

The Perl Journal

Perl Benchmarking

Glenn Wood • 3

Perl and Human-Computer Interaction

Ala Qumsieh • 7

HTML Calendars

brian d foy • 10

Musical Archaeology with Perl

Simon Cozens • 12

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Kevin Carlson:

***Mac OS X Panther Hacks* • 16**

Source Code Appendix • 18

LETTER FROM THE EDITOR

Know What I Mean?

I'm surrounded by folks who take language seriously. I work all day in the company of editors, my wife is a high-school English teacher, and a good friend of mine is a linguist. It's a good thing I'm as interested in language as they are, or I would probably fall asleep during dinner conversations.

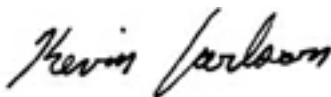
Language is fascinating because so much about the way we parse meaning out of words is still mysterious. Linguists haven't even begun to really understand how our brains do this amazing thing. Meaning is an elusive thing, both in spoken language and in high-level computer code. In a computer language, of course, we have the luxury of specificity, so exact, unambiguous meaning can be parsed, given enough analysis. If that weren't true, compilers wouldn't function.

In the human brain, it seems that meaning is transferred by a much less precise process, sometimes with hilarious results. In her English class, my wife recently used the phrase "putting on airs." In a student's essay, it came back to her—filtered through the student's imprecise parser—as "putting on ears." I couldn't help picturing a nouveau riche Mr. Potato Head doll trying to fit in amongst old money.

Even in computer code, however, there's a problem with conveying meaning. The problem isn't making the compiler understand what we mean (once we have finished debugging, that is)—the problem is making other programmers understand our meaning. Sure, other programmers can do what the compiler does, and track down the exact syntax behind every piece of code in every subroutine in order to understand what it "means" to use that subroutine, but what we strive for is a much more abstract level of comprehension in those who read or use our code. We want to make it make sense on an intuitive level. So we try to give our variables sensible names, and we try to make subroutine parameters meaningful and intuitive. We try to design configuration files so that users can easily understand the options without repeated trips to the documentation.

But it's not that easy. What makes perfect sense to you might be gibberish to someone who hasn't just spent the last two months of his or her life under the hood of your module. I recently rewrote a bit of my old code, focusing on making the code (and specifically its configuration file) much more intuitive. While the new code is much more flexible, I failed utterly to make it any less difficult for an outsider to understand. I may have even made it worse. It was just a big exercise in myopia.

So what do we do? We certainly can't wait for the linguists to figure out how our brains work. The short-term answer, of course, is to rely on the conventions of understanding that crop up in communities of like-minded people. Perl programmers have certain expectations about the way Perl code should be structured. So get others to read your code and tell you what defied their expectations. That may not constitute a full mastery of the linguistics involved, but at least you'll be using the same shorthand as everyone else, and that will go a long way to helping you get your point across.



Kevin Carlson
Executive Editor
kcarlson@tpj.com

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

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



The Perl Journal

EXECUTIVE EDITOR

Kevin Carlson

MANAGING EDITOR

Della Wyser

ART DIRECTOR

Margaret A. Anderson

NEWS EDITOR

Shannon Cochran

EDITORIAL DIRECTOR

Jonathan Erickson

COLUMNISTS

Simon Cozens, Brian d'Joy, Moshe Bar, Andy Lester

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

WEBMASTERS

Sean Coady, Joe Lucca

MARKETING / ADVERTISING

PUBLISHER

Michael Goodman

MARKETING DIRECTOR

Jessica Hamilton

GRAPHIC DESIGNER

Carey Perez

THE PERL JOURNAL

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

CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND MARKETING Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT/GROUP DIRECTOR INTERNET BUSINESS Mike Azara

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

VICE PRESIDENT, COMMUNICATIONS Alexandra Raine

PRESIDENT, CHANNEL GROUP Robert Faletra

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED TECHNOLOGIES Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS Paul Miller

VICE PRESIDENT, GROUP PUBLISHER NETWORK COMPUTING

ENTERPRISE ARCHITECTURE GROUP Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE DEVELOPMENT MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING SOLUTIONS Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES Marie Myers

Perl News

Parrot 0.1.1 Released

It's been about six months since Parrot 0.1, and as Dan Sugalski put it in his blog (<http://www.sidhe.org/~dan/blog/>): "A lot's happened since then. We don't think about it much, since we're all used to just sync-ing up to the CVS server (which anyone can do—there's full anon CVS access and rsync access to the repository) but a lot of folks like having a stable release. So...0.1.1."

Dubbed Poicephalus (after a popular breed of pet parrots), Parrot 0.1.1 is available from <http://www.cpan.org/authors/id/L/LT/LTOETSCH/parrot-0.1.1.tar.gz>. Leo Toetsch's release notes cite better OS support; improved PIR syntax for method calls and = assignment; reworked dynamic loading, including a "make install" target; multimethod dispatch for binary *vtable* methods; an improved and cleaned-up library; and "tons of fixes, improvements, new tests, and documentation updates." Leo sums up: "A lot is unfinished and keeps changing. Nevertheless Parrot is stable and usable at the surface, while internals are moving."

In Perl 5 news, Nicholas Clark has lengthened the release schedule for Perl 5 to four months from three, meaning that code changes for Perl 5.8.6 must be committed to bleed by October 31, and we should expect to see the first release candidate in early November.

Parrots and Pythons and Pie, Oh My

The new Parrot 0.1.1 runs slightly more than half of the pie-thon test suite, but Dan is looking for volunteers to finish the work. The pie-thon, of course, came into being when Dan bet Guido van Rossum that Python bytecode could be made to run faster on Parrot than it does in CPython. The challenge ended on a bittersweet note: Dan took a couple of pies in the face, but a \$520 donation was made to the Perl Foundation in his honor.

However, although the Python-on-Parrot project wasn't finished in time to spare Dan the taste of cream pie, the work so far bears very promising results. The Parrot team got four of the seven benchmarks running—and of those, three came out with faster times than CPython. "Both Leo and my translators were reasonably near completion, and need to be pushed that final bit of the way," Dan wrote on the perl6-internals list. "Neither of us have the time, so...Anyone want to take a shot? Leo's builds faster code but mine's a bit clearer, and both could be mined for ideas for a third, completely different one. (Or you could go the IronPython route and reimplement the parser, which works too)."

One List Summarizer Steps Up As Another Steps Down

Scott Lanning has succeeded Rafael Garcia-Suarez as the new Perl 5 list summarizer. As always, the summaries can be read on use.perl.org, or by sending e-mail to perl5-summary-subscribe@perl.org.

Unfortunately, Piers Cawley has been forced to go on hiatus after two and a half years as the Perl 6 Summarizer; a teaching

course is currently demanding most of his time. "I may not have stopped writing the summaries for good either; I just haven't got computrons to spare for writing them at the moment," he wrote to the list. "But if any of you are thinking 'I could do that!' then don't let me stop you—there's an awful lot goes on on the lists, and there's a lot of interested people who don't have the time to keep up with them. A regular summary helps the interested but busy people get a grasp of how the Perl 6 project is getting on, and that can only be a good thing." As there are now three separate Perl 6 lists—perl6-internals, perl6-language, and perl6-compiler—the job may well call for more than one volunteer.

PerlEx: The End

ActiveState has given notice to its customers that PerlEx engineering support will end on September 30th of next year. "We have decided to better allocate our Perl development and support resources," the company announced. "Although there will be no further maintenance updates or bug fixes for PerlEx, we will continue to offer existing PerlEx customers standard support (installation and configuration) through March 31, 2005. There are no plans to deactivate the PerlEx discussion list or remove product documentation on ASPN."

PerlEx was designed to improve Perl performance on Windows-based web servers. It works similarly to `mod_perl` by precompiling scripts and enabling persistent database connections. It also supports ASP-style embedding of Perl code in HTML files; SOAP and XML-RPC integration; encryption of CGI scripts; and integration with Windows Performance Monitor tools. ActiveState's end-of-life notice is at <http://www.activestate.com/Products/PerlEx/>; PerlEx technical documentation remains at http://aspn.activestate.com/ASPN/docs/ASPNTOC-PERLEX____/.

