

The Perl Journal

Coverage Testing with *Pod::Coverage* and *Devel::Cover*

Andy Lester • 3

CPAN in the Desert

brian d foy • 7

Tracking TV Shows with Palm and Perl

Deborah Pickett • 9

Lessons Learned Converting Java to Perl

Simon Cozens • 13

Evaluating Short-Circuited Boolean Expressions

Randal L. Schwartz • 17

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Eric Forste:

Mastering Perl for Bioinformatics • 20

Source Code Appendix • 22

LETTER FROM THE EDITOR

I Need a Supercomputer

Okay, I don't really need a supercomputer. I don't have any global weather modeling or thermonuclear explosion analysis that I need to do just now. But I *want* to need a supercomputer. There's just something about the thought of all that horsepower at my fingertips that's really appealing. And frankly, it looks like it would be fun to build a supercomputing cluster.

Certainly the folks at Virginia Tech made it look like fun. In fact, their success has led to a new formula for building supercomputers: Buy 1100 Apple G5s, promise some hungry grad students all the pizza they can eat, and set them to work installing I/O cards and rackmounting the computers. In three months, VT built a 10-teraflop cluster for \$5.2 million, which is less than peanuts in the supercomputing world. (Check it out at http://computing.vt.edu/research_computing/terascale/.)

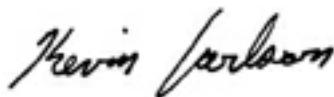
Clusters aren't new. The Beowulf cluster computing project is celebrating its 10th birthday this year. Beowulf proved the concept that commodity, off-the-shelf systems could form a reliable base on which to build the world's fastest computers. The plummeting cost of building such a cluster, however, is new. So is the ease with which they can be set up. You can buy clusters prebuilt from Dell; and Apple, keen to have more research dollars, is releasing its XGrid software to grease the supercomputing skids. (Apple is careful to point out, however, that XGrid all by itself won't replace true high-performance clustering software.)

Of course, most of us won't be building massive clusters in our basements. You still need lots of machines (and lots of expensive refrigeration) to really qualify as a competitive supercomputer. But home networks with more than five computers are becoming commonplace. It's not hard to imagine the day when grid computing services can be built in to mainstream operating systems, transparently allowing grid-aware apps on the desktop that use the untapped CPU cycles on your network.

But do we need that? Processors are already embarrassingly fast, and most apps don't really find themselves CPU-bound. Ten years ago, I used to wait several times a day for my computer to think about something. I can't remember the last time that happened. (With a mainstream desktop app, that is. Compiling with gcc is another story. And I still wait for disk-intensive operations all the time, of course.) But things change. Ten years ago, the horsepower to edit digital video didn't exist in most desktop machines. Ditto with professional-quality digital audio editing. Today, anyone can try their hand at these tasks. Who knows? Maybe the complex 3D animation done by professional special-effects firms will one day become the stuff of a mainstream, commercial app.

We're already seeing CPU-hungry tasks like digital audio moving onto the grid. Computer-based music is highly processor-bound: There's a clear upper limit to the number of simultaneous parts a computer musician can use, and software is emerging that can spread the processing load across a local network. WormHole, from apulSoft (<http://www.apulsoft.ch.vu>), is a good recent example of this. It routes audio across a LAN for real-time processing by other machines.

Only time will tell if we're really on the road toward ubiquitous grid computing. What we can say for certain is that where there are unused compute cycles, there are applications waiting in the wings to consume them.



Kevin Carlson
Executive Editor
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to editors@tpj.com, faxed to (650) 513-4618, or mailed to *The Perl Journal*, 2800 Campus Drive, San Mateo, CA 94403.

THE PERL JOURNAL (ISSN 1545-7567) is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: \$18.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2004 by CMP Media LLC, unless otherwise noted. All rights reserved.



The Perl Journal

EXECUTIVE EDITOR

Kevin Carlson

MANAGING EDITOR

Della Song

ART DIRECTOR

Margaret A. Anderson

NEWS EDITOR

Shannon Cochran

EDITORIAL DIRECTOR

Jonathan Erickson

COLUMNISTS

Simon Cozens, brian d foy, Moshe Bar, Randal Schwartz

CONTRIBUTING EDITORS

Simon Cozens, Dan Sugalski, Eugene Kim, Chris Dent, Matthew Lanier, Kevin O'Malley, Chris Nandor

INTERNET OPERATIONS

DIRECTOR

Michael Calderon

SENIOR WEB DEVELOPER

Steve Goyette

WEB DEVELOPER

Bryan McCormick

WEBMASTERS

Sean Coady, Joe Lucca

MARKETING / ADVERTISING

PUBLISHER

Timothy Trickett

MARKETING DIRECTOR

Jessica Hamilton

GRAPHIC DESIGNER

Carey Perez

THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403
650-513-4300. <http://www.tpj.com/>

CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND

MARKETING Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

PRESIDENT, TECHNOLOGY SOLUTIONS Robert Faletta

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED

TECHNOLOGIES Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenber

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER ENTERPRISE

ARCHITECTURE GROUP Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE

DEVELOPMENT MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING

SOLUTIONS Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES

Marie Myers

Perl News

Happy Sweet Sixteen, Perl

As of December 18, the anniversary of the day in 1987 that Larry Wall released Perl 1.000, Perl is 16 years old. To mark the occasion, Richard Clamp released an updated Perl 1 with support for gcc 3. You can find the new perl-1.0_16 release on CPAN or at http://unixbeard.net/~richardc/perl-1.0_16.tar.gz.

ActiveState also commemorated the day, issuing a press release praising Perl's robustness, rapid development cycle, and enduring popularity. The company said its ActivePerl software had been downloaded 7 million times.

The original man page for Perl 1 described the language as an "interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information." Larry went on to say that Perl "is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal)."

The full Perl Timeline (which begins in 1960, with the invention of hypertext) is at <http://history.perl.org/PerlTimeline.html>.

Gatherings Announced for 2004

A call for papers has been sent out for the Nordic Perl Workshop, which will be held March 27–28 in Copenhagen, Denmark. Papers about prototyping, testing, optimization, and development with Perl are particularly encouraged. Along with lightning talks and standard talks, the workshop will feature panel talks, where the speaker invites a group of experts to discuss the subject. Submissions are due by February 15; for details see <http://perlworkshop.dk/2004/>.

Also scheduled for March 27–28 is YAPC::Taipei::2004. "The topic of this conference is 'Projects for Developers,'" writes the conference organizer, Hsin-Chan Chien. "We will unveil 'Open-Foundry,' a collaboration environment based on widely used Perl projects such as Mason, RT, Sympa, and Kwiki." Details will be posted at <http://taipei.pm.org/>.

Also, proposals for the O'Reilly Open Source Software Convention 2004 will be accepted until February 9. OSCon 2004, which will include the 8th annual Perl Conference, will be held July 26–30 in Portland, Oregon. The call for participation is at http://conferences.oreillynet.com/cs/os2004/create/e_sess; for lightning talk proposals, see <http://perl.plover.com/lt/osc2004/>.

Take the Perl Community Survey

Taiwan's Open Source Software Foundry is conducting a survey of members in the Perl community, "to understand how Taiwan's developers can better communicate and work within international communities." The results of the survey will be used as part of

the Taiwanese government's Free Software Promotion Initiative. A final report and guidelines for local developers will be made available in both Chinese and English, and published under a Creative Commons license. The survey questions are listed at <http://blog.whiteg.net/survey.html>; to participate, mail your answers to whiteg@whiteg.net.

Perthon Translates Python to Perl

Perthon, a new SourceForge project (<http://perthon.sourceforge.net/>) led by David Manura, aims to "automatically translate Python into human-readable Perl." It relies on Damian Conway's *Parse::RecDescent* and Perl regular expressions for the parsing and lexing. Perthon is currently in a 0.1, pre-alpha state; eventually, Manura suggests, it could be useful for running Python code in Perl, easily combining Python and Perl code, or writing Perl programs "using a Pythonic syntactic sugar."

Template Toolkit Gets Funding

Andy Wardley has secured funding from Fotango, a UK consultancy group, to spend the next few months working full-time on version 3 of the Template Toolkit. For the new version, Andy is working on reorganizing the *Template::** module namespace, implementing virtual methods, refactoring the parser, and updating the template compiler. He's also taking feature requests, either through the TT3 mailing list (<http://www.template-toolkit.org/mailman/listinfo/tt3>) or the Kwiki (<http://www.template-toolkit.org/tt3/kwiki/ttkwiki.cgi>).

"The Template Toolkit is a fantastic tool" said Pierre Denis, Development Manager for Fotango, in a prepared statement. "It is fundamental to many of our projects, and with this funding the Template Toolkit will only get better."

Leon Brocard Becomes Pumpking

The 5.005xx maintenance branch of Perl has a new Pumpking: Leon Brocard, who helped to organize the first YAPC::Europe, and founded the Amsterdam.pm and Bath.pm Perl Monger groups. Currently an active member of London.pm, Brocard also maintains the official Perl Monger World Map at <http://www.astray.com/Bath.pm/>.

"My plan is to release a 5.5.4 soonish, which builds under modern operating systems and with modern libraries," Brocard wrote on the [perl.perl5.porters](http://perl.perl5.porters.newsgroup) newsgroup. "I really believe in Nicholas' view that it is a moral imperative to maintain old releases of Perl. I don't plan on changing much at all. Bug fixes will not be part of 5.5.4. Security fixes won't be part of 5.5.4. This is much like 5.6.2. Suggestions and offers of help welcome."

Coverage Testing with *Pod::Coverage* and *Devel::Cover*

Coverage testing lets you automatically find out what parts of your code are covered by tests or by documentation. It's an extension of automated testing that I've written about before. For someone who's releasing code to CPAN, documentation coverage is important for making sure that you have documented everything.

Code coverage lets you make sure that your test suite actually exercises all the options and paths through which the code can travel. I used it to find some functions that had never been tested because they were never actually used anywhere, much less in the test suite.

Starting with Documentation Coverage

The easiest way to think about coverage is by looking at documentation coverage. A simple rule to follow for your code is: "Every subroutine must have a block of POD that describes it." Subroutines that are documented are said to be covered, and those that aren't are uncovered or naked.

