

# *The Perl Journal*

## **Mouse Tracking with JavaScript and Perl**

Peter Sergeant • 3

## **Test-Driven Development in Perl**

Piers Cawley • 7

## **A Magic Header for Starting Perl Scripts**

Péter Szabó • 12

## **Building a Better Mail Handler**

Simon Cozens • 16

## **Tailing Web Logs**

Randal Schwartz • 20

## **PLUS**

**Letter from the Editor • 1**

**Perl News by Shannon Cochran • 2**

**Book Review by Tim Kientzle:**

***Test-Driven Development By Example* • 23**

**Source Code Appendix • 25**

## LETTER FROM THE EDITOR

# The Computer Ate My Homework

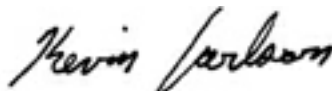
I had an almost entirely technology-free education. Not only did the Web not exist when I was in school, but the idea of computers in the classroom was still mostly science fiction. Today, despite a great push to incorporate computers into both K-12 and college curricula, many students entering our school systems find that *they* are the ones teaching their instructors in the use of computers. The effort to integrate computers with education has met with only partial success.

True, some school systems can't afford the technology. But this is about more than the digital divide between the wealthy and the poor. Even affluent schools that can provide the tools for their students can't seem to make consistent, effective use of them. Are we looking at a failure to train our teachers properly in the use of computers? Perhaps. But I suspect that even if every teacher in America was a technical wizard, we would still have great difficulty in using technology to teach, simply because all our plans have focused on harnessing the benefits of digital systems, and have ignored the important goal of managing the complexity and negative consequences of those systems.

Take the simple issue of maintenance. At a time when budget cuts are forcing teacher layoffs, school districts can't afford the IT personnel to adequately maintain functioning desktop systems and LANs. So even when schools have technology, it's often broken. Students using their own computers at home for assignments are similarly adrift. If they're lucky, they or a family member have the skills to keep system and application software all running smoothly. But more often than not, kids spend their increasingly limited evening hours waiting for Mom or Dad to make tech-support calls to get the printer to print out their term paper. Pencil and paper never threw those sorts of roadblocks in a student's path.

Other negative impacts of technology that can affect students are all too familiar even to those of us not in school. For example, the loss of free time. Unless we make a Herculean effort to the contrary, most of us are enslaved by our e-mail. In a time when worker productivity is on the rise only because those of us who haven't been laid off must pick up the work of those who have departed, it becomes increasingly difficult to leave work at the office. Real vacations are not an option for many of us, as we are forced to be in contact even when we're supposed to be relaxing. Students, too, face mounting pressure to be permanently, electronically jacked-in to the demands of teachers. Students who get assignments on Friday may get Saturday e-mails from teachers demanding that new work be done in time for Monday's class. This is something a teacher would *never* do if it meant picking up the phone and calling each student. And, of course, this goes in the other direction: Teachers can open themselves up to unrealistic demands on their attention when they give out e-mail addresses to students.

Computers have clear benefits in the classroom, and current curriculum guidelines for their use are a good starting point for reaping those benefits. But unless we develop similar guidelines for reducing the effects of these ancillary complications, the adoption of computers in the classroom may become permanently stalled.



Kevin Carlson  
Executive Editor  
kcarlson@tpj.com

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

THE PERL JOURNAL is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: \$12.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2003 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

### 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, Rusa Vuong

### 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 COO

Steve Weitzner

#### EXECUTIVE VICE PRESIDENT AND CFO

John Day

#### CHIEF INFORMATION OFFICER

Mike Mikos

#### PRESIDENT, TECHNOLOGY SOLUTIONS GROUP

Robert Faletra

#### PRESIDENT, HEALTHCARE GROUP

Vicki Masseria

#### PRESIDENT, ELECTRONICS GROUP

Jeff Patterson

#### PRESIDENT, SPECIALIZED TECHNOLOGIES GROUP

Regina Starr Ridley

#### SENIOR VICE PRESIDENT, GLOBAL SALES AND

MARKETING

Bill Howard

#### SENIOR VICE PRESIDENT, HUMAN RESOURCES AND

COMMUNICATIONS

Leah Landro

#### PRESIDENT AND GENERAL COUNSEL

Sandra Grayson

# Perl News

## Apocalypse 6 Is Out

Larry Wall has published the Apocalypse on subroutines, available online at <http://www.perl.com/pub/a/2003/03/07/apocalypse6.html>. "It is radical, but you'll like it anyway (we hope)," he writes. "At least the old way of calling subroutines still works. Unlike regexes, Perl subroutines don't have a lot of historical cruft to get rid of. In fact, the basic problem with Perl 5's subroutines is that they're not crufty enough, so the cruft leaks out into user-defined code instead, by the Conservation of Cruft Principle. Perl 6 will let you migrate the cruft out of the user-defined code and back into the declarations where it belongs."

## Parrot 0.0.10 Released

The latest Parrot release, codenamed "Juice" as a nod to the new -Oj optimization flag, is now downloadable from CPAN (<http://www.cpan.org/authors/id/S/SF/SFINK/parrot-0.0.10.tar.gz>). The -Oj flag, which allows use of IMCC metadata for JIT optimization, is one aspect of Parrot's new IMCC integration. The IMCC features, along with a fast CGP (Computed Goto Prederefed) run-loop for operations not handled by the JIT core, should lead to substantial improvements in speed. Among many other new features, Parrot has also gained *eval* instruction for opcodes; fixes for subroutines, coroutines, and continuations; more benchmarking; optimized math operations; and improvements to the native calling interface. Steve Fink also called attention to the "shiny new Basic compiler to go along with the earlier Basic interpreter."

## Perl to Spark Tourism Boom

The Barcelona Perl Mongers group, founded two years ago, has resumed regular meetings. The group has a mailing list (<http://barcelona.pm.org/>) and is establishing a schedule for gatherings; meetings are to be held on the fourth Thursday of each month. "We welcome Perl Monger tourists!" the group posted to use.perl.org.

Globetrotting Perl aficionados can add Hamburg, Germany to their itinerary as well; the new Perl Mongers group there (<http://hamburg.pm.org/>) meets on the second Wednesday of each month. And on April 8th, the Washington D.C. Perl Mongers group (<http://dc.pm.org/meeting.html>) will throw open their doors to anyone interested in hearing Mark Jason Dominus describe "six useful Perl tools that you probably didn't know you needed."

## YAPC::Europe Searching for Venue

As this year's YAPC::Europe conference in Paris draws near, organizers of the event are starting to think about next year's gathering. The Belfast group has already mentioned a willingness to host YAPC::Europe::2004; any other interested groups should send e-mail to [committee@yapceurope.org](mailto:committee@yapceurope.org). Venue requirements are described in the CPAN module *YAPC::Venue*, or online at

<http://www.yapc.org/venue-reqs.txt>. Briefly, the conference site must offer several rooms of varying size for assemblies, meetings, storage, dining, and so forth; catering services for three days; network and a/v equipment; and access to an airport.

## ActiveState Teams with O'Reilly

The "Perl bundle" in ActiveState's redesigned Programmer Network (ASPEN) now includes access to O'Reilly's Safari line of online books. "Members can now bookmark and annotate this content, even cut-and-paste the code directly into their programs," ActiveState announced. An ASPEN Perl membership goes for \$495; the site is at <http://www.activestate.com/ASPEN/>.

## DBI 1.35 Released

Tim Bunce announced three new DBI releases in quick succession, culminating in DBI 1.35. A few new methods have been implemented, including *clone* (to make a new connection to the database that is identical to a previous connection), *can* (to check if a given method is implemented by the driver or if a default method is provided by the DBI), and *install\_method* ("so driver private methods can be 'installed' into the DBI dispatcher and no longer need to be called using *\$h->func(..., \$method\_name)*"). Furthermore, Bunce warns that "Future versions of the DBI will not support Perl 5.6.0 or earlier: Perl 5.6.1 will be the minimum supported version." The full list of changes to the DBI module are listed at <http://archive.developer.com/dbi-announce@perl.org/msg00149.html>.

## Flash Remoting in Perl

Simon Ilyushchenko and Adrian Arva are working on an open-source project to implement Macromedia's AMF protocol in Perl, allowing developers to create Flash movies that communicate with the server to get data and updates without relying on Macromedia's Flash Remoting server tool. "If you believe in the idea of more and more programmers taking a shot at developing clients in Flash, you must also see the need for a good data gateway between web clients (the Flash movie) and web servers. Macromedia offers just that in Macromedia Flash Remoting, available for ColdFusion, JRun, .NET, J2EE," they write. "We think that it is very important for the Open Source community to make this technology available in Perl and (why not?) in Python as well. We set out to decode the protocol, but soon discovered that PHP folks beat us by a month, so we simply rewrote their code in Perl." Flash Remoting in Perl (FLAP) is currently in a 0.02 release; contributions to the project are welcomed. The project's web site is at <http://simonf.com/flap/>.

*We want your news! Send tips to editors@tpj.com.*

# Mouse Tracking with JavaScript and Perl

You've spent months juggling requirements from many departments around the company, liaising with graphic designers, battling various browser quirks, getting copy approved, and churning out code. You've bravely battled Apache, assimilated a templating ideology, and written a flotilla of scripts to turn access logs into pretty graphs to show management. The resulting web site is beautiful and is backed by enough cool code to ensure your bragging rights for the next five years at your local Perl Mongers meetings.

Trouble is, none of your users can find anything—it turns out, nobody else quite “gets” your idea of basing the site navigation around a fishing analogy, and judging by your pretty graphs, not a single visitor has found the online store. Seems that in the rush to production, nobody thought about site usability.

Something needs to be done, and fast. But usability testing can be expensive, and good usability advice can be hard to find. What you need is a way to watch your users navigate the site, find out where they hesitate, figure out what elements draw their attention and which they ignore, all without requiring your intervention or extra funds. Even in this situation, Perl can help.

## The Plan

In this article, we're going to build a system to track users' mouse-trails across pages on a web site, and then graph them in Perl. We'll look briefly at how we record the raw data (using JavaScript), how we collect it, and how we can use Perl to generate useful and management-friendly graphs to track down usability problems. We'll end the article with a quick look at privacy issues raised by doing this.

## Gathering the Data

Please note: The JavaScript below is written to work with Internet Explorer. It's trivial to port to Mozilla or Opera, but it won't work as is in either.

Like any other user-interface toolkit, JavaScript in the browser allows you to set handlers for many types of user interaction. We're interested in knowing when the user has moved the mouse, and when the user clicks the mouse. Setting these handlers up is easy:

```
// Set up our on-movement function
document.onmousemove = getMouseXY;
document.onmousedown = setMouseClicked;
```

---

Peter is employed as a web developer by Virus Bulletin (<http://www.virusbt.com/>). He can be contacted at [tpj@clueball.com](mailto:tpj@clueball.com).

*getMouseXY* and *setMouseClicked* are the names of the functions we want to call. If we can have a function invoked each time the mouse is moved, and we can retrieve the coordinates of the mouse, then it stands to reason that we could just create a large data structure with an entry for each time *getMouseXY* and *setMouseClicked* are invoked, and pass it back to the server. Sadly, it's a little more complicated.

First, we need timing information about each movement we record so that we can accurately recreate the user's browsing experience—if a user's mouse hovered stationary over a part of the page for 30 seconds, we want to know, rather than just knowing their mouse was there at some point. Second, if we collect coordinates for each movement the mouse makes, we will end up with far too much data—we need to decide just what level of granularity we want or passing the data around becomes impractical. Finally, passing the data back to the server doesn't happen automatically. We need a nonintrusive way to do it.

Storing our data in a data structure that can understand time—well, one that can understand numeric representations of time, anyway—easily solves the first two problems. Simply, if we index our mouse coordinates against time and represent time as a number, we can store our data in an array. The code below clarifies a little:

```
var currentTime = Math.round(((currentTime.getTime() -
    startTime.getTime()) / 10));
movementStore[currentTime] = "a" + coordX + "b" + coordY
```

Here, we divide by 10 the number of milliseconds (in epoch milliseconds) elapsed between the page loading and the current time, and we store the coordinates in our data structure using the time as their index. If the user moves the mouse quickly, then the last coordinates gathered in that 10-millisecond time frame are the ones we keep. To record clicks, we use an almost identical method (as we're storing them in the same data structure), only we store them at *currentTime--1*, so they won't be overwritten.

When it's time to send the data back to the server, we go through *movementStore*, adding the entries to another string. If there's no data at a specific entry, it means the user hadn't moved the mouse, so we add a single character denoting this. For reasons explained later, when the string gets to 3500 characters, we stop. We also add the dimensions of the user's screen to the string, as that information will come in handy.

Now we need to pass this string back to the server. GET and POST are out, as they'd involve dynamically changing the links on our page, and that doesn't qualify as nonintrusive. Therefore, when the user leaves the page, we set them a cookie containing the data string we created. This data string can be a maximum of 4-KB long, hence the restriction on the aforementioned number of characters. If the next page they land on is one on our web site, the cookie is retrieved by a tiny CGI script pretending to be an image, and added to a flat file for later use.