Upcoming Events

The Open Source Developers Conference (OSDC) in Melbourne, Australia—previously known as YAPC::AU::2004—is now open for registrations. The conference, scheduled for December 1–3, now includes tracks on Python, PHP, and open source operating systems, as well as, of course, Perl. Damian Conway and Nathan Torkington will give keynote speeches. Attendance fees are \$265; see <http://www.osdc.com.au/registration/index.html> to register.

The 7th German Perl-Workshop, which will take place next February in Dresden, has issued a call for papers. The deadline for proposals is October 31. According to the conference web site (<http://www.perl-workshop.de/2005/docs/cfp.htm>), "Conference language is German, but you can give your presentation in English if German isn't your native language." The conference organizers are looking for five, 20, or 40 minute talks on "every subject that pertains to Perl or its periphery."

Perl Benchmarking

At my company, we recently faced a dilemma—we had a Perl process that ran along, consumed 2 GB of RAM, then died without a peep (not even a whimper). This process required about an hour of a quad i686 Linux box to do its magic. But in this instance, we got nothing; there was no result, no return code, no reports; not even a message in the error log.

What to do? At this point, the conversation between the programming team and our boss went something like this:

Boss: Ok, everybody, let's do a code review!

Us: We did that, weeks ago. Do you think we'll find anything new?

Boss: More tests?

Us: We've tested this, and so has QA. That's how we found this problem. It works with other datasets; it's just this one that's causing the problem.

Boss: Analysis, then—look at the input data.

Us: You mean all 50 GB of it, or only the 500 MB picked out by the three page SQL SELECT statement?

Boss: Profiling?

Us: Ok, let's! Here we go...looks like most of the time is being spent in function X, which is called in, umm, give us a minute...uh...only 23 different places. Well, that didn't help much.

Boss: Tracing, then. Stick in *print "Hello World"* in a bunch of places, and analyze the result. That will tell us where it quit, at least.

Us: I thought you said you wanted this fixed by Christmas.

The answer in a case like this is a special kind of code test called a "harness." Like profiling, it gives you intimate information about what functions are being invoked, how often, and for how long. It goes beyond profiling in that any kind of analysis can be performed at the moment of each function's invocation (and/or return). It's kind of a cross between profiling and tracing.

Most languages perform this type of analysis by subclassing, code templates, applying test interfaces to the classes under test, or by time-consuming (and expensive) execution using special test environments. This usually means recompiling the program, redeploying the application, perhaps deploying an expensive test tool, and then rerunning the test.

Perl is a special kind of creature, though (as if you didn't know). What many people do not realize (or fail to grasp the significance of) is that Perl is compiled each time it is run, and classes and

methods can be redefined after they have been initially defined, at runtime. This feature can be used to implement a powerful, easily used, generalized benchmark harness.

Benchmark::Harness (available on CPAN) is a module that does most of this work for you. You give it a list of modules/subroutines you want it to investigate (wildcards are accepted), and it interrupts the process at the entry and/or exit of these subroutines to perform whatever operations you want to perform at that time. It then hands control back to the subroutine or caller as if nothing had happened. The CPAN distribution already contains a few sets of such operations you can apply simply by naming one in your parameters to the harness. You can create new sets of operations by using one of these samples as a template.

Basic Example

The graph in Figure 1 was created by a laborious process of sitting and watching the process run, writing down *top* statistics, entering those in a table, and graphing it. Let's look at an easier way to do that with *Benchmark::Harness*.

At the beginning of the process, we insert a single line to activate the harness:

```
new Benchmark::Harness('MemoryTrace', 1, 'Fubar::*');
```

MemoryTrace names a module that will handle the interrupts. The digit "1" instructs the disposition of the report. The final parameter, which may be repeated as often as necessary, names the subroutines of the modules you want to harness. That's all there is to it!

Benchmark::Harness and *MemoryTrace* will create an XML file containing statistics about the runtime environment at the entry and exit of each subroutine in the *Fubar* module (*Fubar::**). The XML file will contain a simple list of elements, one for each event that is harnessed, containing this data. For instance:

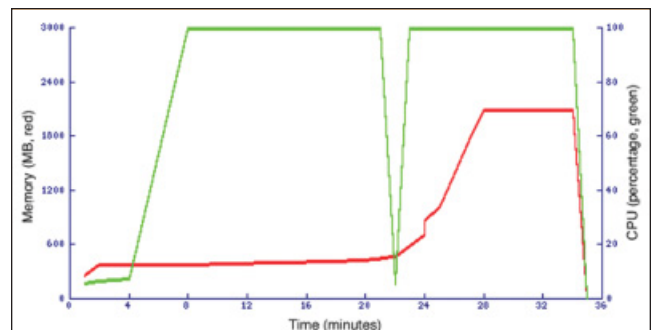


Figure 1: Hand-gathered process statistics.

Glenn works on HazelSt.com, a web site and support service for budding writers, and can be contacted at CTO@HazelSt.com.

```
<T t="60" c="5.0" n="Entry(Fubar::processItem)" m="245"/>
```

This is very compressed. This report may contain many thousands of events, so the names of elements and attributes in the XML are chosen to be short:

- *T*, name of a “trace” element in the XML.
- *t*, the time at which it was captured.
- *c*, the CPU utilization.
- *n*, the method that was harnessed.
- *m*, memory utilization at that point.

This element is created by *MemoryTrace*, so if you were to install your own set of harness operations instead of *MemoryTrace*, you could print any kind of information you like. For our purposes, *c* and *m* are the data we want.

Since it’s in XML, it is incredibly easy to use this data. We ran it through a Perl script to convert it to a graph using *GD::Graph::lines*. Don’t be intimidated by *GD*; it is very easy to use. Most of the folderol is for setting up special graphical characteristics, and that is standard stuff for any graph:

```
use GD::Graph::lines;

# Parse the T element from the report
my $xml = new XML::DOM('Fubar.harness.xml');
my @T = $xml->selectNodeList('/MemoryUsage/T');

# Translate the data to GD::Graph format
my $data = [
    [ map { $_->getAttribute('t') } @T ],
    [ map { $_->getAttribute('m') } @T ],
    [ map { $_->getAttribute('c') } @T ]
];

# Plot the graph
my $graph = new GD::Graph::lines(500,250);
$graph->set(
    # numerous GD::Graph::lines parameters go here . . .
);
$graph->plot($data);

# Write the graph to a file
my $ext = $graph->export_format;
open(IMG, ">Fubar.graph.$ext");
binmode IMG;
print IMG $graph->gd->$ext();
close IMG;
```

The result is shown in Figure 2.

Pretty Pictures

Ok, fine, but that’s only what we knew when we gathered this data by hand earlier. What we want to know is what subroutine is eating all the memory, and where did it die? So we’ll analyze the report a little more.

XSL is extremely valuable in analyzing XML documents and it shouldn’t be gratuitously overlooked, but it doesn’t create pretty pictures. We like pretty pictures, so we decided to extend the graphical representation we began previously.

Through a process of *legerdemain* (which we will not go into here), we transformed the data into arrays marking the entry and exit point of each subroutine, and then graphed those as lines alongside each other on the memory/CPU graph. Some minor trickery in *GD::Graph::lines* was required, but it is simple:

```
# Draw the Memory and CPU data
$graph->plot($data);

# Set parameters to draw the per-function lines
$graph->set(
    # a new set of GD::Graph::lines
    # parameters goes here
);

# Print a graphic line for each function
# @functionData was generated via legerdemain
for ( @functionData ) {
    my $data = [
        [ map { $_->getAttribute('t') } @T ],
        [ @$_ ],
    ];
    $graph->plot($data);
}
```

By the way, a SAX parser can parse XML even if the document is incomplete (as it would be in this case of catastrophic failure). This is an important consideration in selecting which XML parser you use to analyze the report.

Figure 3 is a graph of subroutine activity. There is a legend (not shown here) that identifies what subroutine each horizontal line represents. We see the last subroutine that is called that is still running at the end (top-right gray line), but we’ve also identified which subroutines participate in the ramp up.