The following program, *pcover*, matches up POD sections to subroutines. It's pretty simple and works well for very regular, nontricky code in a single file.

```
#!/usr/bin/perl -w

use strict;
my %subs;
my %docs;

while ( <> ) {
    chomp;
    # Assume a =head1, =head2 or =item is the start
    # of some documentation.
    if ( /^=(head[12]|item [^d*]+\s+([^{()]+) ) {
        my $item = $2;
        $item =~ /[a-zA-Z0-9_+]/ or next;

        $docs{ $1 } = $.;
        next;
    }

    # Find subroutine declarations and stash the line
    # number where it appears.
    if ( /\s*sub\s+([a-zA-Z0-9_+]) / ) {
        my $sub = $1;
        $subs{ $sub } = $.;
        next;
    }
}

my $nerr = 0;
for my $sub ( sort keys %subs ) {
    if ( !$docs{ $sub } ) {
        print "The following subroutines have no docs:\n";
        if ++$nerr == 1;

        print "$sub, line $subs{$sub}\n";
    } # if
} # for
printf "%d sub%s found without docs.\n", $nerr, $nerr == 1 ? "" : "s";
```

Andy manages programmers for Follett Library Resources in McHenry, IL. In his spare time, he works on his CPAN modules and does technical writing and editing. Andy is also the maintainer of Test::Harness and can be contacted at andy@petdance.com.

```
$docs{ $1 } = $.;
next;
}

# Find subroutine declarations and stash the line
# number where it appears.
if ( /\s*sub\s+([a-zA-Z0-9_+]) / ) {
    my $sub = $1;
    $subs{ $sub } = $.;
    next;
}
} # while

my $nerr = 0;
for my $sub ( sort keys %subs ) {
    if ( !$docs{ $sub } ) {
        print "The following subroutines have no docs:\n";
        if ++$nerr == 1;

        print "$sub, line $subs{$sub}\n";
    } # if
} # for
printf "%d sub%s found without docs.\n", $nerr, $nerr == 1 ? "" : "s";
```

Now, when I run *pcover* against *WWW::Mechanize*,

```
$ pcover 'perldoc -l WWW::Mechanize'
```

or

```
$ pcover /usr/local/lib/perl5/site_perl/5.8.1/WWW/Mechanize.pm
```

I get the following:

```
The following subroutines have no docs:
_die, line 1397
_pop_page_stack, line 1350
_warn, line 1390
die, line 1380
```



```
res, line 808
warn, line 1370
6 subs found without docs.
```

(Note: If you haven't seen *perldoc -l Module::Name* before, start using it now. It prints out the full path of any module, so long as it contains some POD. It will save you much typing and hunting for module files.)

So I have six functions that aren't documented, at least according to my simple program. The first three, with the underscores, are for internal use only, so I don't mind. I have docs for *res()*, but the heuristic didn't find it. That leaves me only two that I actually need to document.

Pod::Coverage loads up the module into Perl, then takes a peek at the Perl internals

Aside from the problems described earlier, there's a big, unfixable problem with *pcover*. The heuristic for finding subroutine names relies on a very rudimentary parsing of Perl and doesn't handle the possibility that documentation for a package might be in a different module. Still, for simple coverage checking of simple modules, *pcover* may be all you need, and without having to install any new modules.

Pod::Coverage to the Rescue

For flexibility and accuracy, we turn to *Pod::Coverage*, which takes a different approach to handling the code. As Tom Christiansen said, "Nothing but Perl can parse Perl," so instead of trying to parse the contents of a source file itself, *Pod::Coverage* loads up the module into Perl, then takes a peek at the Perl internals. The documentation is still found with a simple set of heuristics, but they're far more flexible than *pcover*'s.

Pod::Coverage was cowritten and is maintained by the wily Richard Clamp, most notable for his *File::Find::Rule*. Like *File::Find::Rule*, *Pod::Coverage* is clever and flexible, with three different ways to use the module. The easiest is right from the command line:

```
$ perl -MPod::Coverage=WWW::Mechanize -e1
```

The *-M* flag tells the Perl executable to load *Pod::Coverage* as a module, passing *MARC::Record* as a parameter. The *-e1* is just a dummy program that does nothing, since *Pod::Coverage* does all its magic at load time.

When I run that command line, I get output similar to *pcover*:

```
WWW::Mechanize has a Pod::Coverage rating of 0.948717948717949
The following are uncovered: die, warn
```

The coverage rating of 0.948... means that 95 percent of my subroutines are documented. Note that *Pod::Coverage* took care

of the problems that *pcover* didn't handle: It ignored functions prepended with underscores, and it found the documentation for *res()*. As an author, I'm glad that *Pod::Coverage* found these because I realized that their differences from the core *die* and *warn* need to be noted.

Note that when we invoke *Pod::Coverage* this way, it expects that the module has already been installed on your system, and you can't pass a filename. If you've got a module you're working on, but haven't installed it on your system, you'll need to use a different approach. This command-line version also only works with a single module, not multiple modules in a distribution. For this extra flexibility, the *pod_cover* script, installed for you by *Pod::Coverage* as of Version 0.13, is a good place to start.

pod_cover assumes that you have a *lib/* directory that contains the modules you want to check, and checks them all for you. When I run this on my working directory for the *WWW-Mechanize* distribution, I find that at least the other module is covered:

```
Pod coverage analysis v1.00 (C) by Tels 2001.
Using Pod::Coverage v0.13

Sun Dec 28 22:14:52 2003 Starting analysis:

WWW::Mechanize has a doc coverage of 94.87%.
Uncovered routines are:
die
warn

WWW::Mechanize::Link has a doc coverage of 100%.

Summary:
sub routines total      : 47
sub routines covered    : 45
sub routines uncovered: 2
total coverage          : 95.74%
```

This is very convenient for a distribution with many modules. Just imagine having to maintain Dave Rolsky's *DateTime-TimeZone*!

Modifying the Rules

If the default settings for *pod_cover* don't fit your needs, you can quickly whip up a customized script. As part of the Phalanx project (<http://qa.perl.org/phalanx/>), I'm checking coverage of *Test::Reporter*, which has a few quirks. The documentation for it is kept in a *.pod* file, and there are a number of constant functions that need not be documented. Fortunately, the constant functions are all named with all capital letters, so it's easy to write a regular expression to match them. This little script gives me the accurate coverage results:

```
use Pod::Coverage;
use lib 'lib';

my $pc =
    Pod::Coverage->new(
        package => 'Test::Reporter',
        also_private => [ qr/^[A-Z_]+$/ ],
        pod_from => 'lib/Test/Reporter.pod',
    );
print "Coverage = ", $pc->coverage, "\n";
print "Uncovered: ", join( " ", $pc->uncovered ), "\n";
```

pod_from tells the constructor where to find the POD for the module, and *also_private* is a reference to an array of regular expressions that match subroutines that don't need to be documented. I only needed one *regex* to match the functions in question, but I could have passed as many as necessary.

Devel::Cover

Documentation coverage is helpful to ensure that no undocumented subroutines slip through the cracks. Code coverage is the measure of whether or not your tests are exercising all the parts of the code that they should.

Paul Johnson's *Devel::Cover* works by installing itself as a debugger hook, running your code, and then saving its metrics to a special database directory, called *cover_db* by default. Then, when you run the cover program, you'll be given two sets of output: a plain text summary of the coverage stats for each file and a set of HTML files that give in-depth, line-by-line code coverage analysis of your program. Plus, as an added bonus, if you have *Pod::Coverage* installed, *Devel::Cover* does documentation coverage analysis and includes it with the code coverage. Such a deal!

Before running your code coverage analysis, it's important to understand the four different types of code coverage that *Devel::Cover* tracks: subroutine, statement, branch, and condition. To illustrate, let's look at a simple piece of code and the tests for it.

Say I have a module, *My::Math*, with a subroutine, *my_sqrt*, that returns the square root of its parameter, but checks that it's not a negative number or undefined. If it is, it warns the user and returns 0:

```
1  =head2 my_sqrt( $n )
2
3  Returns the square root of I<$n>.  If I<$n> is not defined,
4  or is negative, return 0.
5
6  =cut
7
8  sub my_sqrt {
9      my $n = shift;
10
11     if ( !defined($n) || ($n < 0) ) {
12         warn "my_sqrt() got an invalid value\n";
13         $n = 0;
14     }
15
16     return sqrt($n);
17 }
```

Then, somewhere in my test suite, I have a *t/sqrt.t* that tests that the module works as advertised:

```
use Test::More tests=>2;

is( my_sqrt( 25 ), 5 );
is( int(my_sqrt(2) * 1000), 1414 );
```

Unfortunately, these two tests don't exercise as much of the code as they should. When I run the *t/sqrt.t* under *Devel::Cover*, the coverage percentages are pretty low:

File	stmt	branch	cond	sub
-----	-----	-----	-----	-----
sqrt.pl	60.00	50.00	33.33	100.00

Let's look at each of these different coverage categories.

Subroutine coverage is simple. There's only one subroutine and the tests executed it at least once, so my subroutine coverage is 100 percent.

Statement coverage is the measure of how many of the statements in the code have been run at least once. There are five statements in the example: lines 9, 11, 12, 13, and 16.

```
9      my $n = shift;
11     if ( !defined($n) || ($n < 0) ) {
12         warn "my_sqrt() got an invalid value\n";
```

```
13         $n = 0;
16     return sqrt($n);
```

Since I never pass an *undef* or negative to the subroutine, lines 12 and 13 never get executed in the test. Therefore, the statement coverage is only 3/5, or 60 percent.

Branch coverage measures whether each possible branch has been taken. Any *if* statement has exactly two possible branches, regardless of how complex the expression being evaluated is. My subroutine only has one branch, and only one of its possible branches gets taken. Therefore, my branch coverage is 1/2, or 50 percent.

Conditional coverage is related to branch coverage, but looks inside the contents of the conditionals for possible combinations of values. This is usually represented as a truth table and takes into account short-circuit Boolean evaluation.

For the expression:

```
!defined($n) || ($n < 0)
```

there are three combinations of values:

!defined(\$n)	\$n < 0
-----	-----
0	0
0	1
1	1

Since my tests only pass positive, defined numbers, the last row in the truth table is the only one that's been exercised. Therefore, my conditional coverage is only 33 percent.

Improving My Coverage

Now that we understand the different types of coverage, how can I improve them? I'll start with passing *undef*:

```
is( my_sqrt(undef), 0 );
```

which improves my results:

File	stmt	branch	cond	sub
-----	-----	-----	-----	-----
sqrt.pl	100.00	100.00	66.67	100.00

The main branch does get taken, which means that all the statements have now been executed, bringing statement and branch coverage up to 100 percent. However, I've only taken two of the three conditions in the truth table. To get to 100 percent conditional, I need to add a test for the negative number:

```
is( my_sqrt(-1), 0 );
```

Now, I have 100 percent coverage of all statements, all branches and all conditions:

File	stmt	branch	cond	sub
-----	-----	-----	-----	-----
sqrt.pl	100.00	100.00	100.00	100.00

Devel::Cover in the Real World

Of course, when you use *Devel::Cover* on your code, the results won't be quite so clear. One thing that will help is to make a script to automate the running of your code and generating the resulting HTML pages. I call mine *gocover*, and it works on any module because everything is relative to the current directory:

```
cover -delete
HARNESS_PERL_SWITCHES=-MDevel::Cover make test
```

```
cover
open ./cover_db/coverage.html
```

Note that this script is specifically for coverage testing on a standard module that uses *make test* to run its test suite. See the *Devel::Cover* documentation for examples of how to run other sorts of tests. The last line with the *open* command opens the main HTML file in my browser under Mac OS X. Adjust accordingly for your operating system.

Devel::Cover works by installing itself as a debugger hook, running your code, and then saving its metrics to a special database directory

When you run *gocover*, you'll notice things being significantly slower, taking about five to eight times as long to run because *Devel::Cover* is watching your program as it runs and collecting information along the way. Fortunately, the *cover* command runs quickly, generating a set of HTML files.