## Working with the Data

At this point, we have lots of options. On a site with fairly high traffic, you'll have built up quite a large selection of data in very little time. So what can we do with it that's useful? When I first built this system, I did two things with it. First, I wrote some more JavaScript and some Perl so that we could look at individual (and effectively anonymous) users navigating the page—I'd recreate their experience by using a small image to represent the mouse and exactly emulate their mouse as they accessed the page. That

The screenshot shows the Virus Bulletin website with the following content:

**VIRUS BULLETIN** Independent Anti-Virus Advice

news resources magazine vb100award conference support contact

**VB2003 Call for Papers** [ from conference ]  
 Virus Bulletin is seeking submissions from those wishing to present papers at VB2003 in Toronto, Canada. [ more ]

**W32/Opaserv** [ from viruses ]  
 W32/Opaserv.A's originality resides both in its use of an exploit to infect Windows shares, and in its cryptic payload. [ more ]

**What's new...**

**Anti-virus for Lindows**  
 20 February 2003  
 Lindows teams up with Central Command to sell Linux anti-virus - a step in the right direction, but perhaps not far enough... [ more ]

**Symantec press release backfires**  
 17 February 2003  
 Watch out for your marketing department... [ more ]

**Virus writers get a helping hand**  
 23 Jan 2003  
 Two organizations send viruses to mailing list subscribers... [ more ]

Copyright © 2003 Virus Bulletin Ltd

**Upcoming events:**

Date	Event	Location
Feb 24 - 27	Black Hat Windows Security 2003	Seattle, WA, USA
March 7 - 12	SANS	San Diego, USA
March 10 - 12	InfoSec World Conference & Expo 2003	Orlando, FL, USA
March 12 - 19	CeBIT	Hannover, Germany

[ more ]

**Virus profile:**

**Slammer**  
 Slammer exploited a buffer overflow in Microsoft SQL server to become the 'fastest computer worm ever recorded', infecting 75,000 vulnerable hosts in ten minutes - no mean feat. The worm sends UDP packets to port 1434 (the port on which MS SQL server listens on) to random IP addresses. The traffic generated by this caused wide-spread problems.

**NPACI report**  
 A detailed report by the American National Partnership for Advanced Computational Infrastructure on the spread of Slammer.

**NAI's analysis of Slammer**  
 A good technical overview of Slammer

**Sophos's Slammer FAQ**  
 Quick questions and answers about Slammer.

**Quick links:**

- Hoaxes
- What's on
- VGrep
- Recovery tutorials

Search

lost? try the [sitemap](#)

Figure 1: Data gathered on the Virus Bulletin web site (<http://www.virusbtl.com/>) over a three-day period.

approach requires a little Perl and a lot of JavaScript, so we're not going to pursue that avenue in this article.

The second thing I did was to create an image depicting hot spots on the page—by taking a large amount of user data, splitting the page into little chunks, and seeing how many times users' mice had been logged as being in that section. When navigating a web site, users' mouse movements tend to follow their eyes. We can exploit this tendency to see which parts of the page the users are concentrating on and which they are ignoring. I found out, to my dismay, that users were barely considering my (in my opinion) ultra-useful, quick-links section (see Figure 1). I was able to experiment with putting it in different places on the page to see where it got the most user attention.

To plot the data, we're going to use *Imager*, an open-source Perl image-manipulation library. We'll take an image of the web site at a certain resolution to start with, and cover it in blue squares, the opacity of which will show the "popularity" of that square.

## Creating Our Chart

There are several things we need to consider before jumping straight into chart generation. First, the world isn't perfect. We're going to get people who, despite having a high screen resolution, don't have their browser set to fill the whole screen, so their results are going to be slightly funky, especially if the target page is centered rather than left aligned. Also, people may leave their browser open while they go and get themselves a coffee—that will result in one user generating a huge number of hits on a very small part of the screen. Still, we want to try and get meaningful results from the data.

For this reason, we're going to do two things to normalize our data a little. We're only going to allow a given user to affect the graph a certain amount—each user has an equal number of "points" they can use on the graph in total. For example, if we have 10 users, and each user has a maximum of 10 points they can assign to a square, the maximum rating for a square is going to be 100. We'll also ignore all hits to a square over a certain number. This number is fairly arbitrary, and one you'll want to experiment with a little. We're going to use 20 as our starting point.

At this point, we can begin writing our program, which is shown in Listing 1. The program needs to read in our data file created by the aforementioned JavaScript/CGI combination. To simplify matters, we're also only going to deal with people of a certain screen resolution to begin with, so we weed out other entries. This is dealt with in line 34 of the Listing.

As we go through each user, we need to make a note of the last coordinates we read for them, so that if we come across a blank entry that denotes that the cursor didn't move, we know where it was. We create a hash for each user, with the coordinates for the top-left pixel of each square as keys, and increment the value of each hash entry when we find a match. We define the holders for the last coordinates on line 37 and create the user hash on line 38. We then merge the user's hash into the main hash in lines 60–69.

The next step is to build the image itself. We start off by reading the image of the page into an *Imager* object (line 80) and make

sure the image can handle transparency by adding an alpha channel (line 81). We then go through the hash, key by key, and add a blue box for each entry in the hash. To get the opacity for a given number of hits, we keep a running total of the highest number of hits we have (lines 65–67), divide the maximum amount of opacity we want by that (we'll use 200 in this case, but again, that's fairly arbitrary), and then multiply the hit count by it when creating the box fill (lines 97–100). We then draw the box (lines 103–111), and output the image (line 116).

## Interpreting the Data

I've included an image created from data gathered on the Virus Bulletin web site (<http://www.virusbtn.com/>) over a three-day period (see Figure 1). Some time has already been spent analyzing data from this site and improving it, but let's see what we can learn from the image.

The navigation bar at the top is clearly the darkest area, and for fairly obvious reasons. If we look down the page a little, we see that our quick links and search box, considering they're "below the fold," are fairly popular. However, and noticeably, it seems barely anyone is interested in the links on the Slammer story—it seems a fair number started reading it (there are some dark patches near the top), but not so many followed through. Perhaps for the next "Virus profile," it's not worth including links and there should be more copy instead.

(Note how the areas under the main navigation bar are darker than those on the top. People will slow down their mouse as they approach a link, so this suggests people are approaching the links from the bottom. Why? Think about the shape of the default Windows cursor...)

## Other Projects

The only other project of this ilk that I've found is called "Cheese" (<http://cac.media.mit.edu/cheese.htm>), which came from MIT. Cheese aims to look for patterns in the way that users browse the Web, in order to give people a more personalized browsing experience. However, the project seems to have somewhat faded from attention.

## Privacy

So is it wrong to collect this data from users? If you're collecting this sort of data, make sure your privacy policy spells out why you're collecting it and what you intend to do with it. Go as far as you can to make sure the data isn't personally identifiable. There's little reason to also capture the IP address. At the end of the day, your browser gives away an awful lot of information about you anyway: the links you take through the site, the browser software you're using, your IP address (thus, your ISP and quite possibly your physical location). If you can convince people you're using the data responsibly, you'll run less risk of people taking issue with it.

TPJ

### Listing 1

```
1 #####!/usr/bin/perl
2
3 use strict;
4 use Imager;
5 use Data::Dumper;
6 use Imager::Fill;
7
8 my %config = (
9
10 'Box Dimensions'    => 10,
11 'Max Box Score'    => 20,
12 'Site Image'       => 'screenshot.png',
13 'Output Image'     => 'outmouse.png',
14 'Mouse Trails'     => 'cookies.log',
15 'Screen Size'      => '768x1024',
16 'Count Repeats'    => 3,
17 'Max Opacity'      => 200,
18
19 );
20
21 my %grid_score_hash;
22 my $high_score;
23
24 open( LOGFILE, "< $config{'Mouse Trails'}" ) || die $!;
25
26 while(<LOGFILE>) {
27
28     chomp;
29
30
31
32     # Check if it's the right screen-size
```

```

33 # A sample data line looks like: 768z1024|aa234b82aaaaaaaa229b94a223b145
34 next unless substr( $_, 0, 9, '' ) eq $config{'Screen Size'} . '|';
35
36 # Create some useful holding variables
37 my ($old_x, $old_y) = (0, 0);
38 my %user_hash;
39
40 # Extract coordinate readings from our data line
41 for (split(/a/, $_)) {
42
43     # Extract the coordinates themselves from our coordinate block
44     my ($x_coord, $y_coord) = split(/b/, $_);
45
46     # Normalize the coords
47     $x_coord = int( $x_coord / $config{'Box Dimensions'} );
48     $y_coord = int( $y_coord / $config{'Box Dimensions'} );
49
50     # If the coordinate is blank, set it to the last-read one
51     $x_coord = $old_x unless $x_coord;
52     $y_coord = $old_y unless $y_coord;
53
54     # Cache the values
55     $user_hash{"$x_coord|$y_coord"}++
56     unless $user_hash{"$x_coord|$y_coord"} >= $config{'Max Box Score'};
57 }
58
59
60 for (keys %user_hash) {
61
62     $grid_score_hash{$_} += $user_hash{$_} unless $_ eq '0|0';
63
64     # Calculate high-score
65     if ($grid_score_hash{$_} > $high_score) {
66         $high_score = $grid_score_hash{$_}
67     }
68 }
69
70
71 }
72
73 # Work out the opacity multiplier
74 my $opacity_multiplier = ( $config{'Max Opacity'} / $high_score );
75
76 # Create new Imager object
77 my $start_image = Imager->new();
78
79 # Open our site image
80 $start_image->open( file => $config{'Site Image'} )
81     or die $start_image->errstr();
82
83 my $image = $start_image->convert( preset => 'addalpha' );
84
85 # We cache Imager colours here to save duplication
86 my %fill_cache;
87
88 # Go through the hash
89 for (keys %grid_score_hash) {
90
91     my ($xcoord, $ycoord) = split(/\|/);
92     $xcoord *= $config{'Box Dimensions'};
93     $ycoord *= $config{'Box Dimensions'};
94
95     # Work out the opacity
96     my $opacity = int( $grid_score_hash{$_} * $opacity_multiplier );
97
98     # Create a fill in Imager
99     $fill_cache{$opacity} = Imager::Fill->new(
100         solid => Imager::Color->new( 0, 0, 255, $opacity ),
101         combine => 'multiply'
102     ) unless $fill_cache{$opacity};
103
104     # Add a box to the imager in the appropriate place
105     $image->box(
106         fill => $fill_cache{$opacity},
107         xmin => $xcoord,
108         ymin => $ycoord,
109         xmax => ($xcoord + ( $config{'Box Dimensions'} - 1 )),
110         ymax => ($ycoord + ( $config{'Box Dimensions'} - 1 )),
111         color => Imager::Color->new( 0, 0, 255, $opacity ),
112         #filled=>1
113     );
114
115 # Print our image
116 $image->write( file => $config{'Output Image'} ) or die $image->errstr;

```

TPJ

# 101 Perl Articles!



From the pages of *The Perl Journal*, *Dr. Dobb'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*

**\$22.90** To subscribe to *The Perl Journal* and receive *101 Perl Articles*

Go to

**<http://www.tpj.com/>**  
now!

# Test-Driven Development in Perl

Recently, I've been reading Kent Beck's inspirational book *Test Driven Development*, in which he demonstrates with examples (in Java and Python) the process of driving development by writing a test for new functionality, coding until it passes, refactoring ruthlessly, and going back to the start of the loop with a new piece of functionality. I had also been looking for an opportunity to try out this method, and settled on the idea of creating a class "helper" module as an exercise in test-driven development. In this article, I'll build a basic helper module using this technique.

## What Is a Class Helper Module?

If you write object-oriented Perl, you often find yourself writing many simple little pairs of methods with names like *foo/set\_foo*, which simply provide you with uniform access to your object attributes. If you're like me, you get bored with this very quickly and start combing CPAN for a tool to do the grunt work for you—a helper module. And you'll find plenty of them. I recommend that you take a look at *Class::MethodMaker* and *Class::MakeMethods* for starters.

However, none of the CPAN modules I've found do quite what I want. The main issue I have with almost all of them can almost be thought of as philosophical. In the tools I've tried, setting methods generally return either the new or the old value of the attribute being set. However, I don't like writing factory methods that look like:

```
sub Human::make_with_name {
    my $class = shift;
    my($name, $surname) = @_;

    my $self = $class->new;
    $self->set_name($name);
    $self->set_surname($name);
}
```

I'd much rather have my setting methods return the object being altered because that lets me write a method that looks like:

---

*Piers is a freelance writer and programmer, and the writer of the Perl 6 Summary. He has been programming Perl since 1993, and is currently working on Pixie, an object persistence tool. He can be reached at [pdcauley@bdfh.org.uk](mailto:pdcauley@bdfh.org.uk).*

```
sub Human::make_with_name {
    my $class = shift;
    my($name, $surname) = @_;

    return $class->new->set_name($name)
        ->set_surname($surname);
}
```

This eliminates a useless variable and a lot of repetition. It's a matter of style, but I happen to think it's an important one.

## Choice of Tools

I could have written my tests using Michael Schwern's excellent *Test::More* module, but I'm keen on xUnit-style testing, so I chose Adrian Howard's *Test::Class*, which neatly layers an xUnit-style interface on top of Perl's standard *Test::Builder* and *Test::Harness* testing framework.

## Requirements

It's impossible to build anything unless you know what you want. Here's an initial list of my requirements with a few comments about their importance and whether they're "real" requirements or constraints. Note that we won't meet all of these requirements in this article. It's important, however, to start from as complete a set of requirements as possible, so I include them all here for the sake of completeness. These requirements will form the basis of our tests.