Back to the Harness

We now have a better view of what is going on in our errant process, but we’d like to know more. It’s easy, with a harness, to find out more about what’s happening.

We’ll introduce some terminology here, and then extend the example. The moment at which each method is invoked is called an “event.” The operation that is performed by the harness at that time is, therefore, an “event handler.” To those familiar with GUI applications, or with real-time applications, these are familiar terms, and they mean the same thing in this context, with one exception. In a GUI or real-time API, the API determines what events

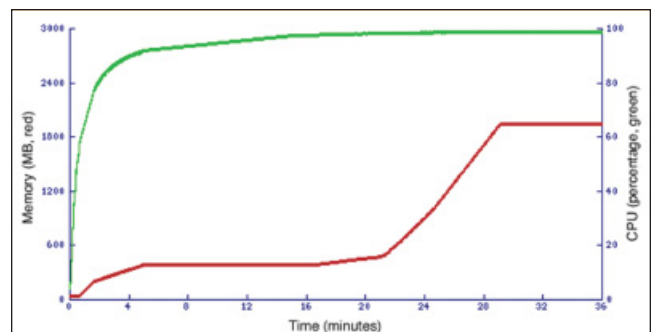


Figure 2: Graph of Benchmark::Harness data.

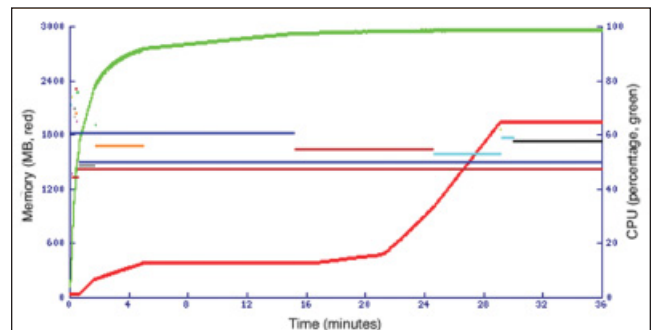


Figure 3: Subroutine activity.

there are. In the harness model, we determine what events are fired. We declare new types of events simply by naming methods of our program in the new *Benchmark::Harness* statement.

Once the event is captured in the event handler, we can do anything we want. It is Perl, after all.

When the process enters a subroutine, *Harness* calls the event handler named *OnSubEntry()*. When it exits the subroutine, *OnSubExit()* is called. These are defined in *MemoryTrace*, which in turn is named by the new *Benchmark::Harness* statement. We can subclass *MemoryTrace*, or we could simply write our own event handlers module. For simplicity, let's take the second route, and for laziness, use the original *MemoryTrace.pm* as our initial template.

When the harness calls *OnSubEntry()* or *OnSubExit()*, it hands over the parameters of the target subroutine, then that subroutine's returned result, respectively. Thus:

```
Package MyHarness; use base qw(Benchmark::Harness);

sub OnSubEntry { # (self, funcName, funcParms)
    my $self = shift; # MyHarness object instantiated
                    # by new Benchmark::Harness()
    my $funcName = shift; # Name of harnessed function
    print "$funcName has $#_ parameters\n"; # Our 'trace'
    return @_;
}

sub OnSubExit { # (self, funcName, funcReturn)
    my $self = shift;
    my $funcName = shift;

    if (wantarray) { # wantarray context is preserved
        print "$funcName returns an array - @_ items\n";
        return @_;
    } else {
        print "$funcName returns a scalar - $_[0]\n";
        return $_[0];
    }
}
```

You can print anything you like in your handler, but *Benchmark::Harness* provides a context to easily generate the kind of XML formatted report we illustrated above. You do this with one or more calls to *\$self->harnessReport()*.

harnessReport() takes any number of parameters, in any order, to specify contents of the *<T>* element that will be generated. The *<T>* element will actually be generated after you return from *OnSubEntry()* or *OnSubExit()*, so all parameters you supply via *harnessReport()* will be included at that time.

The parameters of *harnessReport()* are distinguished by their reference type:

- **scalar**—If the parameter is a scalar, it is the tag name of the element (default "T").
- **hash ref**—A hash of values that will become attributes of the *<T>* element. These are accumulated by subsequent calls to *harnessReport()*.
- **array ref**—References an array of child elements of *<T>*. These items themselves are array refs to ('tagName', {attributes}, [child elements], "text content").
- **scalar ref**—References a string that will become the text content of the *<T>*.

Analyze the Results

Back to our example. We had now determined that the sudden rise in RAM use occurred in a subroutine (we'll call it subroutine *Y()*), but more interestingly, it locked into high RAM use, without chang-

ing, in another subroutine (which we'll call subroutine *Z()*). It looked like an infinite loop in *Z()*.

A major benefit of writing a program as smaller functions is to ease analysis and debugging. Unfortunately, the coder (who shall remain nameless) who wrote this program didn't follow this dictum, and wrote *Z()* as a single 400-line subroutine. Where to start?

*[A harness] gives you
intimate information about what
functions are being invoked, how
often, and for how long*

Extend the Harness

We knew which subroutine was causing the problem, but there was a lot going on inside it. (This isn't really surprising. After all, if the solution were simple, we probably would have found it earlier.)

What we needed was a mechanism to analyze variable use within the subroutine, and also possibly a line-by-line trace. This could be done by inserting traces into *Z()*. This would have been a perfectly adequate approach, since we had localized the problem in one place. We wouldn't have had to put traces everywhere. However, this violates the purity of the harness model by requiring changes in the target code, so there should be a better way. Those of you familiar with Perl's *tie* operation and the *Tie::** modules can probably see the obvious application of these facilities here. We didn't go this direction in solving our problem, but I invite someone out there to perform this extension to *Benchmark::Harness* (if I don't get to it first!).

Browsing through the raw benchmark report, I noticed a strange phenomenon. We had just enhanced the harness to replace "ps -r" to discover memory usage (which is slow) with *Proc::ProcessTable* (which is fast). Just where memory use peaked and flattened out, the number recorded in the report changed from a large positive to a large negative number. What a coincidence, I thought. *Proc::ProcessTable* is having a problem rendering this number at exactly the same place our process stalls.

We began to wonder if there was something magical about the 2-GB threshold? It's a nice round number, after all. We found a few spots in the ramping up subroutine to reduce memory use by 5 or 10 percent, and ran it again.

This time, the process went further. Past 30 minutes, past 2 GB, right up to 2.6 GB in two hours. It was still running 36 hours later; still with no output.

That meant there was no magical 2-GB ceiling, and that the process was crawling to a near halt during its HTML rendering. We use a popular HTML rendering module (whose name is withheld to protect the innocent). It is being swamped by our input data. There isn't much hope in improving things by working to reduce memory use or to make our preprocessing more efficient. The same number of data points will go to rendering. The problem is the entire process itself, rather than any one component of it. There is just too much data for HTML rendering.

So in this case, *Benchmark::Harness* has helped us uncover a fundamental flaw in our design. We have not decided what to do about this problem yet, but thanks to *Benchmark::Harness*, we know that no matter how much code tweaking we do, it won't do any good. That realization has saved us a lot of time and trouble. Now that we know more about what we are up against, we can make a wiser choice. Maybe this report, in its current form, isn't really required at all. Perhaps we can just save the intermediate

*What we needed was a
mechanism to analyze
variable use within
the subroutine*

result and take slices of that to create subsidiary reports that may be as useful (or more so) than our original version. Regardless, the harnessing process has taken us out of the woods and the path ahead is clearer than it was before.

As the legendary Sherlock Holmes said, once you have eliminated all other possibilities, whatever is left (no matter how improbable) must be the case. What better way to eliminate all other possibilities than with a tool that searches for problems everywhere at once, as the harness model does.

There's More

The harness model addresses many of the traditional methods for solving problems in your code, including testing, analysis, profiling, and tracing. What is especially nice is that it addresses all of them at once, with an easy-to-implement method for activating the analyses.

But what about code review—learning how your target program is structured and how it actually works? The harness model has another powerful capability, one that is especially relevant to Perl.