The first report file, *coverage.html*, is your thumbnail sketch of the files in your module, with coverage statistics for each of them. You'll notice a number of hyperlinks in the summary: Each filename and each percentage for branch, conditional, and subroutine coverage. Each of these links jumps to a specific subreport. All the reports are color coded—with green meaning something good and red meaning something bad—making it easy to skim the report for hotspots in your code.

The File Coverage page gives a full listing of a given file with metrics for each statement or subroutine as appropriate. The *stmt* column shows the number of times that the statement was executed, and the *sub* column shows how many times the subroutine was called. The *branch* and *cond* give percentages of coverage for the statement. The *time* column tells the number of milliseconds spent on each statement, which can help provide rudimentary profiling information.

Each of the Branch, Conditional, and Subroutine reports gives a summary of each branch, conditional, or subroutine, and summarizes the coverage for each. It's the same information as on the full File Coverage report, but condensed for easy skimming. You can also get to each of these summary reports from hyperlinks on the File Coverage page. Paul has made navigation between these pages very easy and helpful.

Improving your Coverage

So you've got a report on your module and you find that your coverage is weak. What do you do? First, remember that *Devel::Cover* is a tool, not an arbiter of correctness. It's up to you as the author to make the important decisions. That being said, it's a good idea to have all of your code exercised by your tests to make sure that everything works as you expect.

When I first ran coverage reports on *WWW::Mechanize*, I was disappointed to see how low my coverage numbers were. I'd been

proud of *WWW::Mechanize*'s testing suite, thinking it was fairly comprehensive, but *Devel::Cover* pointed out all the places I hadn't exercised. Most of the deficiencies fell into three areas.

The first, and the easiest to correct, was documentation coverage. I'd added a number of methods that I hadn't bothered to document, so fixing that was easy. I also found some methods that I considered as internal to my module, but that weren't prepended with underscores, so I renamed them.

Another source of the dreaded red boxes in my coverage reports were under-exercised default values. In a function that accepts default values, like a hypothetical text-writing function:

```
sub draw_string {
    my $str = shift;
    my $color = shift || "black";
    ...
}
```

the second statement has two different conditions that need to be tested: when a value for *\$color* is passed in and when it isn't. I found that in all my tests, I'd been taking the default values, but had never exercised the code that overrides them. A few extra tests took care of covering them.

Finally, I never tested my warning conditions to see if they actually generated warnings. For example, *Mech*'s *agent_alias* method lets you set your User-Agent string with an easy-to-read string like "Netscape 6," but if the alias isn't recognized, *agent_alias* emits a warning. Adding tests for this was simple with *Test::Warn*:

```
use Test::Warn;

my $m = WWW::Mechanize->new;
isa_ok( $m, 'WWW::Mechanize' );

warning_is (
    $m->agent_alias( "Blongo" );
) 'Unknown agent alias "Blongo"', 'Unknown alias squawks';
```

For checking behaviors where your code should die or throw exceptions, check out the *Test::Exception* module. Both should be in any module author's testing arsenal.

Pod::Coverage and *Devel::Cover* are both handy tools to help maintain your code, whether it's a module for distribution or just a simple web script. Even if you don't make your code run 100 percent under the tools, take the time to try it once. I guarantee you'll find at least one surprising thing in your code.

TPJ



CPAN in the Desert

brian d foy is currently on active duty with the United States Army as a military policeman in Iraq. He says you can e-mail him at comdog@panix.com, if you don't mind waiting a couple months for a reply.

—Ed.

I am in the middle of the Iraqi desert, and I am patching modules, or at least I am trying to. Adam Turoff sent me a CPAN snapshot that is about two weeks old, since snail mail takes that long to get here. I am copying it from the three CDs (that just barely contain it) to my hard drive (which also has just enough room for it) so I can then use it with the cpan program that comes with CPAN.pm, which comes with Perl.

The cpan program is one of the most useful inventions for the Perl programmer. To install almost any module, I simply tell it the name of the module to install, and it downloads and installs it. If that module relies on other modules I do not have, or even newer versions of modules I already have, it takes care of that, too, recursively. If you cannot find the cpan program on your computer, you can cheat with this one-liner that does the same thing:

```
% perl -MCPAN -e shell
```

The `-M` switch tells Perl to use the CPAN.pm module, and the `-e` switch gives Perl the program on the command line. In this case, `shell` is a function that CPAN.pm exports. The `shell()` function simply starts CPAN.pm's interactive shell. When you use it for the first time, it should take you through a configuration process that sets your personal preferences, which show up in the file `$HOME/.cpan/CPAN/MyConfig.pm`, which CPAN.pm looks at after the site-wide settings in `CPAN/Config.pm`.

When I was back home enjoying my real life, I had a constantly connected cable modem or a big pipe at work. I configured the cpan program the first time that I used it, when it prompted me and hinted at which values I should use. I never thought of it again. Part of the configuration was a selection of CPAN sites to use for download servers.

brian has been a Perl user since 1994. He is founder of the first Perl Users Group, NY.pm, and Perl Mongers, the Perl advocacy organization. He has been teaching Perl through Stonehenge Consulting for the past five years, and has been a featured speaker at The Perl Conference, Perl University, YAPC, COMDEX, and Builder.com.

Unfortunately, in the desert, I have no network connection, so all of those servers are worthless to me. I do have all of a fairly recent CPAN installed on my laptop, though. I need to configure CPAN.pm to use that rather than anything else.

I start my interactive CPAN.pm session by running the cpan program without arguments:

```
brian$ cpan

cpan shell - CPAN exploration and modules installation (v1.63)
ReadLine support enabled

cpan>
```

Since I have just started working with Perl again, being busy with other things out here, I have forgotten how a lot of things work. I, like Matthew Broderick in *War Games*, start by asking for help:

```
cpan> help
```

The program displays a plethora of options and commands. I basically know that I want CPAN.pm to look on my computer and not on the network, and that the list of servers is part of the configuration. From the help message, I print the configuration options and settings. I leave out most of the output, and leave in only the settings for the `urllist` parameter that I want to affect:

```
cpan> o conf

urllist
file:///Users/brian/MINICPAN/
http://ftp.secl.org/pub/mirrors/CPAN/
http://www.perl.com/CPAN/
```

The first URL in the list is my mini-CPAN repository, created by Randal Schwartz's minicpan program (<http://www.stonehenge.com/merlyn/LinuxMag/col42.html>). It includes just the latest modules from CPAN and is about one-fifth the size of CPAN. My mini-CPAN, however, is nine months old and does not have the latest versions of the things I want to play with, and rather than use that, I want to use the CPAN snapshot that Adam sent to me.

To change the `urllist` parameter, which is really just a list, I use Perl's CPAN.pm list commands to change it. In this case, I want

101 Perl Articles!



From the pages of *The Perl Journal*, *Dr. Dobbs's Journal*, *Web Techniques*, *Webreview.com*, and *Byte.com*, we've brought together 101 articles written by the world's leading experts on Perl programming. Including everything from programming tricks and techniques, to utilities ranging from web site searching and embedding dynamic images, this unique collection of *101 Perl Articles* has something for every Perl programmer.

Plus, this collection of articles is fully searchable, and includes a cross-platform search engine so you can immediately find answers you're looking for. Delivered as HTML files in a ZIP archive or CD-ROM image, download *101 Perl Articles* and burn your own CD-ROM or store it on hard disk.

\$9.95 For subscribers to
The Perl Journal

\$12.95 For nonsubscribers to
The Perl Journal

\$24.95 To subscribe to
The Perl Journal and
receive *101 Perl Articles*

Go to

<http://www.tpj.com/>

now!

to add my local CPAN repository to the front of the list so CPAN.pm checks it first:

```
cpan> o conf urllist unshift file:///Users/brian/CPAN/
```

The CPAN.pm module expects the URLs to conform to RFC 1738, although it is a bit liberal in what it actually accepts. To me, this means that my file:// scheme URLs that end with a directory name have a trailing slash.

I check the *urllist* parameter again to make sure that I added the new URL:

```
cpan> o conf urllist
urllist
file:///Users/brian/CPAN/
file:///Users/brian/MINICPAN/
http://ftp.secl.org/pub/mirrors/CPAN/
http://www.perl.com/CPAN/
```

When I am satisfied with my changes, I commit the changes to disk and these changes will apply to future interactive sessions. I do not have to do this if I only want to affect the current situation, which I might like to do if I am traveling or at a temporary location.

```
cpan> o conf commit
commit: wrote /Users/brian/.cpan/CPAN/MyConfig.pm
```

Now, when I use the cpan script, CPAN.pm looks for modules and metadata on my computer rather than the network. I can also use my local CPAN snapshot to update my mini-CPAN, which is much smaller than all of CPAN. The minicpan script uses CPAN.pm to figure out where it should look, and since I just configured CPAN.pm to look on my local computer, that is where minicpan looks.

The Perl Review (http://www.theperlreview.com/at_a_glance.shtml) keeps track of the size of CPAN and the mini-CPAN to measure the Schwartz Factor. As of December 1, that number was 0.2, meaning that the mini-CPAN was about one-fifth the size of the entire CPAN snapshot. Once I update my mini-CPAN, I can get rid of the big snapshot, which is getting close to 2 gigabytes, but then I need to ensure that I get rid of it in my *urllist* for CPAN.pm. If I keep it in the list, CPAN.pm will try to use that directory before it fails and goes on to the next one in the list, which is slow enough to be annoying.

As before, I use a Perlsh command in the CPAN.pm shell. I take off the first item in the list with *shift*, then check to see that it worked. Once I have it right, I save the changes:

```
cpan> o conf urllist shift

cpan> o conf urllist
urllist
file:///Users/brian/MINICPAN/
http://ftp.secl.org/pub/mirrors/CPAN/
http://www.perl.com/CPAN/

cpan> o conf commit
commit: wrote /Users/brian/.cpan/CPAN/MyConfig.pm
```

Now that I have my mini-CPAN updated, and I have CPAN.pm configured to use it, I can install modules as if I were at home and connected to the Internet. I have all the latest versions, and only that in the mini-CPAN, so it does not take up a lot of space on my computer.

TPJ

Tracking TV Shows with Palm and Perl

I find it quaint, bordering on humorous, hearing North Americans complain about television. If you want to experience truly paleolithic TV, come to Australia, where series are picked up years late, episodes are seldom rerun, and shows disappear for nine or more months at a time, only to return unexpectedly in a different time slot. And that's for the popular shows. The series I like to watch (i.e., geek shows) I'm lucky to find at all. I still can't hear the final credits for *Stargate SG-1* without dreading the inevitable voiceover announcing that That Was The Final Episode For Now, and that *Stargate* Will Return Later This Century.

So I did the natural geek thing and found episode listings at <http://www.epguides.com/> for the series I cared about, printed them out, and kept them in a binder under the coffee table. It was a great reference because not only did it give cast lists and episode titles, but it also listed air dates, so in my more morbid moments I could, for instance, look up the fact that *Deep Space Nine* finished airing on June 2, 1999, yet it's still going here, assuming it ever reappears.

Meanwhile, about a year ago in a desperate bid to get organized, I got a Palm PDA. One of the first things I did was buy a database program for it—thinkDB (since bought out by DataViz and rebranded as SmartList To Go)—to catalog my collections.

It's probably taken you seconds to make the connection, but I didn't figure it out until recently: Why not use the Palm to keep track of the TV shows?