- A simple constructor method: The basic constructor should need no arguments and return a simple object.
- Constructors should call *\$self->init(@\_)*.
- It should always be safe to call *\$self->SUPER::init*. This just makes life easier.
- It should generate an accessor method that looks like "*attrib\_name*."
- It should generate a setter method that looks like "*set\_attrib\_name*." The setter method should return the object being modified in preference to the new attribute value.
- Lazy initialization. It should be possible to use lazy initialization for objects, either by specifying a method, a default value, or a coderef. (Ideally, this would be broken up into smaller requirements.)
- Special attribute types should have useful helper methods; for instance, lists should allow *\$obj->attrib\_append(@array)* and



other listlike ops. Again, this needs to be broken up into smaller requirements—at least one per attribute type.

## Coding, Testing, and Refactoring

I'll be presenting this process in a series of very small steps. The idea is that at each step of the way, we should have a simple, clear goal that we can reach with simple, clear code.

First, we need to set up an empty package and test environment, and get the project safely into version control. The following commands did the job for me:

```
$ cd ~/build
$ h2xs -use-new-tests -skip-exporter -skip-autoloader \
> -AXPn Class::Builder
$ cd Class/Builder
$ cvs import Class-Builder vendor release
$ cd ~/build
$ cvs get Class-Builder
$ rm -rf Class
```

Picking “simple constructor method” from the list of requirements means we can write a test. Our first test looks pretty simple:

```
package BuilderTest;
use base 'Test::Class';
use Test::More;

sub test_creation : Test(2) {
    eval q{ package ScratchClass;
            use Class::Builder new => 'new' };
    die $@ if $@;
    my $scratch = ScratchClass->new;
    ok ref $scratch, 'Scratch object is a reference';
    isa_ok $scratch => 'ScratchClass',
        'Scratch object is an instance of ScratchClass';
}

BuilderTest->runtests;
```

And of course, when we run the test suite, “the bar is red.” (The original sUnit interface has a progress bar that updates as all the tests in the suite are executed. When any of the tests in a test suite fail, the bar goes red. When all the tests are passing, the bar is green. Even though *Test::Class* doesn't have a progress bar, the idiom is too useful not to use it here.) So, we write the simplest code we can, just to get a green bar:

```
package Class::Builder

sub import {
    *ScratchClass::new = sub { bless {}, 'ScratchClass' };
}

1;
```

This code isn't great, but our only goal when we have a failing test is to write just enough code to get a green bar. So, we run the test suite, our failing test passes, and the bar is green. Now we can refactor.

Refactoring can be thought of as a process of removing redundancy in code. Looking at the code as it stands, there's some immediately obvious redundancy: the duplication of the string *ScratchClass* in both the test code and the implementation. So we fix that:

```
sub import {
    *ScratchClass::new = sub { bless {}, $_[0] };
}
```

And the bar stays green. Let's see if we can eliminate the other explicit use of *ScratchClass* in our import routine:

```
sub import {
    my $calling_class = caller(0);
    *{"$calling_class::new"} = sub { bless {}, $_[0] };
}
```

Still green. Of course, there's still a bug there, so we'll expose it with another test:

```
sub test_custom_creation : Test {
    eval q{ package ScratchClass2;
            use Class::Builder new => 'custom_new' };
    die $@ if $@;
    my $scratch = ScratchClass2->custom_new;
    ok isa_ok $scratch => 'ScratchClass2',
        'Scratch object is an instance of ScratchClass2';
}
```

---

*Refactoring can be thought  
of as a process of removing  
redundancy in code*

---

This fails because there's no *custom\_new* method in *ScratchClass2*. Writing this test has made me think about constructors. In particular, I wonder if I ever call my simple constructors anything other than *new*? On reflection, I never do. So I'm better off not bothering to deal with custom constructor names. I can always come back and add them later if a need arises.

There's still something to think about, though. I could just have *Class::Builder* always generate a constructor and always call that constructor *new*, but that ignores inheritance. You only need to generate a *new* method for a parent class, or the package's user might need to write a more complex constructor themselves. So, we need some way of specifying whether or not to generate a constructor. Let's rejig the tests:

```
use Test::Exception;

sub test_creation : Test(2) {
    eval q{ package ScratchClass;
            use Class::Builder has_constructor => 1 };
    die $@ if $@;
    my $scratch = ScratchClass->new;
    ok ref $scratch, 'Scratch object is a reference';
    isa_ok $scratch => 'ScratchClass',
        'Scratch object is an instance of ScratchClass';
}

sub test_custom_creation : Test {
    eval q{ package ScratchClass2;
            use Class::Builder };
    die $@ if $@;
    dies_ok { ScratchClass2->new }
        "Constructor not generated by default.";
}
```

Now, instead of passing a `new => <method_name>` pair of arguments, we pass `has_constructor => 1` if we want `Class::Builder` to generate a constructor. Running the tests, we get a red bar. Rewriting `Class::Builder`, we now have:

```
sub import {
  my $class = shift;
  my %args = @_;
  my $calling_class = caller(0);
  if ($args{has_constructor}) {
    no strict 'refs';
    *{"${calling_class}::new"} = sub { bless {}, $_[0] };
  }
}
```

And the bar is green.

## Building Accessor and Modifier Methods

So, let's pull another feature off our desirable list. We're now at the point where we're able to create objects, but we don't have any accessor or modifier methods. Let's add modifier methods first. After all, if you can't set an attribute, you can't very well access it, can you? The first step is coming up with a syntax for specifying them. Poaching from `Class::MethodMaker`, I reckon that:

```
use Class::Builder
  get_set => 'attrib_name'
  ;
```

would be a good start. Writing the test, we have:

**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](mailto:kcarlson@tpj.com).

```
sub test_setter : Test {
  eval q{ package ScratchClass3;
    use Class::Builder
      has_constructor => 1,
      get_set => 'attr' };
  die $@ if $@;
  my $scratch = ScratchClass3->new;
  $scratch->set_attr(10);
  is $scratch->{attr}, 10, "attr successfully set";
}
```

This fails. Note that we're treating the class as a glass box and using our knowledge of how it is implemented to write our test. This is fine for the time being because it's the only way we can get the test written. We'll refactor the test once we have a better way of accessing the attribute. First, we concentrate on getting the test to pass:

```
sub import {
  my $class = shift;
  my %args = @_;
  my $calling_class = caller(0);
  no strict 'refs';
  if ($args{has_constructor}) {
    *{"${calling_class}::new"} = sub { bless {}, $_[0] };
  }
  if ($args{get_set}) {
    my $method = $args{get_set};
    *{"${calling_class}::set_${method}"} =
      sub { $_[0]->{$method} = $_[1] };
  }
}
```

This bar is green, but our implementation is as ugly as sin. It has some nasty repetition going on. I do not like `*{"${calling_class}::new"} = sub {...}` and `*{"${calling_class}::set_${method}"} = sub {...}`, for instance.

So, we put on our refactoring hat and slightly rewrite them:

```
if ($args{has_constructor}) {
  my $method = 'new';
  *{"${calling_class}::${method}"} = sub { bless {}, $_[0] };
}
if ($args{get_set}) {
  my $attr = $args{get_set};
  my $method = "set_${attr}";
  *{"${calling_class}::${method}"} =
    sub { $_[0]->{$attr} = $_[1] };
}
```

The bar is still green, and now we have some absolutely clear duplication. So, let's pull the duplicated behavior out to a new function and rejig import:

```
sub add_class_method {
  my($target_class, $method_name, $methodref) = @_;
  no strict 'refs';
  *{"${target_class}::${method_name}"} = $methodref;
}

sub import {
  my $class = shift;
  my %args = @_;
  my $calling_class = caller(0);
  if ($args{has_constructor}) {
    my $method = "new";

    add_class_method($calling_class, $method,
      sub { bless {}, $_[0] });
  }
}
```

```

}
if ($args{get_set}) {
    my $attr = $args{get_set};
    my $method = "set_$attr";
    add_class_method($calling_class, $method,
                    sub { $_[0]->{$attr} = $_[1] });
}
}

```

The bar is now green. Now that we've added the function call, we can ditch the *\$method* variables we added to make the duplication obvious. (I'm taking baby steps here to make the process explicit. Otherwise, I probably wouldn't have introduced them in the first place. However, in more complex situations, using variables as a way of explaining to yourself what the different bits of a method are doing can

---

*Our only goal when we  
have a failing test is  
to write just enough code  
to get a green bar*

---

be a very handy precursor to extracting sections of code into new methods.) Even after we've removed the useless variable, we still have some duplication. Let's pull the consequent actions from each of those conditionals out into their own methods:

```

sub import {
    my $class = shift;
    my %args = @_;
    my $calling_class = caller(0);
    if ($args{has_constructor}) {
        $class->has_constructor($calling_class,
                               $args{has_constructor});
    }
    if ($args{get_set}) {
        $class->get_set($calling_class, $args{get_set});
    }
}

sub has_constructor {
    my($class, $calling_class, $flag) = @_;
    if ($flag) {
        add_class_method($calling_class, 'new',
                        sub { bless {}, $_[0] });
    }
}

sub get_set {
    my($class, $calling_class, $attr) = @_;
    add_class_method($calling_class, "set_$attr",
                    sub { $_[0]->{$attr} = $_[1] });
}

```

Doing this makes it clear that there's more duplication in the *import* method to deal with. Look at the way the key used in the

conditional clause is repeated as the name of the called method. Let's refactor again:

```

sub import {
    my $class = shift;
    my %args = @_;
    my $calling_class = caller(0);
    foreach my $generation_method ( keys %args ) {
        $class->$generation_method(
            $calling_class, $args{$generation_method}
        );
    }
}

```

The only tricky part of that last refactoring was choosing the name for the loop variable. *\$generation\_method* is admittedly a little long winded, but it does express quite neatly what we're expecting to see.

The bar is still green, and there's no obvious duplication left in our code, so let's choose something else from our list of requirements.

One of the things we want to be able to do with our generated classes is to chain setting methods, so let's write a test for that:

```

sub test_chaining : Test {
    eval q{ package ScratchClass4;
           use Class::Builder
           has_constructor => 1,
           get_set => 'attr1',
           get_set => 'attr2' };
    die $@ if $@;
    my $scratch = ScratchClass4->new;
    $scratch->set_attr1(10)
        ->set_attr2(20);

    ok eq_hash $scratch, { attr1 => 10, attr2 => 20 };
}

```

As expected, we have a red bar, but it doesn't fail where we expected it to fail. It seems that we're not generating a *set\_attr1* method. A quick look at the *import* method shows the problem: We're using the wrong data structure. Hash keys are unique; therefore, instead of having two calls to *get\_set*, we only have a single call. So, let's recode the *import* method, replacing the loop through a set of hash keys with a loop through @\_:

```

sub import {
    my $class = shift;
    die "Arguments must be in pairs!" if @_ % 2;
    my $calling_class = caller(0);
    while (@_) {
        my($generation_method, $attrib) = splice @_, 0, 2;
        $class->$generation_method( $calling_class,
                                    $attrib );
    }
}

```

This still fails, but it fails as expected. Getting it to pass is simple—just alter the *get\_set* generation method:

```

sub get_set {
    my($class, $calling_class, $attr) = @_;
    add_class_method($calling_class, "set_$attr",
                    sub { $_[0]->{$attr} = $_[1]; $_[0] });
}

```

and the bar is green.

So, let's pick some more functionality and write another failing test.

I mentioned before that our tests are treating objects as glass boxes and using our knowledge about their internal representation. We can get rid of that assumption by rewriting our tests to depend on accessor methods (and, as a bonus, we get to test useful new functionality at the same time):

```
sub test_setter : Test {
  eval q{ package ScratchClass3;
    use Class::Builder
      has_constructor => 1,
      get_set => 'attr' };

  die $@ if $@;
  my $scratch = ScratchClass3->new;
  $scratch->set_attr(10);
  is $scratch->attr, 10, "attr successfully set";
}

sub test_chaining : Test(2) {
  eval q{ package ScratchClass4;
    use Class::Builder
      has_constructor => 1,
      get_set => 'attr1',
      get_set => 'attr2' };

  die $@ if $@;
  my $scratch = ScratchClass4->new;
  $scratch->set_attr1(10)
    ->set_attr2(20);

  is $scratch->attr1, 10;
  is $scratch->attr2, 20;
}
```

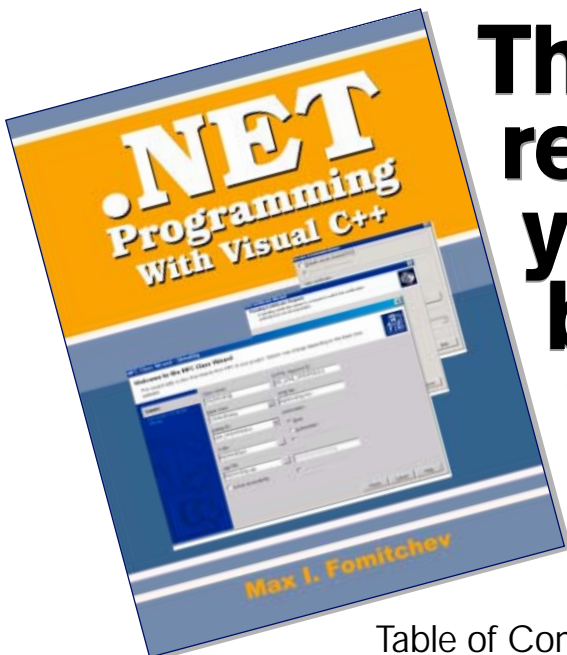
As expected, the bar goes red. So, let's have a look at the `get_set` generator method and change it to make an accessor method:

```
sub get_set {
  my($class, $calling_class, $attr) = @_;
  add_class_method($calling_class, "set_$attr",
    sub { $_[0]->{$attr} = $_[1]; $_[0] });
  add_class_method($calling_class, $attr,
    sub { $_[0]->{$attr} });
}
```

The bar is green, and this is as far as we'll take it in this article. We now have a helper module that will generate the kind of simple accessor methods that are the bread and butter of much OO development, and we have a simple set of tests that we can use to keep us sane as development proceeds. Note that the tests we've written aren't what could be described as comprehensive, but they're enough to have driven us to the point where we have a working module.