One of the frustrations of working with Perl is that it can create new methods and properties for any class/module on the fly (which is both good and bad—few swords are single edged). This means that it can be difficult to know what methods and properties are actually incorporated in a program, even through intensive code review. Even here, the harness model can come to the rescue.

Since a harness can peek into every object at runtime, it could be enhanced to accumulate all the methods and properties it discovers in each object it encounters, as the program is run and as Perl instantiates new methods and properties along the way. Transform the results via XSL or Perl, and viola—instant class diagram!

I leave this as an exercise for the reader. At my company, this may be our next step—marrying *Benchmark::Harness* with *Devel::Diagram* to create beautiful class diagrams of Perl modules.

For more information about *Benchmark::Harness* and to participate in its continued development, visit Benchmark-Harness.org/.

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

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

Go to
<http://www.tpj.com/>
now!

Perl and Human-Computer Interaction

Let's face it—designing a product is hard. Regardless of what the product is, it is not a trivial problem to make people like it. It is even harder to get people to use it regularly. That problem becomes exponentially more complicated if people have to pay for it. That is the dilemma facing the CEO of every company, and obviously, it is not an easy problem. In this article, I will discuss different issues that I faced while designing various things. I will limit myself to Human-Computer Interaction (HCI) in general, and User-Interface (UI) design in particular. Finally, I will bring everything together from a Perl perspective.

I have been writing GUIs for around 10 years now, mostly as a hobby. My main platform for the past five or six years has been Perl/Tk, although I have also used Qt and Delphi. One thing that bothered me about Perl/Tk—and something that generates discussion on `comp.lang.perl.tk`—is the absence of a GUI builder. So I set out to write one, thinking that it's just another GUI that I have to build, albeit a nontrivial one. After all, I have created many GUIs. I just have to slap a few buttons together, create some dialogs, stuff all possible options in menus, and voilà! A GUI builder. In fact, the first version of the program literally took me only a handful of hours to build, from start to end.

But my assumptions turned out to be very wrong. The GUI was more a hindrance than help. So I started to think about what makes one interface better than another; hence, one product better than another. The big difference, I concluded, between this GUI and most of the others I have developed is that this will have a much wider audience, whereas most of the others had been written for myself. This complicates the problem exponentially because different people expect different things to happen, and although I will never satisfy everyone, I will have to cater to everyone's needs—which might be different from mine. So, I threw everything I did out the window, and started again from scratch.

Lessons Learned

I will now outline some of the lessons I learned and design decisions that I had to make, and revise several times, in my quest to create an intuitive GUI builder. The main goal behind every decision was to minimize the effort required by users to get their jobs done, and to let them enjoy themselves in the process. In his book *About Face 2.0* (John Wiley & Sons, 2003), Alan Cooper summarizes the three things that users look for in a product:

1. Users want to have at least a minimal amount of work done, so they please their superiors.
2. Users want to have fun using the product.
3. Users don't want to feel stupid while using the product.

It is interesting to note that these three rules loosely correspond to what Larry Wall considers the three great virtues of a programmer: Hubris, Laziness, and Impatience—getting the job done will give you a feeling of hubris, while having fun is surely just as good as laziness, and tackling a condescending interface will very quickly drive away your patience.

I will now give examples of some products and discuss their usability in light of the three rules above, and show how they mirror different aspects of Perl's design.

Expectation and Predictability

Different people have different experiences; hence, different expectations. It is essential that the user be able to predict the outcome of an action they might take. Otherwise, frustration starts to build. For example, I find it perfectly natural that double-clicking something invokes some action related to the object being double-clicked. I also assumed that this assumption was shared by everyone, so I made double-clicking on a widget open up a properties window for the widget where the user can change its properties. But I was wrong. A person reviewing a beta release of the GUI builder complained that right-clicking did not invoke a menu where he could choose the option to access the properties menu, and he couldn't find another way to do it. So I created a menu short cut to achieve the same behavior, and he was happy. Another example is Adobe's Acrobat Reader. In older versions of the software, pressing the up or down arrows would scroll a complete page up or down, respectively, while pressing the page up or page down keys would scroll the page by a small amount. This mix-up caused lots of confusion, and users resorted to using the mouse to drag the scrollbars for more predictable scrolling.

Luckily, Perl tries very hard to be predictable. For example, when testing numbers for equality using `eq`, Perl automatically converts them to strings. Alternatively, testing two strings that start with numbers using numerical comparison operators transforms them to numbers. Autovivification is another example where Perl does the right thing and creates any intermediate hash keys if necessary:

```
my %hash; $hash{key1}{key2}{key3} = 'value';
```

Ala works at NVidia Corp. as a physical ASIC designer. He can be reached at aqumsieh@cpan.org.

Experience

Past experience plays a big role in GUI design. You want users to feel at home when using your application. A good interface caters to those experiences, thus easing up the user's learning curve. Users will appreciate it if they can sit in front of a new interface and figure it out quickly. This lets them be more productive. For example, although the letter "V" appears nowhere in the word "Paste," binding a "Paste" function to anything other than Ctrl-V is wrong. Similarly, "Copy" has to be bound to Ctrl-C, "Undo"

Different people have different experiences; hence, different expectations

to Ctrl-Z, and "Cut" to Ctrl-X. The last thing you want is for people to have to learn a new technique for doing a common thing. Some other good examples are word processor programs that display a "letter" view of the document, leveraging people's familiarity with paper, and VoIP programs that display the numbers in a pad layout similar to those found on physical phones.

Perl's way of leveraging users' experiences is through its syntax. Perl borrowed heavily from a large number of very different languages, and many of Perl's constructs are very similar, in many cases identical, to those found in other languages. For example, its general syntax is very similar to C, which is the most widely used programming language. Anybody with past experience in C, C++, C#, or Java (among other languages) will be at ease with Perl's syntax, and will need to spend less time getting used to it and have more time to get useful things done.

Simplicity

People like simple things. As Einstein said, "Make things as simple as possible, but not simpler." A simple interface is an interface that people can fully understand quickly. It removes the element of uncertainty of whether the user missed an essential piece of the interface. Simple does not necessarily mean an empty interface, but rather an interface that gets out of your way and allows you to do your job. This usually translates to keeping the most widely used functions easy to access.

Perhaps the most famous example of a nonsimple interface is a VCR's clock. Something this simple should be trivial to set, yet it is anything but trivial. You have to go through a maze of menus, and sometimes press multiple remote buttons simultaneously, in order to change the clock settings. Worse still, most VCR clocks do not retain their setting and quickly diverge from the real time. I have heard countless jokes and even read some serious research papers on how to set VCR clocks. There is even a web site, <http://www.vcrclock.com/>, devoted to this frustrating interface. Yet, the fix is so simple. We just need to have two dials, one labeled "Hours" and the other labeled "Minutes," next to the LED display. Lifting a dial up increments the respective setting, while pushing it down decrements it. Simple and effective.

In the specific case of my GUI builder, I chose to give widgets sensible default configurations, and to hide less frequently used options. Also, I placed any option that the users will most likely modify—like a button's text and callback binding—in a toolbar, which makes it very easy for users to modify. If users need to modify more exotic configuration options, then those are still accessible, but the user has to dig a little deeper.

Some of the more famous products thrived as a direct result of their simple interfaces. Google shot to Internet fame because of its clean, empty, simple homepage. You are presented with a single text entry where you type your search query and hit "Search." That's what a search engine is supposed to do, and Google did it well. It presented the results in a simple, clean way with no intrusive text, allowing it to overtake Yahoo in the blink of an eye. Conversely, Yahoo is the epitome of clutter. I have to use my browser's "Find" function to find anything on their main web page. That's too much work. Another good example is Apple's iPod. It simply does its job through a simple, aesthetically pleasing interface, allowing it to take the portable music player market by storm, even though it didn't support some popular formats from Microsoft and Real.