The Plan

Manual entry was out of the question: Between my partner and me, there are about 20 shows we care about, and most of them are of the *Gunsmoke* kind and hundreds of episodes long. Besides, I'd still have to visit [epguides.com](http://www.epguides.com) every so often to update the lists. What I really wanted was a program that would write the database for me, and update it whenever I felt the need.

Needless to say, this program had to be written in Perl. Not because Perl is inherently a better language, but because all the necessary modules already existed in Perl. This was more important than usual, because Palm databases are messy to work with.

Debbie teaches Perl and assembly at Monash University in Australia. She can be reached at debbiep@csse.monash.edu.au.

Palm databases deserve a bit of explanation because they are quite foreign compared with normal files. First, all files on a Palm (even programs) are databases. Second, Palms are historically stingy when it comes to memory capacity. Third, although databases are composed of records, database records don't have any predefined internal structure, such as fields; that's up to the application. Together, these facts mean that every Palm application has its own unique, compact, and incomprehensible record format. This goes double for thinkDB, which tries to implement its own unique, compact, and incomprehensible database format on top of Palm's built-in one.

Luckily, there is a Perl module for manipulating these databases: *Palm::PDB*. This module reads a PDB file (a flat-file image of a Palm database) and breaks it up into more obvious pieces (doing things like putting its records into an array). If you have a helper class, which is a subclass of *Palm::PDB* with some required methods, then you might even be able to get hashes containing some internal record structures like fields. Helper classes exist for the Palm PIM databases such as *Palm::Address*, *Palm::Memo*, *Palm::ToDo*, and, fortunately for me, *Palm::ThinkDB*.

With *Palm::ThinkDB*, I can open up a database file (obtained from a prior HotSync or using the handy program pilot-xfer), rummage around in its records, add data to fields or even add entire records, and save the resulting database back to a new file (which I would subsequently have to install back onto the Palm using pilot-xfer or another HotSync). Everything in between is merely screen scraping.

Implementation

The first incarnation of my program, available at <http://www.csse.monash.edu.au/~debbiep/palm/update-episodes/>, did exactly that. The aforementioned screen-scraping consists of downloading web pages from [epguides.com](http://www.epguides.com) and parsing them to obtain series and episode names, cast lists, airdates, numbers, and (I was feeling ambitious) episode synopses. It's quite a straightforward use of the *LWP::UserAgent/HTML::Parser* combination. Ultimately, I was left with a database file that I could then send back to my Palm.

Actually, not one database file, but two. Keeping track of TV series and episodes is a two-level task because each series can have many episodes. In relational-database talk, there's a one-to-many

relationship between the Series table and the Episode table. A thinkDB database can link to another using a foreign key, which is how an episode knows which series it belongs to. An example is presented in Figure 1. Each series has a scheme, which says how to go about obtaining episode details, and a URL, which is used by the scheme to locate series information. They also have some long text fields, one for a list of the show's regular cast (if it can be found) and one to satisfy any copyright or terms of use that the web site might have (Epguides, for instance, says that you have to retain the page's copyright notice). The series' IDs are assigned automatically by thinkDB. Episodes have titles, airdates, production numbers used by the studio, sequence numbers (to ensure that the episodes can be ordered properly), the season they belong to, and episode numbers within that season. They also have synopses, if the series is marked as wanting them, and checksums, which are MD5 digests of all of the other episode fields concatenated together; it's a quick way of determining when an episode's details have changed and need updating. If either the series or episode databases have extra fields, the program leaves them untouched. In this way, I can mark whether I've seen an episode, or give it a numeric rating or a note.

The handler for the scheme I factored out into its own module to facilitate development of new ones if, say, Epguides shuts down or blocks robots or doesn't list a series I care about. The code for the Epguides handler is kept in `Scheme/Epguides.pm`. Most of it is pretty mundane. Perhaps the most interesting part is `guessSeriesURL` (line 80 onward), which tries to find the start URL for a series given its name. This is tricky because you want *The Practice* to match just "Practice," and "Practice, The". The approach I take is to strip the words "and," "an," "a," and "the," all punctuation and spaces and capitalization before looking for one being a substring in the other. It gets a few false positives (several hundred for *ER*!) but usually it's spot on.

The HTML parsing code in `Scheme/Epguides.pm` is a fairly typical use of `HTML::Parser`. It is just robust enough to handle the brownian motion in the HTML format used by Epguides. The `parseEpisodes` function (line 552 onward) doesn't use `HTML::Parser` because episode titles are encased in a `<pre>` section and a regular expression offers finer control.

All of the grunt work of manipulating the database is in the update-episodes program, which is available in full at <http://www.tpj.com/source/>. The real business starts at line 143, where the series database is loaded. (See Listing 1; assume `$conduit == 0` for the moment, so we skip right to the `else` statement.) Each series is sent through the `processSeriesDBRecord` mill, and the re-

sulting database is written out to a new file. The corresponding code for the episode database starts from line 196 (see Listing 2). First, existing episodes are processed, and then any new episodes that might have been found are appended to the database (line 215). Finally, the episode database, too, is written back to a new file.

The `processSeriesDBInfo` function mostly performs some sanity checks to ensure that all the fields the program needs are present in the database. It also sets some mappings so that the rest of

*Because each application
has its own database record
format, each application needs its
own conduit, too*

the program knows that, for instance, the "Cast" field is column number 6 in the thinkDB database. This function is also where the handlers for schemes are loaded. `processEpisodeDBInfo` is analogous to the episode database.

`processSeriesDBRecord`'s job is to update the fields for one series. Mostly, it's a matter of inserting whatever the scheme handler methods returned into the record's fields, and remembering the episode data for when it's time to process the episode database. Most of the function is devoted to dealing with the situation where the user has added a new series to the database on the Palm, and not specified a scheme or URL. In this case, update-episodes prompts for appropriate values, asking the scheme handler module to guess the URL, and stores the user's responses in the record for future reference.

`processEpisodeDBRecord` updates information about an episode. By now, all of the web accesses have been done, so this function merely has to compare the checksum stored in the episode's record with the checksum computed earlier (line 463), and update the record's fields if any episode data has changed.

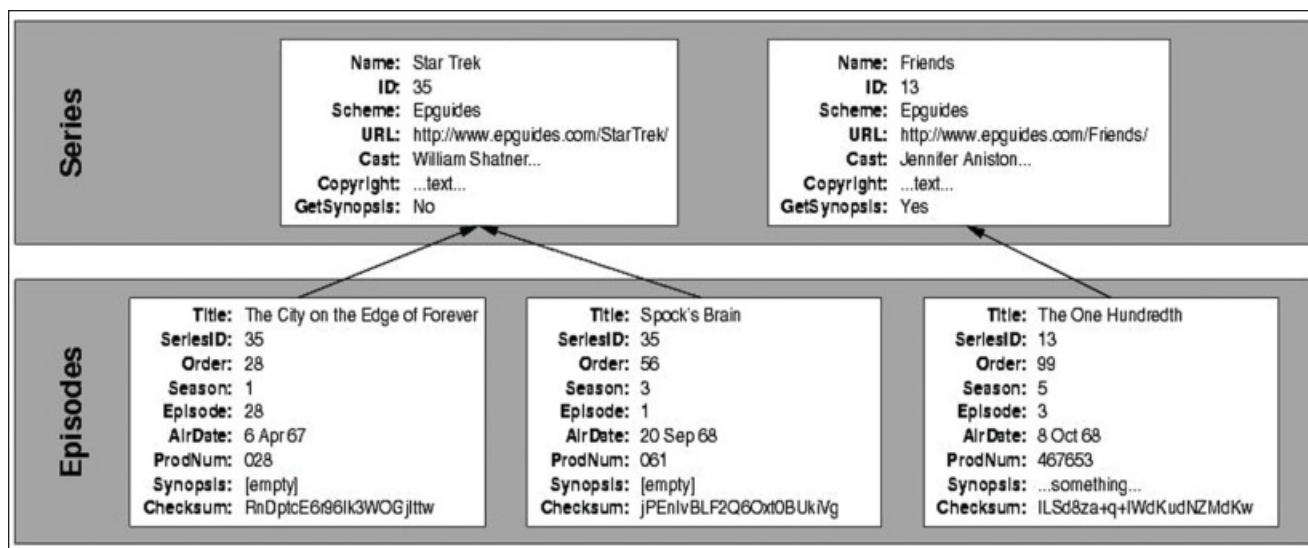


Figure 1: Database structure.

The Result

The screenshots in Figures 2–5 show the series listing, one series in detail, an episode listing, and one episode in detail. The exact appearance of these screens is up to the user, since update-episodes doesn't meddle with the list and form layouts.

Automating

If you're a Palm user, you're probably aghast at the thought of manually having to fetch two files from the Palm, run update-episodes, and install the new files back to the Palm. Palm users have always been able to put their PDA in the cradle and simply press the HotSync button, and have everything on both the palmtop and desktop computers automatically update through the magic of conduits.

A conduit is a program that runs on the desktop. It knows how to communicate with the Palm during a HotSync and it knows how

to update a Palm database (and the corresponding desktop database) so that they are synchronized. Because each application has its own database record format, each application needs its own conduit, too.

While conduits run on the desktop, they are able to instruct the Palm to perform some simple database operations, such as sending a database record to the desktop for analysis, or deleting a record. It's an idea similar in spirit to Remote Procedure Calls (RPC), though the protocol (in two layers, called DLP and SPC) differs in its details. PalmSource (the company) provides DLP APIs for conduits in C, C++, Java, and COM, but any language that can talk to a serial port could do the same. Naturally, someone has already written a Perl API and released it as part of a package called, amusingly, ColdSync.

ColdSync offers several ways of developing conduits, depending on how complex your needs are. At the simplest level, you



Figure 2: Series listing.

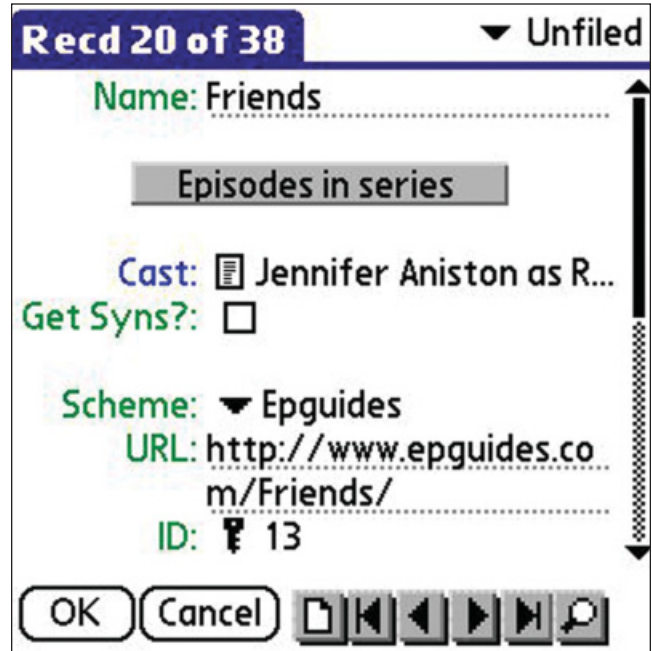


Figure 3: One series in detail.