`Class::Builder` is not available on CPAN. At least, not yet. In this article, I've taken you to the point where it's beginning to be useful, but still a long way from release quality. I've also not provided the final source code, as this whole process was meant only as an exercise, and the module as it stands is just a starting point. I hope this article will help you get started using test-driven development to write your own modules.

TPJ



# The .NET resource you've been waiting for!



- Delivered in PDF format.
- Packed with C++ code examples.
- Thousands of lines of source code.
- A complete reference to the .NET Framework

Table of Contents and sample chapter available at:  
<http://www.ddj.com/dotnetbook/>

Get your copy now!

Available via **download** for just **\$19.95**  
or  
on **CD-ROM** for only **\$24.95** (plus s/h).

# A Magic Header for Starting Perl Scripts

Most Perl scripts start with a line beginning with `#!` and containing the word *perl* (for example, `#!/usr/local/bin/perl -w`). This line tells the UNIX operating system that it shouldn't treat the current file as a binary executable, but it should invoke the specified *perl* interpreter with the specified options, and that interpreter will take care of the Perl script. Although the concept of this first line is simple, writing it to be universally applicable tends to be very hard.

For example, specifying `#!/usr/local/bin/perl -w` will cause a confusing *my\_script.pl: No such file or directory* message on many out-of-the-box Linux systems, on which *perl* is located in `/usr/bin/`. Specifying `#!/usr/bin/perl -w` solves the problem on those Linux boxes, but will break compatibility with most Solaris systems, on which the standard place for *perl* is `/usr/local/bin/perl`. The bad news is that there is no standard place for *perl* to be specified after `#!`. To make matters worse, some older UNIX systems impose very strict rules on what can be specified in the `#!` line. As a result, you may choose between:

- Specifying the path that works for the majority of your user base.
- Documenting the incompatibility, and politely asking the users to manually modify your scripts if there's a problem.
- Writing an install script to automate the modifications. (How will the user start the install script? By typing *perl install.pl*, presumably.)
- Finding a solution that will work anywhere.

This article describes the last option—the Magic Perl Header, a multiline solution for starting a Perl script on any UNIX operating system.

## Common One-Line Pitfalls

The most obvious `#!` one-liners are just not good enough:

- Specifying `#!/usr/local/bin/perl -w -T` doesn't work because some UNIX operating systems allow only a single command-line switch after `#!`. This one-liner won't work on Linux.
- Specifying `#!/usr/local/bin/perl -wT` doesn't work because some UNIX operating systems expect a space after `#!`. This one-liner

---

*Péter is studying computer science and informatics at Budapest University of Technology and Economics, Hungary. His primary interests are programming languages, model verification, and computer typesetting. He can be contacted at [pts@inf.bme.hu](mailto:pts@inf.bme.hu).*

works on Linux only if `/usr/local/bin/perl` exists (but it usually doesn't).

- Specifying `#!/usr/local/bin/perl -wT` doesn't work because *perl* might be located in `/usr/bin` or somewhere else, such as on Debian Linux. (Remember: The user of your script may not be educated enough to be able to find the *perl* binary and modify the first line of the script accordingly; and in some security configurations, they may not have the permission to do it, even when they know exactly what to change.) This one-liner rarely works on out-of-the-box Linux.
- Specifying `#! perl -wT` doesn't work because some UNIX operating systems expect an absolute executable name (starting with `/`) after `#!`. This one-liner doesn't work on Linux.
- Specifying `#!/usr/bin/env perl -wT` doesn't work because some systems allow only zero or one argument after the command name. (Moreover, in some systems there is a limit for the overall length of the first line—it can be as few as 32 or 64 characters.) It would be very hard to specify the `-T` switch from anywhere other than the command line. (The `-w` switch is easier: just write `BEGIN{${W}=1}` in front of the Perl code.) The `-T` switch is a security switch, and specifying it too late opens the backdoor for malicious accidents. You (the programmer) should be extremely careful here, but it is difficult because there is no place to specify the correct switches. This one-liner doesn't work on Linux.
- Specifying `#!/usr/bin/env perl` doesn't work, either, because *env* might be missing or located somewhere else on some systems. This one-liner works on Linux.

## Building the Magic Perl Header

It is clear that there is no single-`#!`-line solution to the problem in the general case, because there is no portable way to start Perl to run a script. A multiple-line solution will be necessary. In this section, I will begin to build this solution. I will identify problems and limitations along the way, and in the next section, present the final, complete magic header that will allow you to start a Perl script on any UNIX system.

The only portable beginning for a script is:

```
#!/bin/sh
```

`/bin/sh` is available on all UNIX systems, but it might be a symlink to any shell, including Bourne shell variants (such as Bash

and ash), Korn shell variants (such as pdksh and zsh), and C shell variants (such as csh and tcsh). Many UNIX utilities, and the `libc system(3)` function (conforming to ANSI C, POSIX.2, BSD 4.3) rely on a working `/bin/sh`. So it is fairly reasonable to assume that `/bin/sh` exists and is a Bourne, Korn, or C shell variant. On Linux, `/bin/sh` is usually a symlink to `/bin/bash`. (On Linux install disks, it is sometimes a symlink to `/bin/ash` or the built-in ash of BusyBox.) On Win32 MinGW MSYS, `/bin/sh` is Bash, but there is no `/bin/bash`. On Solaris, `/bin/sh` is Sun's own simplistic Bourne-shell clone, and Digital UNIX also has a simple Bourne-shell clone in `/bin/sh`. (The line `#!/bin/sh --` that is seen in many shell scripts to allow arbitrary filenames for the executable won't work here because tcsh gives an error for the `--` switch.)

We can write a simple shell wrapper that will find the `perl` executable in `$PATH` and run it with the correct switches. In fact, this is the only way that this works on Win32 systems, using `.bat` batch files. A candidate for the solution is:

```
## file my_script.sh, version 1
#!/bin/sh
perl my_script.pl

## file my_script.pl
# real Perl code begins here
```

This has the following problems:

1. It doesn't pass command-line arguments.
2. It doesn't propagate `exit()` status.
3. It cannot find the Perl script on the `$PATH`—it will take it from the current directory, which is usually wrong, and might also present a security issue.
4. It needs two separate files.

Problems 1–3 can be overcome quite easily:

```
## file my_script.sh, version 2
#!/bin/sh
exec perl -S -- my_script.pl "$@"
```

All Bourne and Korn shells (such as GNU Bash, ash, zsh, pdksh, and Solaris `/bin/sh`) can interpret `my_script.sh` correctly. However, C shells use a different notation for “all the arguments passed to the shell, unmodified.” They use `$argv:q` instead of `"$@"`. The `perlrun(1)` manual page describes a memorable construct that detects the C shell:

```
eval '(exit $?)' && eval 'echo "Korn and Bourne"'
echo All
```

The message “All” gets echoed on all three shell types, but only Korn and Bourne shells print the “Korn and Bourne” message. (In zsh, the result depends on the value of `$_`, but it won't cause a problem since zsh understands both the csh and Bourne shell constructs we use.) The trick here is that `$_` is the exit status of the previous command, with the initial value of 0, but `$_?` in the C shell is a test that returns “1” because the variable `$_` exists.

We can change `echo` in the C shell detection code to `exec perl`, and that's it:

```
## file my_script.sh, version 3
#!/bin/sh
eval '(exit $?)' && exec perl -S - "$@" "$@"
exec perl -S -- "$@" $argv:q
```

Now we're ready to make our first wizard step: Combine `my_script.pl` and `my_script.sh` into a single file, which invokes it-

self using `perl` when run from the shell. (Forget about csh-compatibility for a moment—we'll get to that later.)

A simple attempt would be:

```
#!/bin/sh
eval 'echo DEBUG; exec perl -S $0 ${1+"$@"}'
if 0;
# real Perl code begins here
```

Unfortunately, it doesn't run the real Perl code, but it produces an infinite number of DEBUG messages. That's because Perl has a built-in hack: If the first line begins with `#!` and it doesn't contain the word `perl`, Perl executes the specified program instead of parsing the script. See the beginning of the `perlrun(1)` manual page for further details.

In the following simple trick, suggested by the `perlrun(1)` manual page, we include the word `perl` in the first line:

```
#!/bin/sh -# -*- perl -*-
eval 'exec perl -S $0 ${1+"$@"}'
if 0;
# real Perl code begins here
```

This fails to work on many systems, including Linux, because the OS invokes the command line (`/bin/sh, -- # *- * perl *- , ./my_script.pl`), and the shell gives an unpleasant error message about the completely bogus switch.

So we can omit the first line:

```
eval 'exec perl -S $0 ${1+"$@"}'
if 0;
# real Perl code begins here
```

This solution is inspired by Thomas Esser's `epstopdf` utility, and it seems to work on Linux systems with both `perl my_script.pl` and `./my_script.pl`. But we can do better. The major flaw in this script is that it relies on the fact that the operating system recognizes executables beginning with ASCII characters as scripts, and runs them through `/bin/sh`. On some systems, a “Cannot execute binary file” or “Exec format error” may occur.

Note that this script is quite tricky since the first line is valid in both Perl and Bourne-compatible shells. (It doesn't work in the C shell, but we'll solve that problem later on.)

The solution has another problem: If someone gives the script a weird filename with spaces and other funny characters in it, such as:

```
-e system(halt)
```

then the command

```
perl -S -e system(halt)
```

will be executed, which is a disaster when there is a dangerous program named `halt` on the user's `$PATH`. This problem can be solved easily, by quoting `$0` from the shell, and prefixing it with `--` to prevent Perl from recognizing further options.

We have two conflicting requirements for the `#!` line: The portability requirement is that it must be exactly `#!/bin/sh`; but it must contain the word `perl` to avoid the infinite DEBUG loop described earlier. There is no single line that can satisfy both of these requirements, but what about having two lines, then running `perl -x`, so the OS will parse the first and Perl will find the second?

```
#!/bin/sh
eval 'exec perl -S -x - "$@" ${1+"$@"}'
```

```

if 0;
#!/perl -w
# real Perl code begins here

```

The trick here is that Perl, when invoked with the `-x` switch, ignores everything up to `#!/perl`. Users of nonUNIX systems should invoke this script with `perl -x`. UNIX users may freely choose any of `perl my_script.pl`, `perl -x my_script.pl`, `./my_script.pl`, and even `sh my_script.pl`.

The subtle bilingual tricks in this script are worth studying. When the file is read by `perl -x`, it quickly skips to the real Perl code. When the file is read by the shell, it executes the line with `eval`: it calls `perl -x` with the script filename and command-line arguments. The double-quotes and `$@` are shell script wizardry, so things will work even when arguments contain spaces or quotes. The `-S` option tells Perl to search for the file in `$PATH` again because most shells leave `$0` unchanged (i.e., `$0` is the command the user has typed in).

Although the second and the third lines contain valid no-op Perl code, Perl never interprets these lines because of the `-x` switch. These lines are also completely ignored by `perl my_script.pl` because that immediately invokes `/bin/sh`. However, when the user loads this script with the `do` Perl built-in, the second and third lines get compiled and interpreted, a harmless no-op code is run, and no syntax error occurs.

There are still deficiencies that remain:

- It doesn't work in the C shell. We have already solved this earlier in this section.
- It reports line numbers in error messages relative to the `#!/perl -w` line.
- It prints warnings when locale settings are invalid. (Try setting `'export LANG=invalid'` in Bash before running the script to see the ugly warning messages.)

With regard to the line number problem, the `do` Perl built-in can be used to reread the script with a construct like this:

```

BEGIN{ if(!$second_run){ $second_run=1; do($0); die
    $@ if $@; exit } }

```

`BEGIN` is required here to prevent Perl from compiling the whole file and possibly complaining about syntax errors with the wrong line numbers. The `die $@ if $@` instruction will print runtime error messages correctly. See `perlvar(1)` for details about `$@`. Unfortunately the code

```

BEGIN{ if(!$second_run){ $second_run=1; do($0); die
    $@ if $@; exit } }
die 42;

```

yields an extra error message “BEGIN failed—compilation aborted.” This error is confusing because `die 42` causes a run-time error, not a compile-time error. To get rid of the message, we should eliminate `exit` somehow, and tell Perl not to continue parsing the input after `}}`. We'll use the `__END__` token to stop parsing early enough.

The locale warning is a multiline message starting with “perl: warning: Setting locale failed.” Perl emits this if the locale settings specified in the environment variables `LANG`, `LC_ALL`, and `LC_*` are incorrect. See `perllocale(1)` for details. The real fix for this warning is installing and specifying locale correctly. However, most Perl scripts don't use locale anyway, so a broken locale doesn't do any harm to them.