Perl's simplicity is largely based in its forgiveness and its special variables. Larry Wall said that Perl makes easy things easy and hard things possible. This means that the most widely used constructs should be simple and easy to use. Less common constructs should still be possible. For example, Perl is great at string manipulation, so you'd expect that many people would use it for reading files. The usual idiom for reading from a file is:

```
while (<FH>)
```

which is equivalent to:

```
while (defined($_ = <FH>))
```

but is much simpler to write. Conversely, if you find yourself wanting to read from a binary file, then a little bit more work is needed. You have to tell Perl that this is a binary file, and use the `read` function to get the binary data in chunks:

```
binmode FH; while (read FH, $buff, $chunk_size)
```

Choice

This might seem contradictory to the "Simplicity" rule, but in fact it is not. A good interface can, and should, give its users the option to do anything they wish, within the limits of the product, taking care not to overburden with too much detail at once. After all, choice is a good thing. The hard part is how to give users the choice, while still keeping the interface simple and easy to use. The two most prominent Linux-based desktop managers, Gnome and KDE, serve as the best example of two extremes. Gnome's approach is to hide as much information as possible from the user, and to hide everything behind menus. For example, the latest version of Gnome has an icon on the desktop labeled simply "Web Browser." Their philosophy is that users don't need to know which web browser they are using, as long as it allows them to browse the Web as they please. KDE, on the other hand, comes with at least three different web browsers, all of them hidden under the same menu. I believe that Gnome's approach is the better one as long as it gives users the ability to change what browser to use, which it does.

Back to Perl. Perl's most famous motto is all about choice: "There's More Than One Way To Do It." Larry Wall realized that different people think differently and, therefore, might try different routes to achieve the same goal, so he gave Perl users different tools and the choice to use the ones they liked. For example, there are two versions of the binary logical operators—`&&` and

and, *//* and *or*, *!* and *not*—that have slightly different precedence strengths. Although just one set of those operators together with proper use of parentheses is enough, Perl gives you the choice. This allows code of the form:

```
$flag && open FH, $file or die $!
```

which attempts to open the file *\$file* only if *\$flag* evaluates to True, and will only *die* if the *open()* fails. Simply changing the *or* to *//* will change the meaning of the code, and will require parentheses in order to produce the correct behavior. Another example is the introduction of *unless* and *until*, which are equivalent to *if not* and *while not*, respectively. Although they are not necessary, having those two extra control operators can tremendously increase code readability.

Feedback

Designers often forget that they are developing things for others to work with. Thus, it is very important that they listen to their end users and take their comments and concerns very seriously. I often find that when working on a long project that takes more than a few days to finish, I start to get impatient and want to release something quickly. So I start to cut corners and relegate many features to the “ToDo” status and end up releasing what is really a half product. A few months ago, I read very similar comments on different newsgroups by other people who develop open-source software. That is why

most open-source programs have this feeling of still being in production, that there is something unfinished about them. This is probably the most important lesson of them all.

Feedback is valuable. There are a lot of smart people who could give you interesting ideas that you might not be able to

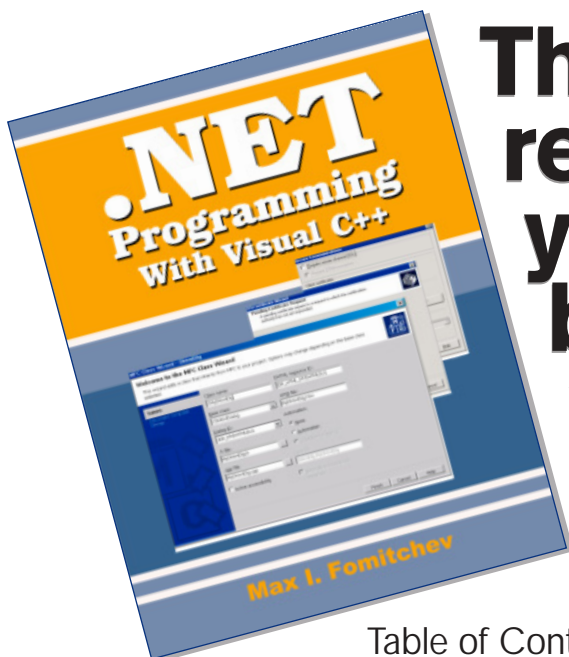
come up with by yourself. For example, one of the testers of my GUI builder suggested that the dumped Perl code be wrapped up as a Perl/Tk mega widget, which could be instantiated via user-specific code. He went so far as to give me a specific example of how to do it, and I thought it was a great idea. Now the GUI builder can dump either standalone code, or a mega widget in the form of a Perl module. That is one use I didn’t plan for that came almost for free.

Perl is fortunate enough to have a large community around it, which generates a tremendous amount of feedback. I wouldn’t imagine such a large community

would have been possible if Larry hadn’t listened to their needs in the first place, and the Perl pumpkins still treat each concern with care. As an example, when the Perl6 project was announced, over 350 RFCs were submitted to Larry for consideration, and he took (and indeed is still taking) account of each one of them in his Perl6 design. That is a sure recipe for success.

TPJ

I have heard countless jokes and even read some serious research papers on how to set VCR clocks



The .NET resource you've been waiting for!

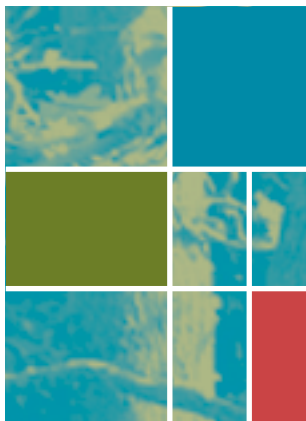


- Delivered in PDF format.
- Packed with C++ code examples.
- Thousands of lines of source code.
- A complete reference to the .NET Framework

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).



HTML Calendars

brian d foy

Instead of having a long list of bookmarks, I wanted to create a web page with links to the Internet radio shows that I listen to each day. It turned out not to be as simple as I thought. I had to find the real links to the programs so I didn't have to go through JavaScript redirection hacks, and I ended up displaying a month's worth of links on the same page. Working with dates tends to be tedious and annoying, but *Date::Simple* and *HTML::Calendar::Simple* made it, well, simple.

Building the Links

I had started with a script to create a page of the current day's links, but then I wanted links to the previous day, too. Then I wanted the previous week. Then I decided to display the entire month. Like a lot of small programs, this one got a lot bigger. Doing all this myself is something I wanted to avoid, so I looked around CPAN and found *HTML::Calendar::Simple*. Perfect: It would make the calendar and allow me to add linked text to each day's cell. I wouldn't have to worry about breaking up a month into weeks or working with months that start in the middle of the week.

Before I could create HTML links, I needed to figure out what they were. Most of the shows I want to listen to are on National Public Radio (NPR), which used to link directly to their Real Audio streams. Now they go through some JavaScript magic. I don't want to click on several links to listen to a show, and I don't want to remind NPR to ask me if I want to use RealPlayer or Windows Media Player every time. I just want to listen to the show.

With a little digging through the NPR web site, I discovered how to make the actual links. I examined the JavaScript functions and the CGI parameters, then turned that same logic into the *npr_ram_href()* subroutine in my *npr_calendar* script (Listing 1). The subroutine needs the date, along with a "program code" that NPR assigns to each show. The program code is the first argument and the date, as a *Date::Simple* object, is the second argument. Since NPR uses this same system for all of its shows, I can use this for all of the NPR shows I want to listen to. I only need to tell the subroutine which show I want, using its program code, and the date I want, which is usually the current day, but sometimes the day before if I missed a show.

I also like to listen to Marketplace from Public Radio International (PRI), which does things a bit differently. I still need the date, but instead of calling the URL directly, I call a CGI script that does it for me. The Marketplace.org server sends me a SMIL file (a Real media playlist file) that my browser, Firefox, does not like, so I use

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

the CGI script to fix it before I send it back to my browser. All of that logic shows up in the *pri_ram_href()* subroutine.

Now that I know how to make the links, I need to figure out how to put them into the calendar. Some of the shows play only on weekdays, and some on the weekend. I don't want to simply put every link in every cell: Each day should have only links to that day's shows.