Figure 4: An episode listing.

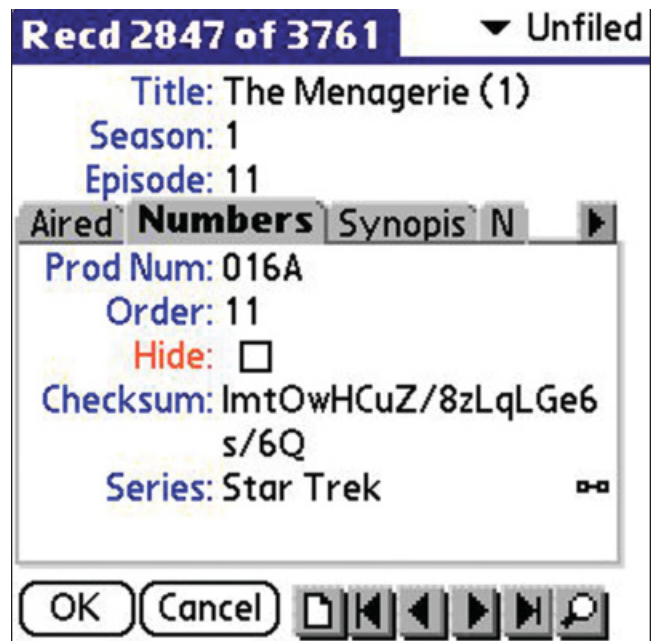


Figure 5: One episode in detail.

don't even need to know any of the DLP API, and you can just work on PRC files with *Palm::PDB* and a helper class, as I've already done. This won't work for update-episodes for two reasons: First, two databases are involved, and the ColdSync generic conduit can work with only one database at a time; second, this is an asymmetric sync because most data needs to be overwritten by the desktop (with its Internet connection) and some data needs to be retained as the Palm currently has (such as the "Seen" field). For a conduit this complex, the only solution is to use the DLP API, the most important functions of which are found in the *ColdSync::SPC* module.

Fortunately, the current CVS builds of ColdSync come with a new module, *ColdSync::PDB*, which wraps up DLP calls so that I can (mostly) use the *Palm::PDB* and *Palm::ThinkDB* methods that I'm already familiar with. This is what I use in update-episodes. (A note of warning: I had to apply a small patch to *ColdSync::PDB* to make this work. It may be that the final released *ColdSync::PDB* API differs, so check <http://www.csse.monash.edu.au/~debbiep/palm/update-episodes/> for news.)

When ColdSync invokes a conduit, it sets *\$ARGV[0]* to "conduit"—update-episodes uses this to determine whether to run as

a conduit or in standalone mode. The conduit-aware code differs only in small ways from the standalone version, as you can see by comparing the *if/else* blocks in update-episodes' main code. Mostly, it comes down to a slightly different API, but the other difference is that a conduit is running while connected to the Palm, so that some things that the standalone code must do in a big chunk (such as writing all episode records to the output file) can be done piecemeal by the conduit (which is why update-episodes needs to know if an episode has changed since the last sync; unchanged records don't need to be written again).

So now, once a week or so, I can fire up ColdSync on my desktop, press the HotSync button on my Palm's cradle, and sit back and watch all of my TV series update. For the 38 series and 3700-odd episodes currently in my database, this takes a couple of minutes. With the time I saved, I ought to be able to catch up on more TV shows, assuming, that is, that I can find any of them on Australian TV stations.

TPJ

(Listings are also available online at <http://www.tpj.com/source/>.)

Listing 1

```
107 # Work on the series database.
108 {
109     if ($conduit)
110     {
111         my $seriesDBH = ColdSync::PDB->Load($seriesDBName, "rwx");
112         if (!defined $seriesDBH)
113         {
114             die "500 Can't load series DB";
115         }
116
117         # Probably shouldn't do this, but how else?
118         my $seriesDB = $seriesDBH->{helper};
119
120         # Load all of the database's records into helper class.
121         # Normally this'd be pretty inefficient, but we have to
122         # load them all anyway, so let's get it over with.
123         my @records = $seriesDBH->records();
124
125         # Go through the series database field names and determine which is
126         # which.
127         processSeriesDBInfo($seriesDB);
128
129         # Process each series record and write it back.
130         foreach my $series (@{$seriesDB->{db_records}})
131         {
132             processSeriesDBRecord($series);
133             # Always write the new record.
134             $seriesDBH->writeRecord($series);
135         }
136
137         # Undefine the database handle to close it.
138         undef $seriesDBH;
139         undef $seriesDB;
140     }
141     else
142     {
143         my $seriesDB = Palm::PDB->new();
144         $seriesDB->Load($seriesDBName, ".pdb");
145
146         # Go through the series database field names and determine which is
147         # which.
148         processSeriesDBInfo($seriesDB);
149
150         foreach my $series (@{$seriesDB->{db_records}})
151         {
152             processSeriesDBRecord($series);
153         }
154
155         # Write the episode database, now finished with it.
156         $seriesDB->Write("new_" . $seriesDBName . ".pdb");
157         undef $seriesDB;
158     }
159 }
```

Listing 2

```
162 # Now load the episode database.
163 {
164     my $episodeDBH;
165     my $episodeDB;
166
167     if ($conduit)
168     {
169         $episodeDBH = ColdSync::PDB->Load($episodeDBName, "rwx");
170         if (!defined $episodeDBH)
171         {
172             die "500 Can't load episode DB";
173         }
174
175         # Probably shouldn't do this, but how else?
176         $episodeDB = $episodeDBH->{helper};
177
178         # Load all of the database's records into helper class.
179         my @records = $episodeDBH->records();
180
181         # Go through the series database field names and determine which is
182         # which.
183         processEpisodeDBInfo($episodeDB);
184
185         # Process each episode record and write it back.
186         foreach my $episode (@{$episodeDB->{db_records}})
187         {
188             if (processEpisodeDBRecord($episode))
189             {
190                 # episode has changed, so write the record.
191                 $episodeDBH->writeRecord($episode);
192             }
193         }
194     }
195     else
196     {
197         $episodeDB = Palm::PDB->new();
198         $episodeDB->Load($episodeDBName, ".pdb");
199
200         # Go through the episode database field names and determine which is
201         # which.
202         processEpisodeDBInfo($episodeDB);
203
204         # Process known episodes first.
205         foreach my $episode (@{$episodeDB->{db_records}})
206         {
207             # Don't care about whether record has changed, so ignore return
208             # value.
209             processEpisodeDBRecord($episode);
210         }
211     }
```

TPJ



Lessons Learned Converting Java to Perl

Simon Cozens

With all the horror stories I've heard over the past few years of Perl projects being packed up and replaced wholesale with Java projects, I recently had the happy opportunity to get back in some small way.

Jakarta Lucene is a Java-based framework for embedding search engines into an application. It provides a simple search engine with analyzers, index writers, index readers, an optimizer, a query parser, and several query processors and scorers. Lucene is steadily being ported to other languages; *Dr Dobb's Journal* recently reported on a C# version, and lupy, the Python version, and ruby-lucene are both in the works.

An application we were working on needed a search engine, and Lucene looked like the best of breed, so we decided to use it in conjunction with the *Inline::Java* module to glue the Perl and Java parts together. However, there were certain problems with this approach.

First, it was extremely complicated—the Java-to-Perl bridge wasn't ideally suited to being used for multiple users and concurrent access. And it was too complicated in terms of architecture—it just didn't feel like a neat design.

Also, we wanted to be able to extend the search in arbitrary ways, including having the ability to dive into the index and pick out indexed terms, and so on. We couldn't really do this in Java as flexibly as we'd liked, not least because only few of us knew enough Java.

But we knew a lot of Perl, and hey, it's only code. I took one and a half man-months to attack the 13,790 lines of Java code in Lucene 1.2, and produced Plucene. It's not quite ready for prime-time at the time of writing, so don't ask me for it yet, but it's rapidly getting there.

However, you can't work for a month on something absolutely new without learning some lessons, can you? So this month, I hope to share with you some of the lessons I've learned over the past month as part of this Java-to-Perl translation project.

Estimating the Job

The first lesson has absolutely nothing to do with the specific technology but everything to do with project management. The con-

version took much longer than I was anticipating, and that's because my estimation was completely off.

When you're converting code, it isn't appropriate to just try converting a few files and making an estimate based on how long that took and how much a percentage it was of the total source. In this case, I started by converting the textual analysis classes and completed about 10 or 15 in a morning. However, these were very simple ancillary classes, many of which abstract classes override in only one or two methods.

When I began messing with the two index writer classes, I found that each one was going to take at least one day to fully understand, and another day to code up. Suddenly, my estimates were laughable.

So the first lesson is, if nothing else, have an understanding of the project as a gestalt before making any estimates. Simply chipping away at a corner of it and then extrapolating will put you in danger of racing through the simple cases and becoming stuck on the actual meat of the project.

There's a particular hacker fallacy that says you should spend your hacking time hacking, since that's what you're good at. That's very often the best solution if the problem is clearly defined, but in my case, I would have benefited from stepping back and taking two days to really understand the intricacies of the task ahead. It's easy to see two days like that as wasted, but time spent planning should not be seen as wasted, but as an investment.

It seems so easy when glibly put like that, but there is an undeniable "urge to hack." If nothing else, thinking time gives you nothing to show to your boss, while hacking does. But on the other hand, as Brian Kernighan and Rob Pike put it in their *Practice of Programming* (Addison-Wesley, 1999): "Resist the urge to start typing; thinking is a worthwhile alternative."

Another important concept to remember when translating existing code is that the vast majority of the code that's there is there for a reason. Our initial port of Lucene was going to be "just enough" to work, and so I estimated that we wouldn't need to port about 20 percent of the Java. But while there were some classes that could be left alone for the moment—for instance, Lucene allows queries that specify that one word should appear "near" another word, but that's not critical to its functionality—most of the classes were there because they were actually useful.

This is another one of those things that should not be a surprise. People don't put code into a project for the fun of it. They put it there because it's used by other code. But it's sometimes

Simon is a freelance programmer and author, whose titles include Beginning Perl (Wrox Press, 2000) and Extending and Embedding Perl (Manning Publications, 2002). He's the creator of over 30 CPAN modules and a former Parrot pumping. Simon can be reached at simon-cozens.org.

tempting to account for pieces of code that we “don’t need to do yet.” You do need to do them, and if you’d spent a couple of days analyzing in advance, you’d know this.

Use Available Tools

How do we do our analysis? Well, there are almost always useful tools to do some kind of static analysis for us. In the case of Java, I picked up the lovely JAnalyzer (<http://www.bodden.de/projects/janalyzer/>), which can perform static analysis and tell you where methods are being called and which methods they in turn call.

*The conversion took
much longer than I was
anticipating, and that's because
my estimation was completely off*

This was particularly useful when I had to do something about the Java tendency toward method overloading. For instance, we have the two methods:

```
public void seek(Term term) throws IOException { ... };
void seek(TermInfo ti) throws IOException { ... };
```

Both are called *seek*, and which one gets called depends on the type of the argument. Of course, there’s a naturally Perlsh way to do this:

```
sub seek {
    my ($self, $t) = @_;
    if ($t->isa("Plucene::Index::Term")) {
        # seek version 1
    } else {
        # seek version 2
    }
}
```

However, this suffers from muddled thinking—it’s certainly not the Perl way to have a subroutine do one thing if it’s called with one type of argument and a completely different thing if it’s called with another type. Since Java has this kind of method overloading built into the language, it’s much more natural to see it in Java; but Perl does not, and so it is not.