Although Perl is a good diagnostics tool for locale problems, most of the time we don't want such warning messages, especially not in CGI (these warnings would fill the web server's log file), or some system daemon processes, when the program is prohib-

```

#!/bin/sh
eval '(exit $?)' && eval 'PERL_BADLANG=x;PATH="$PATH.";export PERL_BADLANG\
;exec perl -T -x -S -- "$0" ${1+"$@"};#if 0;eval 'setenv PERL_BADLANG x\
;setenv PATH "$PATH";exec perl -T -x -S -- "$0" $argv;q;#.q
#!/perl -w
+push@INC, '.'; $0~/(.*)/s;do(index($1,"/")<0?"/$1":$1);die@if$@__END__+if 0
;#Don't touch/remove lines 1--7: http://www.inf.kme.hu/~pts/Magic.Perl.Header
# real Perl code begins here

```

Example 1: The Magic Perl Header.

ited from writing to `stderr` on normal operation. The system administrator should really fix locale settings, but that can take time. Most users don't have time to wait weeks to run a single Perl script that doesn't depend on locale anyway.

The `perllocale(1)` man page says that `PERL_BADLANG` should be set to a true value to get rid of locale warnings. Actually, `PERL_BADLANG` must be set to a nonempty, nonnumeric string (for example, `PERL_BADLANG=1` doesn't work). So we'll set it to `PERL_BADLANG=x` in the shell script section. Note that this has no effect if Perl is invoked before the shell. For example, `perl`, `perl -x`, `perl -S`, and `perl -x -S` all emit the warning long before the shell has a chance to change `PERL_BADLANG`.

## The Finished Header

Combining it all together, we have the final version of the Magic Perl Header; see Example 1. The file should have the executable attribute on UNIX systems.

This header is valid in multiple languages, so its meaning depends on the interpreter. Fortunately, the final effect of the header in all interpreters is that `perl` gets invoked running the real Perl code after the header. Let's see how the header achieves this:

- When executed with `perl`, without the `-x` switch, Perl runs `/bin/sh` immediately. (`/bin/sh` may be any type of shell.)
- Bourne and Korn shell variants interpret the file as:

```

#!/bin/sh -
true && eval '...; exec perl -T -x -S "$0" ${1+"$@"}' # comment
garbage

```

So they run `'perl -x'`.

- C shell variants interpret the file as:

```

#!/bin/sh --
false && eval '...' ;
eval '...; exec perl -T -x -S - "$0" $argv;q' # comment
garbage

```

So they run `'perl -x'`.

The backslash at the end of the second line of the header seems to be superfluous, but it isn't because `csh` doesn't allow the breaking of the line in the midst of the string without a backslash.

- The operating system runs the file by running `/bin/sh`, some shell variant. This is true even for ancient systems that don't know about the `#!/hack`, but just treat ASCII files as shell scripts.
- `perl -x` interprets the file as:

```

#!/perl -w
untaint $0; do $0; die $@ if $@; __END__
garbage

```

So it runs the current file again, with `do`, not respecting the `#!/` lines. This is a good idea to make error line numbers come out correctly.

The only way to untaint a value is regexp subexpression matching. We use it in `$O=~/(.*)/s`.

- `do $O` treats the file as:

```
eval 'garbage' if 0;
eval 'garbage' . q+garbage+ if 0;
# real Perl code
```

`do $O` doesn't consult `$ENV{PATH}` for the location of the script (it iterates over `@INC`), but by the time `do $O` is invoked, `$O` already has the relevant component of `$ENV{PATH}` prepended to it if a path search was done, so `@INC` won't be examined here. Note that `$O` may be a relative pathname, but this isn't a problem since `chdir()` was not called since the path search. Without the `index` function in the script, `do` would have looked at `@INC` and found the Perl built-in `ftp.pl` instead of our magic script named `ftp.pl` when calling `perl -x ftp.pl` in the current directory.

- The real Perl code is compiled only once because the previous read (invoking `do $O`) has finished compilation at the `__END__` token. The real compilation bypasses the `__END__` token because it is part of the single-quoted string `q+garbage+`.
- Error line numbers are reported correctly because compilation occurs inside `do`, which ignores `#!`. Both compile-time and runtime errors, including manual calls to `die()`, are caught and reported early by the `die $@ if $@` statement. Each error is reported only once because the real Perl code is compiled once.
- The real code may contain any number of `exit()`, `exec()`, `fork()`, and `die()` calls, and they will work as expected. `return` outside a subroutine is fortunately disallowed in pure Perl, so we don't have to treat this case.
- `push@INC, "."` is required by `perl 5.8.0 -T`.

So the real Perl code gets executed, even on old UNIX systems, no matter how the user starts the program. The header is suitable for inclusion into CGI scripts. (In nonCGI programs, where extreme security is not important, occurrences of the `-T` option can be removed.)

All of the following work perfectly, without the locale warning:

```
DIR/nice.pl          # preferred
ash DIR/nice.pl
sh DIR/nice.pl
bash DIR/nice.pl
csh DIR/nice.pl
tcsh DIR/nice.pl
ksh DIR/nice.pl
zsh DIR/nice.pl
```

The following invocations are fine:

```
perl -x -S DIR/nice.pl # locale-warning
perl DIR/nice.pl       # locale-warning
perl -x DIR/nice.pl   # locale-warning
perl -x -S nice.pl    # locale-warning; only if on
                        # $PATH, recommended on Win32
perl nice.pl          # locale-warning; only from curdir
perl -x nice.pl       # locale-warning; only from curdir
nice.pl               # only if on $PATH (or $PATH contains '.')
```

The following don't work, because buggy Perl 5.004 tries to run `/bin/sh -S nice.pl`:

```
perl -S nice.pl       # doesn't work
perl -S DIR/nice.pl  # doesn't work
```

Of course, there is a noticeable performance penalty: `/bin/sh` is started each time the script is invoked. This cannot be complete-

ly avoided because `PERL_BADLANG` has to be set before `perl` gets invoked. After the shell has finished running, one line of helper Perl code is parsed (after `#!/perl`), and the `do` causes five lines of helper code to be parsed. The time and memory spent on these six lines is negligible. So the only action that slows script startup is the shell. If the user sets and exports `PERL_BADLANG=x`, fast startup is possible by calling:

```
perl -x -S nice.pl
perl -x DIR/nice.pl
```

In a Makefile, you should write:

```
export PERL_BADLANG=x
goal:
    perl -x DIR/nice.pl
```

The command-line options `-n` and `-p` would fail with this header. This is not a serious problem because `-n` can be implemented as wrapping the code inside `while (<>) { ... }`, and `-p` can be changed to the wrapping `while (<>) { ... } continue { print }`.

## Header Wizard

I've implemented a Header Wizard that automatically adds the Magic Perl Header to existing Perl scripts. The Header Wizard is available from <http://www.inf.bme.hu/~pts/Magic.Pperl.Header/magicc.pl.zip>. [For convenience, we have also posted this at <http://www.tpj.com/source/>, though downloading from the author's site guarantees that you get the most recent version. -Ed.]

The easy recipe for the universally executable Perl script:

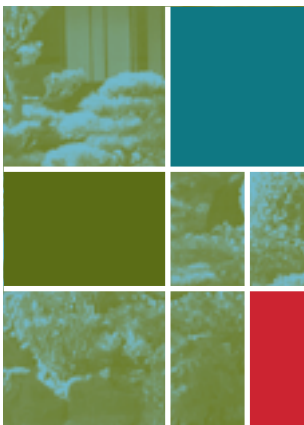
1. Write your Perl script as usual. You may call `exit()` and `die()` as you like.
2. Specify the `#! ... perl` line as usual. You may put any number of options, but the `-T` option must either be missing or specified alone (separated with spaces). Example: `#!/dummy/perl -wi.bak -T`. See `perlrun(1)` and `perlsec(1)` for more information about the `-T` option.
3. Run `magicc.pl` (the Header Wizard), which will prepend an eight-line magic header containing the right options to the script, and it will make the script file executable (with `chmod +x ...`). (The `-T` option will be moved after both `exec perls`, and other options will be moved after `#!/perl` because Perl looks for switches only there.)
4. Run your script with a slash, but without `sh` or `perl` on the command line. For example: `./my_script.pl arg0 arg1 arg2`. After you have moved the script into `$PATH`, run it just as `my_script.pl arg0 arg1 arg2`. (This avoids the locale warnings and makes options take effect.) Should these invocations fail on a UNIX system for whatever reason, please feel free to e-mail me. As a quick fix, run the script with `perl -x -S ./my_script.pl arg0 arg1 arg2`.
5. Note that on Win32 systems, `perl -x -S` is the only way to run the script. You may write a separate `.bat` file that does this.
6. Tell your users that they should run the script the way described in Step 4. There is a high chance that it will work even for those who don't follow the documentation.

## Conclusion

For such a widely implemented language, Perl can be surprisingly hard to invoke reliably on a variety of platforms. I hope this Header Wizard helps you to write Perl scripts that will start with a minimum of fuss on just about any system.

TPJ





# Building a Better Mail Handler

*Simon Cozens*

Long-time readers of my columns will know that I have two particular interests when it comes to Perl programming—mail handling, as evidenced by my articles on *Mail::Miner* and *Mail::Audit*, and also making things as simple as possible for the programmer, but no simpler.

Until recently, I have to admit, these two interests have been a little at war with one another because, as it turns out, mail handling in Perl is anything but simple.

This is quite a shame because mail handling as an abstract concept is very simple indeed. Nine times out of ten, you want to look at a piece of mail, get or set some of its headers, and look at its body. And that's it. Unfortunately, this abstract concept turns out to be anything but simple when it's turned into reality.

Let's first look at the options available, and then what I've proposed to do about them. In the process, we'll see what lessons we can learn about object-oriented (OO) and module design, and the way I've approached the redesign of some of my own modules. This means that this article will turn out rather more philosophical than practical, but that's OK. I promise I'll make up for it next time.

## *Mail::Internet*

There are two main mail message handling libraries in Perl. The most commonly used of these is *Mail::Internet*, and it's not so horrendous to use:

```
use Mail::Internet;
my $mi = Mail::Internet->new([split /\n/, $mail]);
print $mi->as_string;
```

Each message object has an associated *Mail::Header* object, and you can get headers by looking at that:

```
my $from = $mi->head->get("From");
```

Wait a minute—why is this? The *Mail::Header* object is not entirely useful on its own; it's only really useful in the context of the mail it comes from. What's happened here is that an implementation decision—putting header parsing and handling into its own class—has leaked out into the user interface to the module. I shouldn't need to care how the header handling is imple-

---

*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 pumpking. Simon can be reached at [simon@simon-cozens.org](mailto:simon@simon-cozens.org).*

mented. Getting headers is, as far as I'm concerned, just a part of looking at mail. Sure, you get a bit of extra flexibility by this implementation decision, but not enough to warrant exposing it to the user.

If you do want to do it this way, using a separate *Mail::Header* object, that's fine. You can hide the implementation decision from the user by means of OO delegation. What's really going on here is that *Mail::Internet* has what's called a HAS-A relationship with *Mail::Header*.

Unlike the usual vertical IS-A relationships, HAS-A relationships are horizontal. *Mail::Internet* doesn't inherit from *Mail::Header*, or the other way around, but it contains one, encapsulates it, and uses it within itself.

When you see a HAS-A relationship, you also often see delegation. Delegation is an OO principle by which you direct methods through a HAS-A relationship: We should be able to call something like *head\_get* on *Mail::Internet* and it should pass the request onto the *Header* that it has. Again, this avoids exposing the implementation detail of the *Mail::Header* class in the first place. Sadly, *Mail::Internet* doesn't support delegation.

But let's back up a step. Why bother having a separate *Mail::Header* anyway? This was supposed to be a simple problem. Before we move on to looking at the other solution, let's just tot up a quick score for *Mail::Internet*; see Table 1.

## *Mail::Message*

The main competitor to *Mail::Internet*, at least until a couple of days ago, was *Mail::Message*, written by Mark Overmeer—who also, coincidentally, now maintains *Mail::Internet*. This is part of the *Mail::Box* suite of libraries.

If you think that *Mail::Internet* was overkill, then you probably want to avert your eyes. *Mail::Box* is a full-featured mail handling suite, comprising around 90 full-featured classes and over 14,000 full-featured lines of code.

*Mail::Message* similarly splits off header handling to *Mail::Message::Head*, but does provide some delegate methods that access fields. These return *Mail::Message::Field* objects in most cases, but sometimes return *Mail::Address* objects in the case of headers such as "From," "Cc," and so on. These two classes magically stringify to the value you're expecting if used in string context.

NAME	CLASSES	LINES OF CODE
<i>Mail::Internet</i>	2	1978

Table 1: Code tally for *Mail::Internet*.

All of these classes inherit from a single *Mail::Reporter* class that handles error reporting, and some of these classes have special subclasses that are used to “lazy-load” for speed. For instance, the header can originally be returned as a *Mail::Message::Head::Delayed*, which doesn’t do any parsing of the header, and this is then turned into a parsed *Mail::Message::Head::Complete* when a field is requested. Speed is very important in the design of this module, which explains in part why it is so horrendously slow compared to the much simpler *Mail::Internet*.