Eight Days a Week

After trying a couple things, I settled on using bit fields to remember which days of the week that each show plays. I had started with a lookup table, and then I changed that to an array, but the code to look up values dominated the script. Since there are seven days in a week, I can use 1 byte to represent the whole week, 1 bit per day, with a day left over in case you are the Beatles. Sunday is 0, Monday is 1, and so on. With a bit field, the lookup is just a bit operation.

On lines 23–25 of Listing 1, I create the bit field for shows that play on weekdays. The 0b numeric format introduced in Perl 5.6 helps out by allowing me to write out binary numbers as literals, and as long as I remember to read from right to left, I can use a bit field to represent the days of the week. I added underscores between Sunday and Monday and between Friday and Saturday. Perl ignores the underscores, so they are really just there to help me read the code. The *\$saturday* variable has the high bit set (well, the seventh bit), and *\$sunday* has the low bit set. The *\$weekday* variable has all of the middle bits set. Later, when I need to compare a day of the week to one of these variables, I just use bit operators.

I have most things in place for creating a playlist description. I use an array to represent the shows. Each show is an array element that is an anonymous hash. The first element in the hash is the program code, the second element is the appropriate bit field, and the last element is the subroutine reference to create the right reference for that show.

Lines 36–38 set up the basic calendar. I use the *today()* function from *Date::Simple* to get today's date. I could just as easily use *localtime()* here (although I need to add 1900 to the year and 1 to the month), but I already need *Date::Simple*, and it looks much nicer than the *localtime()* gymnastics. I use *today()* in a *map {}* and go through a list of methods to call on the *Date::Simple* object that *today()* returns. I assign the resulting list to *\$year* and *\$month*.

I run this script just after midnight at the beginning of each month, so I only generate the calendar once a month, but if I wanted to turn this into a CGI script that always dynamically generates the calendar for other months, I could get the *\$year* and *\$month* values from the CGI parameters just as easily.

Generating the Calendar

Once I have the year and the month, I can create the calendar object with *HTML::Calendar::Simple*, on line 38. Its *new()* method

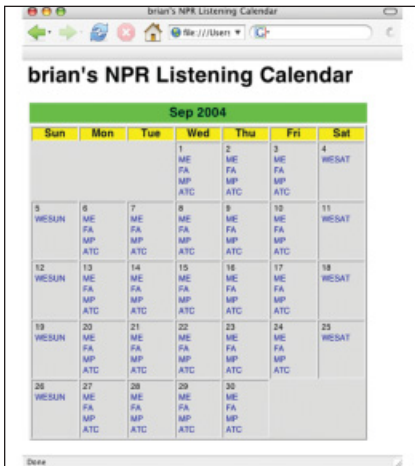


Figure 1: HTML output from the calendar script.

figuring out leap years), I use the `days_in_month()` function to figure it out. This `Date::Simple` stuff is pretty slick!

Inside the DAY loop, I need to do some date manipulations, so I take `$year` and `$month`, which I assigned outside the loop, along with the loop variable `$day`, to get a `Date::Simple` object using its `ymd()` constructor. It doesn't get much simpler than that. I immediately turn that into the day of the week value using the `day_of_the_week()` method. I turn that into a bit mask by left shifting by 1 the index for the day of the week. Again, Sunday is 0, Monday is 1, and so on, just like the positions in my bit fields.

On line 46, in the SHOW loop, I go through each show. On the next line, I check if the show plays on that day by AND-ing the bit mask with the second element of the show array. That operation only returns True if the bits in the `$day_of_week` mask are also set in `$show->[1]`. If the operation does not return True, I skip to the next show. Only the shows that play on that day get to continue with the loop.

takes a hash reference with keys for the year and the month. Now I just need to add the text for each date.

I need to add links to each of the days in that month, so on line 41, I run through each day with `foreach()`, which I also label with "DAY" so I can refer to it easily. I need to know how many days to go through, and `Date::Simple` comes to the rescue again. Instead of creating my own hash that knows how many days are in each month (and also

On line 49, I use the program code in `$show->[0]` along with the date as arguments to the anonymous subroutine, which is the last element of the show array, `$show->[2]`. Although my shows list only has two sources for shows, I could easily add more (like "Car Talk" maybe) without making this loop more complicated.

In the next statement, on line 51, I add the link to the calendar with the `daily_info()` method. I pass it an anonymous hash in which I specify the day and the link text. The link text that actually shows up in the HTML has its own key (in case I want to change it), so I just use the show's code. As I go through each show, I might add more links to a single day, and I can add as many as I like.

When I have finished this short loop, I create some HTML, and from within my HERE document, I call the `calendar_month()` method on my `$calendar` object to get back an HTML table that represents the calendar. I create it inside an anonymous array that I wrapped in an array dereference. It's just a little trick to interpolate a method call. In the HTML that I wrap around the calendar, I include a reference to a stylesheet so I can make things look pretty—the final result is shown in Figure 1.

That's it. It took me longer to set everything up than it did to actually make the calendar. With everything I actually did in this script, I created very little date or calendar logic myself. Most of the script dealt with creating the links for each show. The `Date::Simple` module handled all of the date processing, and `HTML::Calendar::Simple` handled most of the HTML bits. I concentrated on the bits that mattered to me: Creating direct links to the shows and listing all the shows I wanted to listen to. When those modules say "Simple," they mean it.

References

- <http://www.npr.org/>
- <http://www.marketplace.org/>
- <http://search.cpan.org/dist/Date-Simple/>
- <http://search.cpan.org/dist/HTML-Calendar-Simple/>

TPJ

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

Listing 1

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

use Date::Simple qw(:all);
use HTML::Calendar::Simple;

my $npr = sub {
    "http://www.npr.org/dmg/dmg.php?prgCode=" .
    $_[0]
    "&showDate=" .
    $_[1]->format( "%d-%B-%Y" )
    "&segNum=&mediaPref=RM"
};

my $pri = sub {
    "http://www.panix.com/~comdog/marketplace.cgi/marketplace.smil?" .
    "http://www.marketplace.org/play/audio.php?media=" .
    $_[1]->format( "%Y/%m/%d" )
    "_mmp"
};

my $weekdays = 0b0_1111_0; # read from right to left. low
my $saturday = 0b1_00000_0; # bit it sunday, high bit is
my $sunday = 0b0_00000_1; # saturday

my @shows = (
    [ ATC => $weekdays, $npr ], # All Things Considered
    [ ME => $weekdays, $npr ], # Morning Edition
    [ FA => $weekdays, $npr ], # Fresh Air
    [ WESAT => $saturday, $npr ], # Weekend Edition Saturday
    [ WESUN => $sunday, $npr ], # Weekend Edition Sunday
```

```
[ MP => $weekdays, $pri ], # Marketplace
);

my( $year, $month ) = map { today()->$_ } qw(year month);

my $calendar = HTML::Calendar::Simple->new(
    { year => $year, month => $month } );

DAY: foreach my $day ( 1 .. days_in_month( $year, $month ) )
{
    my $date = ymd( $year, $month, $day );
    my $day_of_week = 1 << $date->day_of_week;

    SHOW: foreach my $show ( @shows )
    {
        next SHOW unless $show->[1] & $day_of_week;
        my $link = $show->[2]( $show->[0], $date );

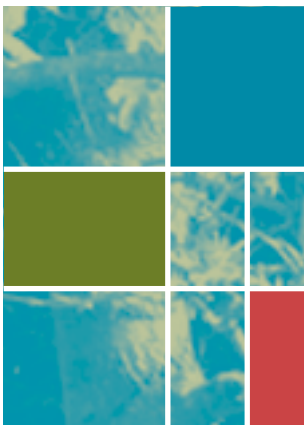
        $calendar->daily_info( {
            day => $day,
            $show->[0] => qq|<a href="$link">$show[0]</a>|,
        } );
    }
}

print <<"HERE";
<html><head><title>brian's NPR Listening Calendar</title>
<link rel="stylesheet" type="text/css" href="npr_calendar.css" />
</head><body><h1>brian's NPR Listening Calendar</h1>

@[ [ $calendar->calendar_month ] ]