Instead, the best way to do this is to identify what’s really going on—the *TermInfo* version does all the work, and the *Term* one is a front end that turns the *Term* into a *TermInfo*. So we’ll call one *seek* and the other *seek_ti*. Now we need to work out where the two different methods are actually called, and rename appropriately; this is where our analyzer comes in.

With a decent set of analysis tools, this is a simple process—click on the method, you get a list of places where it was called, and you track them down in your ported version. Without analysis tools, it’s down to grep, checking the context of each returned line, and painstakingly looking through each one. It’s worth taking the time out to see what tools are available to play with the code.

The Joy of Tests

Another thing that held me back and could have been done better was the identification of distinct subprojects. Once you’ve identified the major components of the program, you can treat them as individual, isolated parts, port across the relevant files, test, rinse, and repeat.

Did I mention tests? Tests are your friend. Really, they are. It took me many years to realize this, but tests are not just a tedious thing you do after writing the code to make it look professional. Once you’ve properly componentized your task, you can use unit tests to ensure each component is doing what you think it ought to be doing.

I’m not one of those people who believes that you should write your tests first, watch them fail, and then build your code until they pass; and yes, I have heard all the arguments for it, thank you very much. However, I have far too many moments of enlightenment just after staring at the code and just prior to uttering, “Wait, does this actually do anything right at all?”

That’s where unit tests come in, and there’s been a lot of work put in to make unit tests really quite easy in Perl. My favorite testing module is *Test::More*, which provides, among others, the *ok*, *is*, *isa_ok*, and *is_deeply* routines. For instance, here’s a portion of Plucene’s test suite:

```
my $size = -s DIRECTORY . "/words.tis";
ok($size, "Wrote index of $size bytes");
```

First, we check that the index writer produced a nonzero sized index. *ok* takes an argument and prints “ok” if it is a True value and “not ok” if it is not. It also, like all the other *Test::More* routines, takes an optional comment to identify the test.

```
my $reader = Plucene::Index::TermInfosReader->new(DIRECTORY, "words", $fis);
isa_ok($reader, "Plucene::Index::TermInfosReader", "Got reader");
my $enum = $reader->terms;
isa_ok($enum, "Plucene::Index::SegmentTermEnum", "Got term enum");
```

isa_ok is used to ensure that a value is the type we expect it to be.

```
for my $i (0 .. $#keys) {
    $enum->next;
    my $key = $keys[$i];
    is_deeply($enum->term, $key, "Key $i matches");
```

is_deeply compares two structures recursively, reporting on where they differ.

```
my $ti = $enum->term_info;
is($ti->doc_freq, $doc_freqs[$i], "Doc frequency at $i matches");
}
```

And *is* compares two scalar values, reporting a difference. That’s essentially all there is to testing in Perl, so unfortunately, there’s hardly any excuse for not doing it.

Asserting Your Rights

Even once you’ve got all your code ported across and your unit tests in place, there will be bugs. You can’t avoid it. And these will not be friendly bugs, which are easy to diagnose. They will be bugs you don’t understand, that will take you a day to work out where they’re coming from. They will be bugs that manifest themselves somewhere completely different in the program, and say things like:

```
Can't take log of 0 at blib/lib/Plucene/Search/Similarity.pm line 61
```

And that means that you didn’t pass in the appropriate parameter to a method 10 frames up the call stack. Of course.

How are we supposed to know this? Because when we find something like this, where there's obviously a parameter gone adrift somewhere, we take the relevant subroutine:

```
sub idf {
    my ($self, $tf, $docs) = @_;
    my ($x, $y) = ($docs->doc_freq($tf), $docs->max_doc);
    return 1 + log($y / (1 + $x));
}
```

and just before the failing line, we inject the following code:

```
use Carp qw(confess);
confess("No documents for that term?")
    unless $x;
```

or some similarly informational message. This time, instead of a single cryptic error message, you'll get something like:

```
No documents for that term? at Plucene/Search/Similarity.pm line 62
Plucene::Search::Similarity::idf('Plucene::Search::Similarity', 'Plucene::Index::Term=HASH(0x942054)', 'Plucene::Search::IndexSearcher=HASH(0x940890)') called at Plucene/Search/TermQuery.pm line 64
Plucene::Search::TermQuery::sum_squared_weights('Plucene::Search::TermQuery=HASH(0x9423c0)', 'Plucene::Search::IndexSearcher=HASH(0x940890)') called at Plucene/Search/Query.pm line 78
Plucene::Search::Query::scorer('Plucene::Search::Query', 'Plucene::Search::TermQuery=HASH(0x9423c0)', 'Plucene::Search::IndexSearcher=HASH(0x940890)', 'Plucene::Index::SegmentsReader=HASH(0x93d6d0)') called at Plucene/Search/IndexSearcher.pm line 138
Plucene::Search::IndexSearcher::_search_hc('Plucene::Search::IndexSearcher=HASH(0x940890)', 'Plucene::Search::TermQuery=HASH(0x9423c0)', 'undef', 'Plucene::Search::HitCollector=HASH(0x8cfd84)') called at Plucene/Search/Searcher.pm line 67
Plucene::Search::Searcher::search_hc('Plucene::Search::IndexSearcher=HASH(0x940890)', 'Plucene::Search::TermQuery=HASH(0x9423c0)', 'Plucene::Search::HitCollector=HASH(0x8cfd84)') called at Plucene/Simple.pm line 114
```

Now we know what we're doing and how we got to where we are. This saves us a lot of tedious tracing through the program and trying to find out where it's getting itself in a knot. Because it shows us the arguments to each subroutine, sometimes this trace is enough to spot a stray *undef* or wrongly typed parameter. Other times, you need to crawl through the values of the arguments; *Data::Dumper* is an excellent way to do this.

In this case, temporarily adding in:

```
use Data::Dumper;
print Dumper($docs);
```

would show me that there's something wrong with the data in the *IndexSearcher*.

The key point here, though, is that once you've worked out what the bug is, and you've written a handy test case to stop it from coming back again, you don't necessarily have to remove your *confess* assertions. They'll be helpful for catching similar bugs and things that shouldn't be able to happen in the future.

One particularly good way to turn your bug tracing into assertions is to use the *Carp::Assert* module. This provides a number of functions, the most useful being *assert*. For instance, given this code, to read a string from a network socket:

```
my $length = read_string_length($socket);
my $string = " " x $length;
$socket->read($string, $length);
```

You could ensure that the first thing read, the string's length, is a sensible value, like so:

```
my $length = read_string_length($socket);
assert($length >= 0);
```

```
my $string = " " x $length;
$socket->read($string, $length);
```

By peppering your code with these assertions, you can be confident that your data is what you think it should be at each stage of your program's operation. If the length returned is negative, you'll get an error, and also a stack trace just like the one we saw earlier. But, surely, it takes up a lot of time to constantly check these assertions, and what happens when you want to go into production?

It took me many years to realize this, but tests are not just a tedious thing you do after writing the code to make it look professional

Carp::Assert also provides the symbolic constant *DEBUG*, which it sets to 1 on import. This allows you to say:

```
my $length = read_string_length($socket);
assert($length >= 0) if DEBUG;
```

and the condition will be tested just like before. However, when you want to go into production and need to get rid of these assertions, just change *use Carp::Assert* to *no Carp::Assert*. This sets the *DEBUG* constant to 0, and Perl is smart enough to know that code followed by 0 never needs to run and optimizes it away.

This is particularly useful for testing the interfaces to internal API functions in the absence of strict type checking. In Java, you can declare that a subroutine takes a *Plucene::Index::Reader* and the compiler can tell at compile time if you've passed it a value that's not going to be a *Plucene::Index::Reader*.

In Perl, however, variables can contain any kind of scalar, so they can't easily be type checked at compile time. However, we can use *Carp::Assert* to check them at runtime, which is the next best thing, and saves even more obscure errors later:

```
sub add {
    my ($self, $reader) = shift;
    assert($reader->isa("Plucene::Index::Reader"));
    ...
}
```

Mind Your Interfaces

Why is this important? The final lesson to learn is that interface consistency is a massive help to avoiding bugs in a large project. For instance, let's consider two things: first, styles of passing parameters. Java and the C-related languages have only one style: You pass a list of typed parameters in a defined order:


```
public IndexWriter(String path, Analyzer a, boolean create)
```

But Perl has several different styles that are in common use. There's the C-like style:

```
IndexWriter->new($path, $analyzer, $create)
```

Or there's the named parameter style:

```
IndexWriter->new(path => $p, analyzer => $a, create => $c)
```

Or sometimes the hash reference style:

```
IndexWriter->new({path => $p, analyzer => $a, create => $c})
```

The second thing to consider is that Java has a rather neat way of creating constructors for a class and accessors to its members. You simply declare the accessors as variables inside the class, and create a function with the same name as the class:

```
final class TermInfo {

    int docFreq = 0;
    long freqPointer = 0;
    long proxPointer = 0;

    TermInfo() {}

}
```

This creates a very simple, data-only class with a constructor and three accessors, with default values, in very little code at all.

The *Class::Accessor* Perl module gives us very much the same sort of thing:

```
package TermInfo;
use base 'Class::Accessor';
TermInfo->mk_accessors(qw/ doc_freq freq_pointer prox_pointer);
```

This gives us a new method that takes parameters in the hashref style above, and three methods to get or set the values of the appropriate data members. (Did you notice, incidentally, how we changed the names of the members from the usual Java camel-case style to the Perl lower-case-and-underscore style?) Now we can say:

```
my $ti = TermInfo->new({
    doc_freq    => 2,
    freq_pointer => 12,
    prox_pointer => 28
});

$ti->doc_freq(3);
...
```

Now I came to a dilemma. I wanted to use *Class::Accessor* to get this rapid development and clean access to data members, but I was also trying to emulate the Lucene API and wanted to keep the arguments roughly the same. This led to a mix of styles in the same program. This is, of course, very bad.

The reason this is particularly bad is that the interfaces between functions are the best place to spot erroneous parameters being passed around, and that's where *Carp::Assert* comes in handy.

Wouldn't it be nice, I thought, if there was some way to mix *Class::Accessor* with *Carp::Assert* to ensure that the values that you give to your constructor and accessors are what you expect? After quite a lot of struggling with the intricacies of *Class::Accessor*, I produced *Class::Accessor::Assert*.