For example, I benchmarked reading an e-mail into a *Mail::Message* and *Mail::Internet* object, respectively, and retrieving its “From” header:

```
Benchmark: timing 10000 iterations of internet, message
internet: 59 wallclock secs (58.85 usr + 0.02 sys =
58.87 CPU) @ 169.87/s (n=10000)
message: 122 wallclock secs (117.97 usr + 0.41 sys
= 118.38 CPU) @ 84.47/s (n=10000)
```

And to be fair, in another test, I read in an e-mail and spat it back out as a string:

```
Benchmark: timing 10000 iterations of internet,
message, simple...
internet: 60 wallclock secs (59.17 usr + 0.05 sys =
59.22 CPU) @ 168.86/s (n=10000)
message: 128 wallclock secs (124.27 usr + 0.54 sys
= 124.81 CPU) @ 80.12/s (n=10000)
```

There’s an important lesson here. The object-oriented model is good, like vintage wine is good. But if you drink several gallons of vintage wine in a sitting, you’re liable to end up getting a little confused. You end up with what’s called “lasagna code,” the object-oriented equivalent of spaghetti code; your inheritance tree becomes so towering it’s nearly impossible to tell which classes you’re really using and where their methods are coming from.

Then you find yourself having to optimize your code, which is by now extremely complex, by adding more complexity, whereas the rules of optimization tell you that you optimize by taking complexity away.

Our example of loading up a message and looking at its “From” header, which took twice as long as the *Mail::Internet* version, used the following Perl modules: *Mail::Address*, *Mail::Box::Parser*, *Mail::Box::Parser::Perl*, *Mail::Message*, *Mail::Message::Body*, *Mail::Message::Body::File*, *Mail::Message::Body::Lines*, *Mail::Message::Body::Multipart*, *Mail::Message::Body::Nested*, *Mail::Message::Construct*, *Mail::Message::Field*, *Mail::Message::Field::Fast*, *Mail::Message::Head*, *Mail::Message::Head::Complete*, *Mail::Message::Part*, and *Mail::Reporter*, for a total score shown in Table 2.

Ouch.

### **Email::Simple**

Let’s go back to solving the nine-out-of-ten case: getting the body and setting the headers. Once we’ve got this case nailed down, then we can start adding complexity. I’m not a massive fan of Extreme Programming, but one of its doctrines is that you start as simply as possible, and only add more complex functionality when you need it. I like that idea. With Perl modules, as with writing, the time to stop is not when there is nothing more to add but when there is nothing more to take away.

So I decided to set out and reinvent the mail-handling wheel but at least try to make it less square this time and forgo the ornamental carvings, fuzzy dice, and the attachment for getting stones out of horse’s hooves. I wanted to write a Perl mail-handling library that was stunningly simple in every way, even at the cost of a little flexibility later.

NAME	CLASSES	LINES OF CODE	SPEED
Mail::Internet	2	1978	59s
Mail::Message	16	4394	122s

Table 2: Code tally including *Mail::Message*.

In a fit of pique, I decided that the whole *Mail::\** namespace was rotten to the core (especially the bits of it that I wrote), and if we were going to have a fresh start at mail handling, we should start afresh in a different namespace. Sometimes heresy is an important part of innovation.

I started the design of *Email::Simple* by working out what methods I would need. I came up with six, which I still think is probably too many. We want to create a new object; we want to get and set a header; we want to get and set the body; we want to be able to output the whole mail as a string again.

After a lot of consultation and argument with peers in the Perl community, I decided upon having separate accessor and setter methods: I could have cut my method count down to four without this—a pleasing thought—but I would lose a lot in the process.

First, I wanted to stick to UNIX design principles: Small, single-purpose tools. Every module, every method, every line of code, should do one thing and do it well. Having combined accessors smacks of doing two things. It loses regularity because the same method does different things depending on whether or not you give it an argument, and it loses symmetry.

Second, constructing and examining mail are usually two very distinct operations. Generally, you’re either examining existing mail or making up new mail. These are separate concepts that deserve not to be confused, and hence have separate methods to distinguish them.

In the same way, while it is possible to use *Email::Simple* to create a new e-mail from scratch, this is discouraged. Creating mail is a separate action and needs a separate module. Small, single-purpose tools.

This also led me to rethink how I would lay out the code into subroutines; again, I tried to think of subroutines as small, single-purpose tools that do one thing and do it well. This means that most of the subroutines in *Email::Simple* are four or five lines long, and must fit in one screen in the absolute worst case.

I also decided that *Email::Simple* was doing such a fundamental and simple task that it should not use any external modules. Not because I’m not a fan of code reuse—this is a library after all!—but because I wanted to minimize dependencies, making this portable, easy to install, and easy to use. In the end, I caved in while writing the *as\_string* method and used the core module *Text::Wrapper* to fold long header lines. Pragmatism must beat principles every time.

As well as being simple to implement, it’s fairly important that this module is simple to use. By trimming down the number of classes and methods to the bare minimum, I think I’ve achieved this:

```
my $es = Email::Simple->new($email);
print "From ", $es->get_header("From");
print "\n\n";
print $es->body;
```

The e-mail is expected to be a single scalar; the other modules can happily take a glob, an array of lines, an *IO::File* object, and who knows what else. I chose not to do that because another design principle for this project is predictability. “Do What I Mean” is very useful when it does do what you mean, but causes all sorts of fun when it doesn’t.

While many of my other modules are perfectly happy to try their hardest to work out what you actually mean, and do that, this

one is different. In my opinion, high-level magic belongs in high-level modules, not in fundamental libraries like this one. Clever is for high-level stuff; low-level modules should be as dumb as possible.

For the same reasons, *Email::Simple* objects don't automatically stringify, or indeed do anything without you asking specifically for it. If you use an *Email::Simple* object, you know exactly how it's going to behave in every situation. (Some languages call this the "principle of least surprise," and in my opinion, overloading goes against this in most cases—you don't expect random objects to stringify, so they should avoid doing so unless there's a very good reason.)

The minimalist interface and the defined behavior standards are small enough to fit inside your head. Ideally, you should only need to read the *Email::Simple* manual page once.

What about speed? I'm a fervent, passionate believer that if you follow the design principles I've outlined, with good, clean algorithms and simple design, you won't need to worry about speed; it'll just fall neatly out. The best way to optimize is to remove complexity, not to add it, and if you design your code to have very little extraneous complexity anyway, you'll find your modules already optimized!

And, in fact, that's how it turns out—because *Email::Simple* is so simple, because it does nothing extraneous, and because it's cleanly designed, it's very, very fast:

```
Benchmark: timing 10000 iterations of internet,
  message, simple...
internet: 59 wallclock secs (58.85 usr + 0.02 sys =
 58.87 CPU) @ 169.87/s (n=10000)
message: 122 wallclock secs (117.97 usr + 0.41 sys
 = 118.38 CPU) @ 84.47/s (n=10000)
simple: 9 wallclock secs ( 9.17 usr + 0.00 sys =
 9.17 CPU) @ 1090.51/s (n=10000)
```

Naturally, it's much faster than the other modules because it only does a single job and does it well; but it's the job that most users of these modules will want to be doing. Oh, and to total it up, see Table 3.

I apologize for it containing so many lines of code, but I wanted to quote extensively from RFC2822 in the comments, to help keep it standard compliant during whatever small amount of maintenance it might need. But nevertheless, I think we have a winner.

## *Email::LocalDelivery*

Now we have a foundation module, and we can start to build on this foundation.

I've done a lot of ragging on other people's code in this article, so the next module I wanted to raze to the ground and replace was one of my own: *Mail::LocalDelivery*.

*Mail::LocalDelivery* grew organically out of the *Mail::Audit* mail filter; one of the major aspects of *Mail::Audit* is that it delivers mail into your mailboxes. Once upon a time, this was a relatively simple process, with a bit of locking, and only a few lines of code, and it worked fine inside of *Mail::Audit*.

But then I made a stunningly stupid mistake, one I hope to never repeat. I accepted a patch that added *maildir* support. OK, that wasn't the mistake—the mistake was that I didn't then take the opportunity to refactor the code before adding the patch. I just added in an *if* statement that separated *maildir* from *mailbox*, and

NAME	CLASSES	LINES OF CODE	SPEED
Mail::Internet	2	1978	59s
Mail::Message	16	4394	122s
Email::Simple	1	239	9s

Table 3: Code tally including *Email::Simple*.

the code continued to grow organically again. And grow, and grow...

Before I realized what was going on, *Mail::Audit*'s *accept* method was the bulk of the module and was almost impossible to follow. So a golden lesson was learned there: Once your code grows two separate ways to achieve a task, always modularize at that point, at the very least.

And while you're about it, take the opportunity to see if you can split the whole thing off to a separate module or procedure. One of the design principles I've learned from my wise boss is that if it looks like a problem is getting complex, try adding another layer of abstraction. It's a very generic rule, but it can help this sort of situation. (Of course, it must be coupled with another principle: If it doesn't look like your problem is going to get complex, don't add another layer of abstraction, or you end up with *Mail::Box*. We're still trying to keep things as simple as possible, but no simpler.)

I eventually came to my senses when I was reminded that local delivery was a useful thing to be doing outside of the context of *Mail::Audit*, and I split out the *accept* logic to a new module, *Mail::LocalDelivery*. But I still had the problem that the multiple delivery methods weren't abstracted out at all, and although I planned to rewrite *Mail::Audit*'s *accept* method in terms of *Mail::LocalDelivery*, to tell the truth, I was scared to do so because the code had grown too hairy and I didn't know what would break if I touched it. And *Mail::Audit*, which handles every piece of mail I receive, is rather important to me. I didn't want to touch it if things might break.

So I took the same design principles that guided me in *Email::Simple* and applied them to the local delivery problem. I was very happy when designing this module since I could reduce the number of methods down to one—*deliver* will deliver a message to a bunch of mailboxes.

Why, then, did I make it OO anyway? Surely if you're just providing a single function, you could export it. Well, I could, but I wanted to implement it in terms of methods that could be inherited from, in case someone wanted to come along with a more featureful version in the future.

This is something that has to be learned the hard way—although I've been claiming that you don't unnecessarily abstract things out before a definite need arises, the decision whether or not to go OO is something that must be done right at the start of your planning: For one thing, you will want to avoid changing your user interface from procedural to OO style as a result of changing your implementation, something we talked about with respect to delegation.

It seemed appropriate to have a single front-end that delivered a message, but to use separate back-end modules to implement *mailbox*, *maildir*, and other delivery mechanisms. Again, we're keeping the implementation details away from the interface.

*Mail::LocalDelivery* was not badly designed. It had, for instance, separate *deliver\_to\_mbox* and *deliver\_to\_maildir* back ends, which called a unified *write\_message*, and so on. So I tried to keep this design in my rewrite.

Unfortunately, I learned something that you can only learn when you rebuild something from scratch: Most of the "shared code" in the unified method wasn't actually shared at all. The power of *write\_message* was that it handled locking a mailbox, opening it for append, writing the message to the end, unlocking, and so on. But mailbox delivery doesn't do any of these things: There's no need for locking and mailboxes are written from scratch, not appended to. The original *write\_message* used arguments and *if* statements to decide whether to lock or unlock, which essentially removed any of the abstraction that it was designed to provide. Oops.

It turned out that there was nothing to be shared between the various back ends anyway. This made me happy. Inheritance is a

good thing when used sparingly, or to extend a given class with replaced or additional functionality. When used to excess, it can lead to the unhappy situation where you need to grovel through five or six levels to find out where a given method is defined in order to debug or understand it. Walking a class inheritance tree is a great thing for a computer to do, but not so great for a human.

Once again, the principle of building one to throw away is vindicated: By rewriting the module, I managed to remove a load of extraneous code, and even though it's now split across multiple back-end modules, it weighs-in much lighter than the original *Mail::LocalDelivery*, and it's much, much easier to understand. All the accumulated cruft that had grown organically on the module got swept away, and this cut down the line count and the complexity of the code dramatically:

The old version:

```
% wc -l LocalDelivery/LocalDelivery.pm
534 LocalDelivery/LocalDelivery.pm
```

The new version:

```
% wc -l LocalDelivery.pm LocalDelivery/*.pm
77 LocalDelivery.pm
86 LocalDelivery/Maildir.pm
70 LocalDelivery/Mbox.pm
233 total
```

Less than half the size.

### **Email::Filter**

Inspired by this, I felt it was finally time to reinvent the venerable *Mail::Audit*.

In the same way as *Email::Simple*, I wanted to cut out as much of the extraneous functionality as possible, and pare it to the essentials. This means that *Email::Filter* doesn't support any logging, any local options, or anything of the like.

However, there's a problem here—these things, especially logging, are actually very useful. We want to be able to support them somehow. One possibility was to design the module to be easily subclassable, so that people would be able to write *Email::Filter::Logging* very easily, and then it'd be up to the end user.

But subclassing's a pain; I don't want to be writing a new class to accommodate my particular foibles about what logging should be done and how. There ought to be a nice, easier way to allow a user to customize a class's behavior. Thankfully, there is, and it's implemented by the *Class::Trigger* module. This provides simple, inheritable “trigger points” where the end user can attach callbacks. For instance, if I say

```
$item->add_trigger( ignore => sub { log("This mail
was ignored") } );
```

then the subroutine will be called every time the *ignore* method is invoked. All I need to do in my class is to use *Class::Trigger* and this gives me a new method to call in my own class:

```
sub ignore {
    my $self = shift;
    $self->call_trigger("ignore");
    ...
}
```

But wait—where did these two methods come from? We don't inherit from *Class::Trigger*, we only use it. What's happening is that *Class::Trigger* imports the methods into the caller's namespace, just like an ordinary, non-object-oriented method. We gain the two methods, as do any of our subclasses, but we don't have

an inheritance dependency on *Class::Trigger* itself. This technique is called using a “mix-in,” and it's quite a popular one in Ruby and Python for adding functionality to a class without inheritance.

