl programmers write it.

* *Understand community wisdom.* The Perl community often debates techniques, sometimes fiercely. Yet even these disagreements offer enlightenment on specific design tradeoffs and styles. You know your specific needs, but CPAN authors, CPAN developers, your local Perl Mongers group, and other programmers have experience solving similar problems. Talk to them. Read their public code. Ask questions. Learn from them and let them learn from you.

* *Follow community norms.* The Perl community isn't always right, especially if your needs are very specific or unique, but it works continually to solve problems as broadly as possible. Perl's testing and packaging tools work best when you organize your code as if you were to distribute it on the CPAN. Adopt the standard approaches to writing, documenting, packaging, testing, and distributing your code, to take advantage of these tools.

> Similarly, CPAN distributions such as `Perl::Critic` and `Perl::Tidy` and `CPAN::Mini` can make your work simpler and easier.

* *Read code.* Join in a mailing list such as the Perl Beginners list (http://learn.perl.org/faq/beginners.html), sign up for PerlMonks (http://perlmonks.org/), and otherwise immerse yourself in the Perl Community (http://www.perl.org/community.html has copious links). You'll have plenty of opportunities to see how other people solve their problems, good and bad. Learn from the good (it's often obvious) and the bad (to see what to avoid).

> Writing a few lines of code to solve a problem someone else posted is a great way to learn.

Writing Effective Perl

Knowing Perl's syntax and semantics is only the beginning. You can only achieve good design if you follow habits to *encourage* good design.

* *Write testable code.* Perhaps the best way to ensure that you can maintain code is to write an effective test suite. Writing test code well exercises the same design skills as designing programs well; never forget that test code is still code. Even so, a good test suite will give you confidence that you can modify a program and not break existing behaviors you care about.

* *Modularize.* Break your code into individual modules to enforce encapsulation and abstraction boundaries. Make a habit of this and you'll recognize individual units of code which do too many things. You'll identify multiple units that work too tightly together.

> Modularity also forces you to manage different levels of abstraction; you must consider how the entities of your system work together. There's no better way to learn the value of abstraction than having to revise systems into effective abstractions.

* *Take advantage of the CPAN.* The single best force multiplier for any Perl 5 program is the amazing library of reusable code available for anyone to use. Thousands of developers have written tens of thousands of modules to solve more problems than you can imagine, and the CPAN only continues to grow. Community standards for documentation, for packaging, for installation, and for testing contribute to the quality of the code, and the CPAN's centrality in modern Perl has helped the Perl community grow in knowledge, in wisdom, and in efficacy.

> Whenever possible, search the CPAN first--and ask your fellow community members--for advice on solving your problems. You may even report a bug, or submit a patch, or produce your own distribution on the CPAN. Nothing demonstrates you're an effective Perl programmer more than helping other people solve their problems.

* *Establish sensible coding standards.* Effective guidelines establish policies for error handling, security, encapsulation, API design, project layout, and other maintainability concerns. Excellent guidelines evolve as you and your team understand each other and your projects better. The goal of

programming is to solve problems, and the goal of coding standards is to help you communicate your intentions clearly.