This extends *Class::Accessor* with a tiny smattering of syntax: If you add a + before the name of a data member, it will be marked as required, and the constructor will fail if it is not present:

```
package Person;
use base 'Class::Accessor::Assert';
__PACKAGE__->mk_accessors(qw/ +name address date_of_birth /);

my $x = Person->new({ name => "Joel" }); # OK
my $y = Person->new({}); # Dies with backtrace
```

Additionally, if you add *=Some::Class* to the end of a member's name, it will ensure that that member is always an object of that class:

```
package Plucene::Index::Writer;
use base 'Class::Accessor::Assert';
__PACKAGE__->mk_accessors(qw/ +path create
                           +analyzer=Plucene::Analysis::Analyzer /);

my $x = Plucene::Index::Writer->new({ path => "/tmp/index/",
    analyzer => Plucene::Analysis::SimpleAnalyzer->new() });

$x->analyzer(undef); # OK
$x->analyzer(1);     # Dies with backtrace - not an ::Analyzer
```

Unfortunately, of course, I wrote the module after all of the more heinous interface incompatibility bugs in Plucene had been worked out, but it's something I'll be sure to use next time I'm ever converting code in a typed language into Perl...

TPJ

**Fame & Fortune
Await You!**

**Become a
TPJ
author!**

The Perl Journal is on the hunt for articles about interesting and unique applications of Perl (and other lightweight languages), updates on the Perl community, book reviews, programming tips, and more.

If you'd like share your Perl coding tips and techniques with your fellow programmers – *not to mention becoming rich and famous in the process* – contact Kevin Carlson at kcarlson@tpj.com.



Evaluating Short-Circuited Boolean Expressions

Randal L. Schwartz

In my recently released *File::Finder* module, one of the more interesting problems I found myself solving was how to evaluate the Boolean expression resulting from the *find*-like predicates. The *find* command evaluates the various tests using a combination of AND, OR, NOT, and parentheses operators with the traditional precedence rules, including short circuiting. For example:

```
find /tmp -size +30 -atime +3 -print
```

In this case, *find* first tries the size test. Only if that succeeds, the implied AND operator then tests the access time. Again, only if that succeeds do we finally evaluate the *print* operation, which always returns *true*. Had we included an OR operator in the chain:

```
find /tmp -size +30 -atime +3 -o -print
```

then the rules for short circuiting an OR apply: If the file is both big enough and old enough, the left side of the OR is true, so we do not execute the right side, and the printing is skipped. Similarly, if we move the OR:

```
find /tmp -size +30 -o -atime +3 -print
```

then the implied AND between the time rule and the printing rule now binds tighter, so we'll print only small old files now, instead of big ones.

In *File::Finder*, I chose to represent an expression such as this using an array of individual steps, with *coderefs* for the actual tests and simple strings for the operators (the AND operator always being implied). So, the test would look something like:

```
my @steps = (  
  \&code_for_testing_size_greater_than_30,  
  "OR",
```

Randal is a coauthor of Programming Perl, Learning Perl, Learning Perl for Win32 Systems, and Effective Perl Programming, as well as a founding board member of the Perl Mongers (perl.org). Randal can be reached at merlyn@stonehenge.com.

```
\&code_for_testing_atime_greater_than_3,  
\&code_for_printing,  
);
```

In practice, these *coderefs* were generated by closures that held the magic 30 and 3 constants in closure variables, but let's not get distracted here. The code to evaluate a series of *coderefs*, stopping at the first false *coderef*, started out looking something like this:

```
my $state = 1; # 1 = true, 0 = false  
for (@steps) {  
  if ($state) { # should we execute this?  
    my $result = $_->(); # run the coderef  
    unless ($result) { # did this fail?  
      $state = 0; # result is false  
    }  
  }  
}
```

This ensures that we stop running steps when the first step fails. This is our implied AND operator. The reason we keep going rather than exiting the loop is that even if we end up in a *false* state, we need to notice if there's an OR operator. This complicates things a bit. We'll need to introduce a third state: *skipping*. Why? Because we now have three states:

```
true AND ... # we execute this, thus true state  
false AND ... # we don't execute this, thus false state  
false OR ... # we do execute this, thus true state  
true OR ... # we don't execute this, but...  
true OR ... OR ... # we don't want the third part to execute!
```

So, after the OR is seen in a *true* state, we need to enter a sticky *false* state; hence, *skipping*.

```
my $state = 1; # 1 = true, 0 = false, -1 = skipping  
for (@steps) {  
  if (ref $_) { # is it a coderef?  
    if ($state > 0) { # should we execute this?  
      my $result = $_->(); # run the coderef
```

```

unless ($result) { # did this fail?
    $state = 0; # result is false
}
}
} elsif ($_ eq "OR") {
    if ($state > 0) { # true becomes skipping
        $state = -1;
    } elsif ($state == 0) { # false becomes true
        $state = 1;
    }
}
}
}

```

*It's always better to do
some explicit tests than to
try to perform fancy math to
get it to work right*

So, an OR causes *true* to become *skipping*, *false* to become *true*, and *skipping* to stay *skipping*. If you work through this, you'll see that this three-state decision tree now properly handles the implied AND and the explicit OR operator.

But wait, there's more. Let's add a NOT prefix operator into the mix, represented as a "NOT" string for a step. And let's presume that we can stack them. I found the easiest way to handle that was to introduce a fourth state: *true* (but invert the next test), encoded as 2 in *\$state*. Now we get something more complicated again:

```

my $state = 1; # 1 = true, 0 = false, -1 = skipping, 2 = "not"
for (@steps) {
    if (ref $_) { # is it a coderef?
        if ($state > 0) { # should we execute this?
            my $result = $_->(); # run the coderef
            $result = !$result if $state > 1; # flip result if needed
            $state = ($result ? 1 : 0);
        }
    } elsif ($_ eq "OR") {
        die "NOT before OR!" if $state > 1;
        if ($state > 0) { # true becomes skipping
            $state = -1;
        } elsif ($state == 0) { # false becomes true
            $state = 1;
        }
    } elsif ($_ eq "NOT") {
        # "true" and "not" swap states
        if ($state == 2) {
            $state = 1;
        } elsif ($state == 1) {
            $state = 2;
        }
    }
}
}

```

Note that if a NOT was seen before a *coderef*, we essentially need to act on the reverse of the outcome of the *coderef*, so we negate the result logically. Also note that it'd be wrong to copy *\$result* into *\$state* there because we don't demand that the *coderefs* return a simple 1 or 0: They can return any *true/false* value, so we have to normalize the results with a small question-colon operator.

It's also tempting to flip between the 2 and 1 in the last *elsif* block by trying something clever like:

```
$state = 3 - $state;
```

until you realize that *\$state* could also be 0 or -1, and we're trying to leave those alone. It's always better to do some explicit tests than to try to perform fancy math to get it to work right. The one notable exception I've seen to that is the AM/PM modulator to convert a 24-hour hour into a 12-hour hour:

```
my $hour_12 = ($hour_24 + 11) % 12 + 1;
```

although the magical "11" constant there deserves a well-placed comment in the source.

At this point, we have a full expression evaluator for AND, OR, and NOT operators with proper relative precedence. Let's add the COMMA operator now, from GNU *find*. This has the effect of restoring a true-state in the expression, ignoring any previous values (although their side-effect has already been acted upon). Seems simple enough:

```

my $state = 1; # 1 = true, 0 = false, -1 = skipping, 2 = "not"
for (@steps) {
    if (ref $_) { # is it a coderef?
        if ($state > 0) { # should we execute this?
            my $result = $_->(); # run the coderef
            $result = !$result if $state > 1; # flip result if needed
            $state = ($result ? 1 : 0);
        }
    } elsif ($_ eq "OR") {
        die "NOT before OR!" if $state > 1;
        if ($state > 0) { # true becomes skipping
            $state = -1;
        } elsif ($state == 0) { # false becomes true
            $state = 1;
        }
    } elsif ($_ eq "NOT") {
        # "true" and "not" swap states
        if ($state == 2) {
            $state = 1;
        } elsif ($state == 1) {
            $state = 2;
        }
    } elsif ($_ eq "COMMA") {
        die "NOT before COMMA!" if $state > 1;
        $state = 1; # restore true
    }
}
}

```

Now we're happy. We're dealing with NOT, AND, OR, and COMMA. At least, we remain happy until we remember that it's time to look at how to handle parentheses!

Shaking loose some old cobwebs in my brain, I remember an expression evaluator I saw some 30 years ago in a textbook that used a stack to handle the nested subexpressions. So, I took the same tactic here. (No doubt Mark Jason-Dominus will write me and tell me that a stack wasn't needed here, provided I applied *clever trick N*, but as I am a bear-of-very-little-brain, I'll take the straightforward approach.)

So, we'll do that by changing `$state` into `@state`, and every place we had `$state` before, we'll simply use `$state[-1]` (note that the top of the stack is the highest element). Without changing the functionality (or adding the paren-handling), that gets us this code:

```
my @state = (1); # start as true
for (@steps) {
    if (ref $_) { # is it a coderef?
        if ($state[-1] > 0) { # should we execute this?
            my $result = $_->(); # run the coderef
            $result = !$result if $state[-1] > 1; # flipresult if needed
            $state[-1] = ($result ? 1 : 0);
        }
    } elsif ($_ eq "OR") {
        die "NOT before OR!" if $state[-1] > 1;
        if ($state[-1] > 0) { # true becomes skipping
            $state[-1] = -1;
        } elsif ($state[-1] == 0) { # false becomes true
            $state[-1] = 1;
        }
    } elsif ($_ eq "NOT") {
        # "true" and "not" swap states
        if ($state[-1] == 2) {
            $state[-1] = 1;
        } elsif ($state[-1] == 1) {
            $state[-1] = 2;
        }
    } elsif ($_ eq "COMMA") {
        die "NOT before COMMA!" if $state[-1] > 1;
        $state[-1] = 1; # restore true
    }
}
```

A little messier, but no change in functionality. Next, we have to figure out how the state changes as we enter and leave a parenthesized area. First, let's see what happens when we enter a parenthesized area:

```
not ( ... now true
true ( ... now true
false ( ... now skipping
skipping ( ... now skipping
```

So, on a left paren, if `$state[-1]` is greater than 0, we push a 1, otherwise we push a -1. That's easy enough. The harder part is what to do at the end of the subexpression. The *false* or skipping value in front of the subexpression won't change at the end, so we can write that off right away. But we have six other combinations to consider: all the combinations of *not* or *true* before the subexpression crossed with *true*, *false*, and *skipping* at the end of the subexpression. Working through the combinations, we find that the rules are:

```
not ( ... true ) ... now false
not ( ... false ) ... now true
not ( ... skipping ) ... now false
true ( ... true ) ... now true
true ( ... false ) ... now false
true ( ... skipping ) ... now true
```

So, it looks like we can take *true* and *skipping* *xored* whether there's a *not*-state in front of the subexpression. The one other thing is that *comma* no longer restores always to *true*: It restores to whatever was set up at the start of the subexpression. Adding that code back in to the engine gives us the final result:

```
my @state = (1);
for (@steps) {
```

```
    if (ref $_) {
        if ($state[-1] > 0) {
            my $result = $_->();
            $result = !$result if $state[-1] > 1;
            $state[-1] = ($result ? 1 : 0);
        }
    } elsif ($_ eq "OR") {
        die "NOT before OR!" if $state[-1] > 1;
        if ($state[-1] > 0) {
            $state[-1] = -1;
        } elsif ($state[-1] == 0) {
            $state[-1] = 1;
        }
    } elsif ($_ eq "NOT") {
        if ($state[-1] == 2) {
            $state[-1] = 1;
        } elsif ($state[-1] == 1) {
            $state[-1] = 2;
        }
    } elsif ($_ eq "LEFT") {
        push @state, ($state[-1] > 0 ? 1 : -1);
    } elsif ($_ eq "RIGHT") {
        die "RIGHT without LEFT!" unless @state > 1;
        die "NOT before RIGHT!" if $state[-1] > 1;
        my $child = pop @state;
        $child = !$child if $state[-1] > 1; # reverse for NOT
        $state[-1] = ($child ? 1 : 0) if $state[-1] > 0; # inherit state
    } elsif ($_ eq "COMMA") {
        die "NOT before COMMA!" if $state[-1] > 1;
        if (@state > 1) { # in subexpression, restore to initial value
            $state[-1] = ($state[-2] > 0 ? 1 : -1);
        } else {
            $state[-1] = 1; # restore true if at outer level
        }
    }
}
```

Whew! I hope you enjoyed walking through this logic as much as I did in creating it. And if not, aren't you happy it's all nicely encapsulated in *File::Finder* so that you don't have to figure it out? Until next time, enjoy!