In a bizarre way, I've found that some of the restrictions in *Email::Filter* caused by trying to make it as simple as possible have led to interesting ways of doing things that I wouldn't have otherwise considered. For instance, in *Mail::Audit*, there's a *pipe* method that dispatches the mail off to an external program. I modified this in *Email::Filter* so that it returns the standard output from the program. This means that, for instance, in the absence of direct support for *Mail::SpamAssassin*, you can say:

```
$mail->simple(Email::Simple->new($mail->pipe("spam
assassin")));
```

---

*You end up with what's called  
“lasagna code,” the  
object-oriented equivalent of  
spaghetti code*

---

In other words, pipe the mail to the *spamassassin* command-line command, which outputs a marked-up mail message; take this mail message and turn it into an *Email::Simple* object, and then use that as the underlying object for the *Email::Filter* entity.

*Email::Filter* currently weighs in at 255 lines of code compared to *Mail::Audit*'s 1053, but to be fair, this is because *Email::Filter* isn't anywhere near finished yet.

### **The Future of Email**

*Email::Simple* and *Email::LocalDelivery* are currently released on the CPAN; I wanted to hold them back until I had finished a few more modules in the *Email::\** project, but I got a few requests for them to be released early and, to be honest, there were a couple of bug reports in *Mail::LocalDelivery* that I couldn't be bothered to fix.

Right now, I'm working on *Email::Filter*, and after that will be the next big challenge—*Email::MIME*. This will be a relatively high-level library, but built around *Email::Simple* and many of its design principles, including, of course, simplicity; it's planned that this will be another “reader” module, concentrating on separating out the parts of a MIME message, rather than creating a new one from scratch. As such, it'll probably only add one or two methods to *Email::Simple*—“parts” to return a list of attachments in some format, and probably something to get some additional MIME-related metadata about the message.

After these reader modules are done, it will be time to start the creator modules, *Email::Creator* and *Email::MIME::Creator*. Who knows what I'll be working on after that; I have my beady eye on *Mail::SpamAssassin*. (People keep telling me that *LWP* is in dire need of a rewrite, but there just isn't enough time and coffee in the world.)

But I'd like to commend my design principles to you: simplicity, single-purpose tools, predictability, and not being afraid to sacrifice a little bit of functionality to achieve the nine-out-of-ten-cases solution. As we've seen, they lead to clean, maintainable code, that often turns out to be quite a lot faster than the all-encompassing solution anyway.

TPJ



# Tailing Web Logs

*Randal Schwartz*

One of the most boring tasks I perform as a system administrator is watching a log file. It's like watching a pot of stew boil. Or ice melt.

But watching a log file is also a great way to get a quick view of the activity on the system. For example, watching an Apache access log is great to see how “bursty” the traffic can be. But unless you're staring at the `tail -f` on the log file constantly, or you can pick out the timestamps of each logged line and do the math in your head quickly, you can't really tell which lines were “together” in a particular burst.

So, one day recently I got the idea for a “web tail” program. A browser would fetch data from a custom web server, which would be watching the log file, noting the arrival time of each line. The display would show the last, say, 16 lines of log, but colorized in such a way that I could see the age of each line at a glance. That way, lines that arrived at similar times would have a similar color coding.

And then I stumbled across two pieces of the Perl Object Environment (POE) package that made writing such a program easier. POE is a great collection of tools to manage events and threaded Perl code, good for servers, clients, and event-based glue. POE is described in more detail at <http://poe.perl.org/>.

The results of all this are shown in Listing 1.

Line 1 is the standard hash-bang line for Perl on my system. Obviously, this will need to be changed depending on where Perl is installed.

Line 2 turns on the standard restrictions: Symbolic references are disabled, implicit package variables are disabled (forcing most variables to be declared with `my`), and “bare words” are disabled. This is always a good thing for programs exceeding a dozen lines or so.

Lines 5 through 8 are the tweakable parts of this script. The `$FILENAME` gives the file to be followed. In this case, it's the Apache access log for my web server front-end reverse proxy. As I was testing this script, I found that to be a nice source of bursty data.

The `$TAILING` variable controls how many lines of the log file are visible. Too little, and we lose context. Too much, and we scroll too often.

The `$URGENT` and `$OLD` variables both define the number of seconds. Data that has arrived within `$URGENT` seconds is al-

---

*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](http://perl.org)). Randal can be reached at [merlyn@stonehenge.com](mailto:merlyn@stonehenge.com).*

ways pure white on the output. As it passes that threshold, its color changes from light green through dark green, maximizing the darkest pure green at `$URGENT` plus `$OLD` seconds. In this case, I'm letting it fade to dark green in about a minute. Again, the values will depend on the data being watched. I get about 20,000 hits a day on the web server, so these particular settings showed lots of variations between everything being all white and all green. At a glance, I could quickly gauge recent traffic.

Line 11 defines the rolling data buffer, holding the most recent `$TAILING` events. Items are pushed onto the end and shifted off the front as needed. Each element of the array is a two-element arrayref, holding the UNIX epoch-based integer time value when the data was added, and the line itself (without a newline, apparently).

Line 12 pulls in the HTML-generating shortcuts from the core `CGI` module. Although we actually aren't in any kind of CGI mode here, I find the HTML shortcuts to be easier than typing a lot of angle brackets myself.

Line 13 brings in the `POE` pieces. Arguments passed to `use POE` act like I had said:

```
use POE;  
use POE::Component::Server::HTTP;  
use POE::Wheel::FollowTail;
```

That's a convenient convention with `POE`'s nonstandard import list.

Lines 15 through 54 create the web-server side of the POE process, using the `POE::Component::Server::HTTP` module (often referenced as `PoCo::HTTPD` in brief). I lifted this example almost directly from the POE Cookbook, available in the Wiki at <http://poe.perl.org/>.

Line 17 defines the port number of the web server. By default, the web server listens on all IP addresses at this port number. I picked 42042 as an easy number to remember. You'd definitely want to check to ensure that such a port was not already in use by another application.

Lines 18 to 53 define the only `ContentHandler` for this server. Any request to any URL below slash (therefore, any request) gets directed to this handler. The two parameters to the subroutine are the `request` and `response` objects. The handler is responsible for examining the `request` object, then updating the `response` object as needed, and then returning the appropriate HTTP status code. Because we are returning the same contents regardless of the request, no checking of the `request` is performed.

Line 22 sets the response to the HTTP *OK* value of 200. Line 23 sets the content type of the *response* to *text/html*, indicating a typical HTML *response*.

Lines 24 to 50 generate the content for the response, using the *CGLpm*'s HTML shortcuts. The *join* in line 25 turns the many pieces into one string. I wasn't sure if I needed to do that by looking at the manpage for *HTTP::Message*, but I knew it couldn't hurt.

Lines 26 to 30 generate the HTML head and title section, including the very important "meta-refresh." This refresh will cause most modern browsers to reload the data repeatedly every five seconds. The *#bottom* anchor will ensure that most browsers also scroll to the bottom anchor (defined in line 49). The background of the window is set to black, which seemed to work out best during testing.

Lines 31 to 48 generate the bulk of the output: a single table. Each row of the table is one of the *\$TAILING* lines, and consists of two table cells. The left cell is the timestamp (in the *localtime* of the web server) in a nice cyan lettering, while the right cell is the log file line.

The log line is in Courier font, selected explicitly. During testing, I tried putting it inside a *pre* element, but that kept the line from wrapping, making it difficult to read very wide log lines.

The lines of the table come from a *map*, which processes the elements of *@data* (line 48), invoking the block defined in lines 32 to 47 for each element. Line 32 extracts the timestamp and the log line itself by dereferencing the *\$\_* (the current element of *@data*) as an arrayref.

Lines 33 to 42 figure out what color a line that was seen at *\$stamp* time should be. First, the age of the line is computed in line 33. If the age is below *\$URGENT* seconds (line 35), then the color is forced to white. Otherwise, the age is scaled by subtracting *\$URGENT* (line 38) and maximized to be no more than *\$OLD* seconds (line 39).

Line 40 computes a *\$c* value that will be 0 for the oldest lines and 255 for the very newest. Line 41 uses this value twice to create a color that scales from *#ffffff* (white) down to *#00ff00* (pure green) linearly. Thus, lines that arrived at identical times have identical values, and lines that arrived at differing times generally can be rapidly visually distinguished.

Once we have a color, we can construct the table row, starting in line 43. Lines 43 through 45 create the timestamp. The *valign* parameter is set to *top* so that the timestamp always appears at the start of the log line, even if the log line wraps. The *font* element is used to assign the color *cyan* to the timestamp. Line 45 uses a *sprintf* to create a zero-padded time value. The *localtime* operator is called in a list context for a list slice; and the hours, minutes, and seconds values are selected in the proper order for the *sprintf*. Nice trick.

Lines 46 and 47 create the table cell for the log line. The line is HTML-entitized using *escapeHTML*, ensuring that less-thans don't ruin my day. I can just imagine someone deliberately visiting a URL of *<BLINK>* just to mess up the rest of my display. But on a more serious note, we don't want to open ourselves up to a cross-site-scripting attack either. Line 46 uses another *font* element to define the proper color and font face for the text.

Line 49 adds an anchor to the bottom of the display, which will be made visible by most browsers because of the address of the meta-refresh. This ensures that we are looking at the most recent data (at the bottom), even if the data exceeds the window size.

Finally, the closing body and HTML tags are created with line 50.

Once the content of the response has been established, we return from the subroutine with an *OK* code again, thus passing back the proper HTTP status to the browser.

That wraps up the web-server side of the process. As a browser connects to port 42042, this code gets activated, the current *@data* gets filtered and formatted, and the response is returned to the browser. The meta-refresh causes the browser to refetch the same URL five seconds later, with the updated response sent to the browser again.

But how does *@data* change? Well, within the same process, we also have a *POE::Session* object that is running a *POE wheel*. This session is created in lines 57 to 75. A *POE wheel* defines a correlated set of states and events for a higher-level function. This particular wheel knows how to "tail" a file, generating events when new data is available.

Line 59 declares that our session will have inline states, one of the most common ways to create a session. We could also have object states or package states, but this seems to work the easiest for this example.

---

*I can just imagine someone  
deliberately visiting a URL of  
<BLINK> just to mess up the rest  
of my display*

---

When the session is first started, the *\_start* state is called. We'll define this start handler as an inline *coderef* starting in line 61. The start handler creates a *POE::Wheel::FollowTail* wheel, and stores it into the session's heap area, ensuring that it stays alive as long as the session is alive. Because we never destroy this session, this essentially means forever. The heap key of *wheel* is arbitrary: We could have used "mongoose" with no consequences.

The wheel takes a few parameters, defined in lines 65 through 67. The *Filename* parameter is the file to be tailed. Here, we get the name directly from the configuration parameter *\$FILENAME* defined at the top of the program. Because we've provided an explicit filename, this wheel is smart enough to close and reopen the file when the file finally changes, such as when we roll the web logs (which I do nightly). That's a lot better than my *tail -f* sessions, which seem to freeze around 2AM, until I remember that the logs have been rolled!

The *InputEvent* defines a callback event that will be triggered whenever new data is available from the file. It appears to be called once per line, although that wasn't clear in the documentation.

*SeekBack* defines how many bytes of the file are initially processed, by seeking back from the end of the file. Note that this is a byte count and not a line-number boundary, so we might end up in the middle of the line for the first entry.

As new lines arrive (or initially while processing the seekback tail of the file), the event handler *got\_line* is called, and defined starting in line 70. Line 71 simply takes the line from *\$\_[ARG0]*, prepends the current timestamp, and pushes the resulting arrayref onto the end of *@data*. Line 72 ensures that the array never exceeds *\$TAILING* elements by trimming elements from the beginning of the array when necessary.

Finally, since *POE* is an event loop manager, we need to put the *POE* kernel in charge of the program. This is done in line 77. In this case, the loop never exits because we always have a web-server session and a followtail session, so the program never returns from this call.

So there you have it. A proof-of-concept program to inspire you to write more interesting and cool things with *POE*, especially now that you've seen how easy it is to create a tiny web server. Until next time, enjoy!

TPJ

## Listing 1

```
1  #!/usr/bin/perl
2  use strict;
3
4  ### configuration items ###
5  my $FILENAME = "/web/stonehenge-proxy/var/log/access_log";
6  my $TAILING = 16;
7  my $URGENT = 5;           # this many seconds stays white
8  my $OLD = 55;           # this many seconds after urgent is dark
9  ### end configuration items ###
10
11 my @data;
12 use CGI qw(:html);
13 use POE qw(Component::Server::HTTP Wheel::FollowTail);
14
15 POE::Component::Server::HTTP->new
16 (
17   Port => 42042,
18   ContentHandler =>
19     {"/" => sub {
20       my ($request, $response) = @_;
21
22       $response->code(RC_OK);
23       $response->content_type("text/html");
24       $response->content
25         (join "",
26          start_html(
27            -title => "web tail on $FILENAME",
28            -bgcolor => 'black',
29            -head => ["<meta http-equiv=refresh content=5;#bottom>"],
30          ),
31          table (map {
32            my ($stamp, $message) = @$_;
33            my $age = time - $stamp;
34            my $color;
35            if ($age < $URGENT) {
36              $color = 'white';
37            } else {
38              $age -= $URGENT;
39              $age = $OLD if $age > $OLD;
40              my $c = 255 * (1 - $age / $OLD);
41              $color = sprintf '#%02x*%02x*%02x', $c, 255, $c;
42            }
43
```

```
43         Tr(td({valign => 'top'},
44             font({color => 'cyan'},
45                 sprintf "%02d:%02d:%02d", (localtime $stamp)[2,1,0])),
46             td(font({color => $color, face => 'courier'},
47                 escapeHTML($message))));
48             ) @data,
49         a({name => "bottom"}),
50         end_html,
51       );
52       return RC_OK;
53     }},
54   );
55
56
57 POE::Session->create
58 (
59   inline_states =>
60     {
61       _start => sub {
62         $_[HEAP]->{wheel} =
63           POE::Wheel::FollowTail->new
64             (
65               Filename => $FILENAME,
66               InputEvent => 'got_line',
67               SeekBack => 8192,
68             );
69       },
70       got_line => sub {
71         push @data, [time, $_[ARGO]];
72         splice @data, 0, -$TAILING if @data > $TAILING;
73       },
74     },
75   );
76
77 $poe_kernel->run();
```

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/>



# Test-Driven Development By Example

*Tim Kientzle*

A few weeks ago, I started developing a new library. The first step, of course, was to write a short test program that used the library. This gave me a chance to think through the library interface carefully and see if it would really be useful in the situations I envisioned. The next step was to stub out the library functions so that the test program would actually compile and run. Only then was I ready to start implementing the library, testing each small addition as I went.

This informal approach grew out of my personal experience, heavily influenced by the writings of Fred Brooks, Donald Knuth, and Leo Brodie. However, Test-Driven Development (TDD), as it is now known, has been recognized as a formal software development technique and carefully refined and documented as a part of the Extreme Programming (XP) methodology.

Kent Beck's book *Test-Driven Development By Example* takes a careful look at the ideas and rationale behind TDD in its own right. As such, it is one of the first thorough introductions to TDD that can be used by developers working alone or working on teams that have not adopted XP.

The book starts with two carefully presented examples. Through these examples, Beck introduces TDD as a minute-by-minute approach to software development, a way to organize your work so that you are continually realizing small, achievable goals. The approach can be succinctly summarized as a three-step cycle:

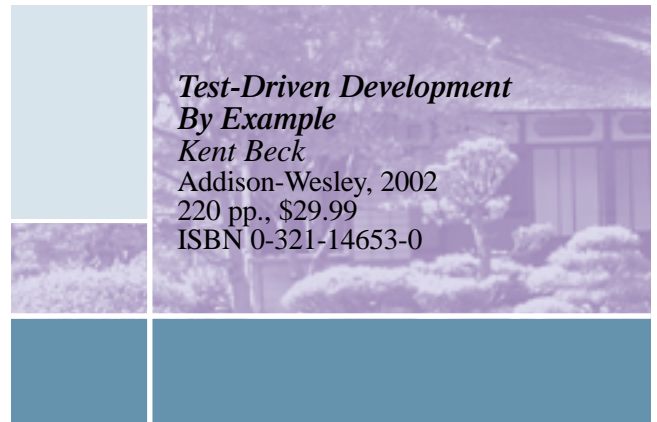
1. Red: Write the minimal test code to break your implementation.
2. Green: Write the minimal implementation to pass your tests.
3. Refactor: Clean up any code duplication introduced by the above.

You should take the word "minimal" very seriously. Here's a brief summary of one development sequence from the book:

- Write a test that invokes a multiplication function with arguments 5 and 2, and checks that the result is 10.

---

*Tim is a freelance software developer and consultant. He can be reached at [kientzle@acm.org](mailto:kientzle@acm.org).*



- Test fails (program doesn't compile) because the function is not implemented.
- Implement multiplication function to return constant 10.
- Test succeeds.

The lesson here is subtle. On the one hand, your test code can be thought of as a list of flaws in your implementation. Conversely, your implementation provides a way to explore flaws in your test code. If an obviously incorrect implementation satisfies your tests, then your test code itself must be incomplete. In this case, the next step is to beef up your test code until the only reasonable implementation that will pass the tests is the obviously correct multiplication function.

Of course, tiny development steps don't always create clean code. That's why TDD requires you to continually examine your code for ways to improve the structure and eliminate duplication. This is another reason to minimize your implementation at each step—extraneous code makes it more difficult to clarify and improve the structure.

At its best, this cycle generates a positive feedback loop: Because each small change to the implementation is preceded by an addition to your test code, you can have nearly perfect test coverage.



# Sign Up For BYTE.com!

**DON'T GET LEFT BEHIND!**  
**Register for BYTE.com today**  
**and have access to...**

- Jerry Pournelle's "Chaos Manor"
- Moshe Bar's "Serving with Linux"
- Martin Heller's "Mr. Computer Language Person"
- David Em's "Media Lab"
- and much more!...

**BYTE.com will keep you up-to-date on emerging trends and technologies with even more rich technical articles and opinions than ever before! Expert opinions, in-depth analysis, trusted information...find all this and more on BYTE.com!**



**As a special thank you for signing up, receive the BYTE CD-ROM and a year of access for the low price of \$26.95!**

**Registering is easy...go to [www.byte.com](http://www.byte.com) and sign up today! Don't delay!**



**BYTE**

Because you have nearly perfect test coverage, you can have a high confidence that code reorganizations won't break any existing functionality. Because you are constantly striving to keep your code clean and well organized, you are able to make small changes that incrementally add new capabilities.

The example above is taken from the first section of Beck's book, which develops a currency-manipulating class. The slow pace here allows the author to make his message clear—"small steps, test first, clean as you go"—and gives readers some space to think about how this approach might apply to their own work.

---

*You'll need a passing familiarity with Java, Python, and SmallTalk to follow all of the examples*

---

By the end of this section, I saw many ways to improve my own informal use of TDD.

The middle section works through a more ambitious example: a full testing framework for Python, itself developed using TDD. I found this portion of the book considerably more confusing than it should have been. This is partly because I don't know Python very well and the author's promised "commentary on Python, for those...who haven't seen it before" never really materialized. More seriously, the author starts writing code and verifying small details without ever discussing the overall architecture. He also refers to a number of concepts that are not explained until much later in the book. Although there is some useful material here, I suggest skipping this entire section on a first reading.

Fortunately, the final part of the book is worth the wait. Here, the author uses a series of patterns to more deeply explore the mechanisms and philosophy of TDD. I found the refactoring patterns especially interesting, partly because they are really development patterns and not design patterns. Rather than describing a common code structure, they explore common approaches for developing code, such as breaking a single function in two or combining redundant functions into one.

Although this is a good book overall, I did find a few annoying problems: You'll need a passing familiarity with Java, Python, and SmallTalk to follow all of the examples. The Python code was not always correctly indented. The "traffic light" metaphor is heavily used and never explained. Similarly, the xUnit test architecture is never really clearly explained.

Extreme Programming is not a monolithic creation. Rather, it is a collection of techniques, not all of which are appropriate for every development team. By presenting TDD independently of XP, Beck is opening these ideas to a much wider audience. Ironically, this book may be responsible for many more people easing their way into XP. Small steps often work best.

TPJ

---

# Source Code Appendix

---

## Peter Sergeant "Mouse Tracking with JavaScript and Perl"

### Listing 1

```
1 #####!/usr/bin/perl
2
3 use strict;
4 use Imager;
5 use Data::Dumper;
6 use Imager::Fill;
7
8 my %config = (
9
10 'Box Dimensions' => 10,
11 'Max Box Score'  => 20,
12 'Site Image'    => 'screenshot.png',
13 'Output Image'  => 'outmouse.png',
14 'Mouse Trails'  => 'cookies.log',
15 'Screen Size'   => '768z1024',
16 'Count Repeats' => 3,
17 'Max Opacity'   => 200,
18
19 );
20
21 my %grid_score_hash;
22 my $high_score;
23
24 open( LOGFILE, "< $config{'Mouse Trails'}" ) || die $!;
25
26 while(<LOGFILE>) {
27
28     chomp;
29
30
31
32     # Check if it's the right screen-size
33     # A sample data line looks like: 768z1024|aa234b82aaaaaaaa229b94a223b145
34     next unless substr( $_, 0, 9, '' ) eq $config{'Screen Size'} . '|';
35
36     # Create some useful holding variables
37     my ($old_x, $old_y) = (0, 0);
38     my %user_hash;
39
40     # Extract coordinate readings from our data line
41     for (split(/a/, $_)) {
42
43         # Extract the coordinates themselves from our coordinate block
44         my ($x_coord, $y_coord) = split(/b/, $_);
45
46         # Normalize the coords
47         $x_coord = int( $x_coord / $config{'Box Dimensions'} );
48         $y_coord = int( $y_coord / $config{'Box Dimensions'} );
49
50         # If the coordinate is blank, set it to the last-read one
51         $x_coord = $old_x unless $x_coord;
52         $y_coord = $old_y unless $y_coord;
53
54         # Cache the values
55         $user_hash{"$x_coord|$y_coord"}++;
56         unless $user_hash{"$x_coord|$y_coord"} >= $config{'Max Box Score'};
57     }
58 }
59
60 for (keys %user_hash) {
61
62     $grid_score_hash{$_} += $user_hash{$_} unless $_ eq '0|0';
63
64     # Calculate high-score
65     if ($grid_score_hash{$_} > $high_score) {
66         $high_score = $grid_score_hash{$_}
67     }
68 }
69 }
70
71 }
72
73 # Work out the opacity multiplier
74 my $opacity_multiplier = ( $config{'Max Opacity'} / $high_score );
75
76 # Create new Imager object
77 my $start_image = Imager->new();
78
79 # Open our site image
80 $start_image->open( file => $config{'Site Image'} )
```

```

        or die $start_image->errstr();
81 my $image = $start_image->convert( preset => 'addalpha' );
82
83 # We cache Imager colours here to save duplication
84 my %fill_cache;
85
86 # Go through the hash
87 for (keys %grid_score_hash) {
88
89     my ($xcoord, $ycoord) = split(/\|/);
90     $xcoord *= $config{'Box Dimensions'};
91     $ycoord *= $config{'Box Dimensions'};
92
93     # Work out the opacity
94     my $opacity = int( $grid_score_hash{$_} * $opacity_multiplier );
95
96     # Create a fill in Imager
97     $fill_cache{$opacity} = Imager::Fill->new(
98         solid => Imager::Color->new( 0, 0, 255, $opacity ),
99         combine => 'multiply'
100    ) unless $fill_cache{$opacity};
101
102    # Add a box to the imager in the appropriate place
103    $image->box(
104        fill => $fill_cache{$opacity},
105        xmin => $xcoord,
106        ymin => $ycoord,
107        xmax => ($xcoord + ( $config{'Box Dimensions'} - 1 ) ),
108        ymax => ($ycoord + ( $config{'Box Dimensions'} - 1 ) ),
109        color => Imager::Color->new( 0, 0, 255, $opacity ),
110        #filled=>1
111    );
112
113 }
114
115 # Print our image
116 $image->write( file => $config{'Output Image'} ) or die $image->errstr;

```

## Randal Schwartz “Tailing Web Logs”

### Listing 1

```

1  #!/usr/bin/perl
2  use strict;
3
4  ### configuration items ###
5  my $FILENAME = "/web/stonehenge-proxy/var/log/access_log";
6  my $TAILING = 16;
7  my $URGENT = 5;           # this many seconds stays white
8  my $OLD = 55;           # this many seconds after urgent is dark
9  ### end configuration items ###
10
11 my @data;
12 use CGI qw(:html);
13 use POE qw(Component::Server::HTTP Wheel::FollowTail);
14
15 POE::Component::Server::HTTP->new
16 (
17     Port => 42042,
18     ContentHandler =>
19     {"/" => sub {
20         my ($request, $response) = @_;
21
22         $response->code(RC_OK);
23         $response->content_type("text/html");
24         $response->content
25             (join "",
26              start_html(
27                  -title => "web tail on $FILENAME",
28                  -bgcolor => 'black',
29                  -head => ["<meta http-equiv=refresh content='5;#bottom'>"],
30              ),
31              table (map {
32                  my ($stamp, $message) = @$_;
33                  my $age = time - $stamp;
34                  my $color;
35                  if ($age < $URGENT) {
36                      $color = 'white';
37                  } else {
38                      $age -= $URGENT;
39                      $age = $OLD if $age > $OLD;
40                      my $c = 255 * (1 - $age / $OLD);
41                      $color = sprintf "%02x%02x%02x", $c, 255, $c;
42                  }
43                  Tr(td({valign => 'top'},
44                      font({color => 'cyan'},
45                          sprintf "%02d:%02d:%02d", (localtime $stamp)[2,1,0]),
46                      td(font({color => $color, face => 'courier'},

```

```

47         escapeHTML($message)));
48     } @data),
49     a({name => "bottom"}),
50     end_html,
51     );
52     return RC_OK;
53     }},
54     );
55
56
57 POE::Session->create
58 (
59     inline_states =>
60     (
61         _start => sub {
62             $_[HEAP]->{wheel} =
63             POE::Wheel::FollowTail->new
64             (
65                 Filename => $FILENAME,
66                 InputEvent => 'got_line',
67                 SeekBack => 8192,
68             );
69         },
70         got_line => sub {
71             push @data, [time, $_[ARG0]];
72             splice @data, 0, -$TAILING if @data > $TAILING;
73         },
74     ),
75 );
76
77 $poe_kernel->run();

```