TPJ





Mastering Perl for Bioinformatics

Eric Forste

Mastering Perl for Bioinformatics, James Tisdall's sequel to his earlier *Beginning Perl for Bioinformatics* (2001), continues and extends the approach of the earlier book. Both books are aimed more toward working biologists seeking to learn Perl than at working programmers seeking to get into bioinformatics work. The new book is divided into two parts. Part I covers an introduction to data structures and object-oriented programming in Perl. Since the previous book restricted itself to declarative-style programming, this book is mostly concerned with developing the OO style used in most standard bioinformatics libraries. Part II gives a variety of one-chapter introductions to more specialized areas such as relational databases and graphics.

Part I, "Object-Oriented Programming in Perl," is written at a more elementary level than the manpages (e. g., `perlbot`, `perltoot`, `perltooc`) that accompany the Perl distribution, and is tailored for people who want more handholding, elaboration, and repetition of the concepts than can be found in the Perl documentation itself. Also, the simple fact that all the examples are motivated by problems in bioinformatics makes this discussion much easier to understand than the generic *Foo.pm* sort of examples used in the Perl documentation.

Tisdall opens with a discussion of modules and CPAN, developing a small sample module, *Geneticcode.pm*, which translates a string of DNA to their corresponding proteins. (The genetic code proper is implemented as a hash.) Then he introduces data structures by developing an approximate-string matching algorithm (suitable for finding mutated versions of the same gene) that uses two-dimensional arrays. This is a beautiful algorithm, the use of references is explained carefully, and the bioinformatic motivation is clear.

Chapter 3 is the heart of Part I, and we are treated to three successive versions of a *Gene* object, each using (and thoroughly explaining) a successively larger set of Perl's OO features. What we've learned is immediately put to work in the next chapter in the development of a class that will read in genes (and write them out again) in any of several different standard formats: a very familiar bioinformatics problem of the last decade or so. Some of the choices in class design seem strange to me. It is not clear why Tisdall makes the ">" write-mode flag (an invariant feature of Perl syntax) a class attribute called "`_writemode`" in some of the file access classes, but perhaps Tisdall feels this improves readability for novices. His explanation of the use of closures, on the other hand, is the clearest and most careful I have seen so far.

Eric has worked as a Perl programmer for more than five years and is currently working toward graduate school. He can be reached at arkuat@idiom.com

Mastering Perl for Bioinformatics
James D. Tisdall
O'Reilly & Associates, 2003
396 pp., \$39.95
ISBN 0-596-00307-2

Finally, a restriction enzyme database system is developed. This chapter may be very challenging for programmers who are new to biology because many of the biological terms discussed are only very briefly explained, if at all. Classes are developed to implement both the restriction enzyme database, and to map the binding sites (cutting points) for various restriction enzymes on any given gene. The Rebase system extends similar software that was developed and discussed in the earlier book, but this time around, everything is done according to standard object-oriented techniques.

Part II deals with four topics: RDBMS, web programming, graphics programming, and the installation and testing of the modules of the Bioperl project (bioperl.org). The database chapter gives a very brief introduction to RDBMS concepts, touching on the various levels of normal form and database design. The examples all use MySQL. Perl's DBI modules are explained and their use demonstrated. Finally, the Rebase system is extended by moving its originally DBM-file-based data into an RDBMS.

The web-programming chapter explains the benefits of web-based user interfaces (which are extremely widespread among bioinformatics applications) and then introduces the CGI module. Tisdall again extends the Rebase system, equipping it with a web-driven front end. The chapter on graphics programming introduces the *GD.pm* module, and extends Rebase by developing code to draw restriction maps. It is disappointing that none of the graphics of restriction maps produced by this code are illustrated in the book, although some ordinary spreadsheet-type graphs produced by the GD-based code are reproduced. Finally, we get a very detailed look at the process of installing and testing the Bioperl modules from CPAN.

This final chapter shows all the warts of working with new open-source code, which may often have bugs in the documentation and

inadequate tutorials. Tisdall does an excellent job of detailing the process of debugging a new installation, correcting bugs in tutorial code, and identifying missing pieces that you can get by without. At the end of the chapter, Tisdall explains the use of an invaluable debugging function of the `bptutorial.pl` script, which gives you a list of all methods invoked by any given Bioperl method, and names and locations of the Bioperl modules in which each of those methods is defined. This chapter on Bioperl will be very helpful for anyone who has ever been stymied in attempting to use CPAN to install a large and complex set of modules, as several of the most common problems are given detailed examples, along with a thorough account of how to work through to the solution of each one.

The book concludes with two appendices, a summary of Perl and a guide to installing Perl on various platforms. The summary is excellent, using bioinformatics-motivated examples throughout. It often gives examples that work together well and concisely illustrates aspects of Perl that I hadn't noticed as clearly before. The book is marred by a few typographical errors of the sort that might be especially confusing for beginners (e.g., the `mysql` prompt is occasionally shown as "`mysql>`"). I expect these will be fixed by the second printing, and despite them, the book is well worth reading.

This book is a very good way for someone who has been doing bioinformatics with traditional declarative programming to

get started with object-oriented Perl, and with the development and use of OO module libraries. It also gives a good first-hack approach to web programming and RDBMS programming, although any bioinformaticist needing to do more extensive work in these areas should seek further information elsewhere. Some of the explanations (in particular, the discussion of closures and some parts of the Perl summary in Appendix A) are clearer than I have seen elsewhere. I would heartily recommend this book to any working biologist wanting to master OO-style Perl (although it would be good to start with the predecessor volume if you're completely new to Perl), and would recommend it with only a few reservations to experienced Perl programmers wanting to start working with bioinformatics problems. Most of these reservations are due to the book's detailed explanations of things that require no explanation for the latter audience, while being a little vague about the workings of, say, reading frames and restriction enzymes. Several of these biological concepts are explained in the predecessor volume, but that volume is going to seem even more elementary (from the programming point of view) to experienced Perl programmers.

TPJ

Subscribe now to

Dr. Dobb's E-mail Newsletters

They're Free! <http://www.ddj.com/maillists/>

- ✓ **AI Expert Newsletter.** Edited by Dennis Merritt; the AI Expert Newsletter is all about artificial intelligence in practice.
- ✓ **Dr. Dobb's Linux Digest.** Edited by Steven Gibson, a monthly compendium that highlights the most important Linux newsgroup discussions.
- ✓ **Al Stevens C Programming Newsletter.** There's more than one way to spell "C." Al Stevens keeps you up-to-date on C and all its variants.
- ✓ **Dr. Dobb's Software Tools Newsletter.** Having a hard time keeping up with new developer tools and version updates? If so, Dr. Dobb's Software Tools e-mail newsletter is just the deal for you.
- ✓ **Dr. Dobb's Data Compression Newsletter.** Mark Nelson reports on the most recent compression techniques, algorithms, products, tools, and utilities.
- ✓ **Dr. Dobb's Math Power Newsletter.** Join Homer B. Tilton and expand your base of math knowledge.
- ✓ **Dr. Dobb's Active Scripting Newsletter.** Find out the most clever Active Scripting techniques from Mark Baker.

Sign up now at <http://www.ddj.com/maillists/>

Source Code Appendix

Deborah Pickett "Tracking TV Shows with Palm and Perl"

Listing 1

```
107 # Work on the series database.
108 {
109   if ($conduit)
110   {
111     my $seriesDBH = ColdSync::PDB->Load($seriesDBName, "rwx");
112     if (!defined $seriesDBH)
113     {
114       die "500 Can't load series DB";
115     }
116
117     # Probably shouldn't do this, but how else?
118     my $seriesDB = $seriesDBH->{helper};
119
120     # Load all of the database's records into helper class.
121     # Normally this'd be pretty inefficient, but we have to
122     # load them all anyway, so let's get it over with.
123     my @records = $seriesDBH->records();
124
125     # Go through the series database field names and determine which is
126     # which.
127     processSeriesDBInfo($seriesDB);
128
129     # Process each series record and write it back.
130     foreach my $series (@{$seriesDB->{db_records}})
131     {
132       processSeriesDBRecord($series);
133       # Always write the new record.
134       $seriesDBH->writeRecord($series);
135     }
136
137     # Undefine the database handle to close it.
138     undef $seriesDBH;
139     undef $seriesDB;
140   }
141   else
142   {
143     my $seriesDB = Palm::PDB->new();
144     $seriesDB->Load($seriesDBName . ".pdb");
145
146     # Go through the series database field names and determine which is
147     # which.
148     processSeriesDBInfo($seriesDB);
149
150     foreach my $series (@{$seriesDB->{db_records}})
151     {
152       processSeriesDBRecord($series);
153     }
154
155     # Write the episode database, now finished with it.
156     $seriesDB->Write("new_" . $seriesDBName . ".pdb");
157     undef $seriesDB;
158   }
159 }
```

Listing 2

```
162 # Now load the episode database.
163 {
164   my $episodeDBH;
165   my $episodeDB;
166
167   if ($conduit)
168   {
169     $episodeDBH = ColdSync::PDB->Load($episodeDBName, "rwx");
170     if (!defined $episodeDBH)
171     {
172       die "500 Can't load episode DB";
173     }
174
175     # Probably shouldn't do this, but how else?
176     $episodeDB = $episodeDBH->{helper};
177
178     # Load all of the database's records into helper class.
179     my @records = $episodeDBH->records();
180
181     # Go through the series database field names and determine which is
182     # which.
183     processEpisodeDBInfo($episodeDB);
184
185     # Process each episode record and write it back.
186     foreach my $episode (@{$episodeDB->{db_records}})
187     {
```

```
187     if (processEpisodeDBRecord($episode))
188     {
189         # episode has changed, so write the record.
190         $episodeDBH->writeRecord($episode);
191     }
192 }
193 }
194 else
195 {
196     $episodeDB = Palm::PDB->new();
197     $episodeDB->Load($episodeDBName.".pdb");
198
199     # Go through the episode database field names and determine which is
200     # which.
201     processEpisodeDBInfo($episodeDB);
202
203     # Process known episodes first.
204     foreach my $episode (@{$episodeDB->{db_records}})
205     {
206         # Don't care about whether record has changed, so ignore return
207         # value.
208         processEpisodeDBRecord($episode);
209     }
210 }
211 }
```