</body></html>
HERE
```

TPJ



Musical Archaeology With Perl

Simon Cozens

I've recently moved house, and one of the joys of moving house is that you occasionally turn up things you'd forgotten existed. One of the most bittersweet things I turned up was a backup CD of my computer from about 1998. The sense of morbid curiosity was overpowering, and after wading through the pictures of ex-girlfriends, university-era essays, and curious Windows applications, I found a directory that was itself a backup of another computer. From 1994.

At the time I was in high school, and was a very prolific music composer. Looking at the titles of the song files and remembering the songs I wrote at the time, I realized that one or two of them weren't all that bad, and I'd quite like to listen to them again. This is when I remembered that I didn't use real computers in 1994. The files came from an Atari ST and were written by a program called Notator SL.

Searching the Web, the only two ways I could find to use these files again were either to find an Atari ST already running Notator, or buy a copy of Notator's grandchild, Logic (formerly from eMagic, now owned by Apple). Logic is expensive, and the chances of finding anyone in my neighborhood with a 10-year-old music computer set-up were pretty slim.

When something like this happens, it takes me over. I have to find a solution. Besides, it's more fun than packing. There was only one thing for it: I was going to have to decode the file format myself.

Laying the Foundations

Since I had a lot of these .SON files, the first step was to take a couple of them, and discover how they differed. Thankfully, I knew that I had a few files that were very similar in nonessential regards—the drum mappings were the same, the track names and parameters were the same, and only the actual notes played were different. So I sat down with a hex editor and each of the files open in its own window, and wrote out what I saw (see Figure 1).

For the time being, all I cared about was the data on each track. With the track name appearing in plain text, 8 bytes padded with spaces, I knew where the tracks began. Unfortunately, I couldn't work out where a given track ended. I could see that "PIANO"

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-cozens.org.

was a track name and started a new track, but I couldn't make a rule for that to explain it a computer.

So I came at things from a different angle. I knew that the data was going to be something like MIDI data—a series of n -byte messages. So the first thing I needed to do was work out the value of n . We can do this by spotting patterns. I wrote something quick to dump the output in rows of n bytes:

```
open IN, shift or die $!;
seek IN, 0x5ae0, 0;
my $track;
read(IN, $track, 8) and print "Track name: '$track'\n";
my $n = 3;
my $data;
while (read(IN, $data, $n)) {
    my @bytes = map ord, split //, $data;
    print join " ", map { sprintf "%x", $_ } @bytes;
    print "\n";
}
```

```
0x0 - 0x2a60: Identical
0x2a60 - 0x2a60: Character set?
0x2ce0 - 0x2d5f: ?
0x2d60 - 0x2f5f: Drum pattern?
0x2f60 - 0x335f: Copyright
0x3360 - 0x348f: Identical
0x3490 - 0x381f: DATA
    0x7f 0xf0 - identical until 0x3490,
    then 0x7f 0xbe in one, still 0xf0 in another
    until 0x3510
in another:
    0x3820: 0x80 0x7f until 0x3c00
```

```
0x3c00 - 0x3f00: Fader descriptions
0x3f60 - 0x4750: Fader names?
0x4758 - 0x48af: Font data
0x48b0 - 0xf91f: DATA
0x4920 - 0x5ac8: Grooves and empty space
0x5ac9 - 0x5acf: DATA
```

TRACKS

```
0x5ae0 offset of first track
0x5ae0-5ae8: Track name
Track ends with 14 0x0s? 9 0x0s? 16?
```

Figure 1: First pass at analyzing the file structure.

We loop over the file, reading in $\$n$ bytes at a time into $\$data$. Then these are split into their individual bytes—*split* // splits every single character into a separate array element. We convert them to their ASCII code, turn that into hex, and then dump them out.

I started with $\$n = 3$ because a lot of MIDI commands are 3 bytes, and that seemed like a good guess. I then ran it on the shortest file I could find, and got this:

```
Track name: '*Undo* '

0 0 9c
22 20 59
58 0 80
21 0 0
2 23 2d
90 21 0
81 0 39
90 21 0
81 0 3d
90 21 0
81 0 40
90 21 0
```

Not bad. We have a few rows that I guessed were to do with setting up, then we settle into a pattern of 90 21 0 and then 81 0 3x. These could be something to do with notes, I guess. But the way that they alternate like that, maybe the two rows are part of the same event. We set $\$n = 6$, and we see:

```
Track name: '*Undo* '

0 0 9c 22 20 59
58 0 80 21 0 0
2 23 2d 90 21 0
81 0 39 90 21 0
81 0 3d 90 21 0
81 0 40 90 21 0
81 0 2d 90 23 ff
1 0 39 90 23 ff
1 0 3d 90 23 ff
1 0 40 90 23 ff
1 0 31 90 24 0
```

Oh, now this is a bit more interesting. We notice that the last 2 bytes form a number that is monotonically increasing. What could monotonically increase over events in a track? Maybe that's the time that the event occurs. The fourth column is almost always 90; maybe that signifies that this is a note.

There's a way to check—we run it on a file with a drum part as the first track, and, given that I vaguely remember what the song sounds like, I can work out the interval between drum beats. That will tell us how to convert between 21 0 and some measure-beat-tick value (or bar-beat-tick, since I'm from the UK and we use different terminology over here). Here's a bit of the dump of a drum track:

```
01 00 24 90 36 00
81 00 2e 90 36 00
81 00 36 90 36 00
81 00 24 90 36 5f
01 00 2e 90 36 5f
01 00 36 90 36 5f
01 00 36 90 36 60
81 00 36 90 36 bf
01 00 26 90 36 c0
81 00 2e 90 36 c0
81 00 36 90 36 c0
```

Now we only see two values for the first row: 81 and 01. We'll assume that these are somehow paired, so we'll only look at the 81 events for now. We'll also only concentrate on unique values of the last two columns, and we get a sequence like this:

```
36 00 / 36 5f / 36 bf / 36 c0 / 37 1f / 37 7f
37 80 / 37 df / 38 3f / 38 40 / 38 9f / 38 ff
39 00 / 39 5f / 39 bf / 39 c0 / 3a 1f / 3a 7f
3a 80 ...
```

I happen to know that there's a tambourine hit every quaver (that's English for an eighth-note) in this drum track, and there's an increment of 0x5f (96) between every note. That's when it all came back to me—Notator uses 768 ticks in a measure. An eighth of 768 is 96. So if we take these 2 bytes as representing a 2-byte integer, we can convert them to musical time, like so:

```
ub tick2time (
    my ($hi, $lo) = @_;
    my $ticks = (256*$hi) + $lo;
    my $bar = int($ticks / 768);
    $ticks %= 768;
    my $beat = int($ticks / 192);
    $ticks %= 192;
    "$bar/$beat/$ticks";
)
```

Now we can improve our dumping tool a little:

```
while (read(IN, $data, $n)) {
    my @bytes = map ord, split //, $data;
    print join " ", map { sprintf "%02x", $_ } @bytes[0..3];
    print " [".tick2time(@bytes[4..5])."]";
    print "\n";
}
```

This gives us the beginning of a drums track, like so:

```
00 00 9c 22 [10/2/152]
18 00 80 a1 [0/0/0]
02 23 2e 90 [12/3/96]
81 00 2e 90 [12/3/191]
01 00 24 90 [13/0/0]
81 00 2e 90 [13/0/0]
81 00 31 90 [13/0/0]
81 00 36 90 [13/0/0]
81 00 24 90 [13/0/95]
01 00 2e 90 [13/0/95]
01 00 31 90 [13/0/95]
```

From this we can tell that the first two lines really aren't related, and that the song appears to start 10 measures in. We'll ignore the latter detail for now, since it's just cosmetic—we can shift all the tracks back to the start in a more sophisticated sequencer—and concentrate on those two lines of set-up data. The beginning of the third line starts to look a little suspect too: Every other line starts 81 00 or 01 00.

Perhaps what's actually going on is that the data starts at 2e 90, and we have not 12 but 14 bytes of setup. Then we have one number that fluctuates a bit; 90 for a note, the time, then some other numbers.

This requires some major changes to our dumper script:

```
read(IN, $track, 8) and print "Track name: '$track'\n";
my $setup;
read(IN, $setup, 14);
while (read(IN, $data, $n)) {
```



```

my @bytes = map ord, split //, $data;
print join " ", map { sprintf "%02x", $_ } @bytes[0..1];
print " [".tick2time(@bytes[2..3])."] ";
print join " ", map { sprintf "%02x", $_ } @bytes[4..5];
print "\n";
}

```

And we now see:

```

Track name: 'Drums'
2e 90 [12/3/96] 81 00
2e 90 [12/3/191] 01 00
24 90 [13/0/0] 81 00
2e 90 [13/0/0] 81 00
31 90 [13/0/0] 81 00
36 90 [13/0/0] 81 00
24 90 [13/0/95] 01 00
2e 90 [13/0/95] 01 00
31 90 [13/0/95] 01 00
36 90 [13/0/95] 01 00

```

Ah, now this is looking promising. But now what? This is the curse of reverse engineering—you solve one piece of the problem, but then you’re back to square one for the other pieces until inspiration strikes again.

MIDI Inspiration

Inspiration struck again while I was reading Sean Burke’s MIDI-Perl documentation. I figured I needed to translate these files into MIDI files to do anything with them, so I took a look at how to produce MIDI files. I learned two things there—first, MIDI files don’t encode note length, but have paired “note on” and “note off” events. Maybe that’s our paired 81 and 01 events.

Second, the MIDI Perl module exports a number of hashes, including `%number2note` and `%notenum2percussion`, which turn a MIDI file’s representation of a pitch or percussion instrument name into an English representation. Maybe even if Notator didn’t exactly store its events in MIDI file format, it at least used the same representation for pitch. So we take the first column of our dump, the one that bobbles about a bit, and feed it through one of these hashes:

```

my @bytes = map ord, split //, $data;
print $track =~ /drum|percuss/i ?
    $MIDI::notenum2percussion{$bytes[0]} :
    $MIDI::number2note{$bytes[0]};
print join " ", map { sprintf "%02x", $_ } $bytes[1];
print " [".tick2time(@bytes[2..3])."] ";
print join " ", map { sprintf "%02x", $_ } @bytes[4..5];
print "\n";

```

This turns our drum track into:

```

Bass Drum 1 90 [17/2/0] 81 00
Open Hi-Hat 90 [17/2/0] 81 00
Tambourine 90 [17/2/0] 81 00
Bass Drum 1 90 [17/2/95] 01 00
Open Hi-Hat 90 [17/2/95] 01 00
Tambourine 90 [17/2/95] 01 00
Bass Drum 1 90 [17/2/96] 81 00
Tambourine 90 [17/2/96] 81 00
Bass Drum 1 90 [17/2/191] 01 00
Tambourine 90 [17/2/191] 01 00

```

This looks good enough, and we can check to see if an ordinary music track is more or less in key:

```

Track name: 'Vocals'
E5 90 [10/3/144] 81 00
E5 90 [10/3/191] 01 00
G#5 90 [11/0/0] 81 00
G#5 90 [11/0/47] 01 00
G#5 90 [11/0/48] 81 00
G#5 90 [11/0/191] 01 00
F#5 90 [11/1/0] 81 00
F#5 90 [11/1/95] 01 00
E5 90 [11/1/96] 81 00
B5 90 [11/1/144] 81 00
E5 90 [11/1/191] 01 00
B5 90 [11/2/47] 01 00

```

This is consistent with a song in E major. Finally, I noticed that for some songs, as well as 81 and 01, there were note events with other values for this column; I guessed that this related to the velocity of the note, another MIDI concept. Velocity represents conceptually how hard the note is struck; it’s a bit like volume, but can also change the timbre of the tone. For now, though, we’ll take numbers more than 0x80 to mean note on with full velocity, and less than 0x80 to mean note off.

We have everything we need to convert a single track into MIDI format—except we still don’t know how a track ends. Our notes say that it ends with some number of 0 bytes, but we don’t know how many. It was back to the hex editor.

After comparing files and tracks in the hex editor once again, I came up with an idea—what if the track name wasn’t the first bit of data in the track? What if there was some other setup data before the name of each track? That section starting just before the tracks that I’d labeled as “DATA” might be part of the track header. This would mean that the stray null bytes I’d been seeing were not the end of the track, but the beginning.

This cracked it—I found that each track began with either the 4 bytes `7f ff ff ff` or `00 0f ff ff`. Then there was a 24-byte header, followed by the track name, and the 14 other header bytes I had determined earlier. Now I could write the MIDI file translator.

The Translator

First, I decided to read the data in with the Perl slurp operator and regular expressions, instead of the more cumbersome `read`. This allowed me to use *split* to split up the tracks on the boundaries that I’d just discovered. I also decided to get the data together first, then pass over it, converting it to MIDI tracks. Here’s the part that reads in the tracks:

```

my ($input, $output) = @ARGV;
open IN, $input or die $!;

seek IN, 0x5ac8, 0;
local $/;

my $boundary = qr/(?!\x7f\xff\xff\xff)|(?!\x00\x0f\xff\xff)/;
my @lines = split /$boundary/, scalar <IN>;
my @tracks;

for my $track (@lines) {
    my $stuff = {};
    $stuff->{header} = substr($track, 0, 24, "");
    $track =~ s/^(.{8})//; $stuff->{title} = $1;
    $stuff->{data} = $track;
    push @tracks, $stuff;
}

```

Now we need to convert our Notator events to MIDI events. There are a couple of things we have to know about MIDI events first, however. First, although we have “absolute” times, in terms of measures, beats and ticks, MIDI files actually deal in terms of delta time—that is, an event is placed a number of ticks after the

previous event. This means we need a counter to keep track of where we're up to in the track.

Next, whereas Notator deals in terms of tracks and events on a track, a MIDI file has both tracks and channels. We need to assign a channel to each track, and then spit out that channel number with each of the track's events. Finally, MIDI events are binary data, but the MIDI-Perl distribution makes it relatively easy to construct them by allowing us to specify them as an array of arrays. For instance, the first event we want to spit out is:

```
[ track_name => 0 => $track->{title} ]
```

This puts the track name at time-position zero in the track. Note that events look like this:

```
[ note_on => $time, $track->{channel}, $note, $velocity ]
```

We want to build up an array of these events, and then turn them into a *MIDI::Track* object. Here's how we do it.

```
my $channel = 0;
my @midi;
for my $track (@tracks) {
    $track->{data} =~ s/.[14]//; # Rest of header
    next if length($track->{data}) == 0;
    print $track->{title}. "\n";

    $track->{counter} = 0;
    $track->{channel} = $channel++;
    $track->{events} = [
        [track_name => 0 => $track->{title} ],
    ];
    my $size = 6;
    while (my $event = substr($track->{data}, 0, $size, "")) {
        push @{$track->{events}}, data2event($track, $event);
    }
    my $midi_track = MIDI::Track->new;
    $midi_track->events(@{$track->{events}});
    push @midi, $midi_track;
}
```

Notice that we're treating *\$track* a little like an object, which stores all its own data inside the hash reference; it knows where we are in the song (*\$track->{counter}*), the track's channel, the MIDI events, and so on. The only mystery is *data2event*, which turns the 6 bytes of data into an array reference representing a MIDI event:

```
sub data2event {
    my $track = shift;
    my $line = shift;
    my ($note, $status, $pos1, $pos2, $vel, $arg3) =
        map ord, split//, $line;
    if ($status == 0x90) {
        $status = "note_on";
        $vel = $vel - 0x80;
        if ($vel < 0) { $status = "note_off" }
        $vel = 127;
        my $pos = $pos1*256 + $pos2;
        my $delta = $pos - $track->{counter};
        $track->{counter} = $pos;
        return [ $status, $delta, $track->{channel}, $note, $vel ]
    }
    warn "Skipping over unknown event $status ($note, $vel, $arg3)
        at position ".$tick2time($pos1,$pos2);
    return;
}
```

Notice that at present, we don't know what *\$arg3* is for. However, we do now know that if we see any events that aren't 0x90, then we get a warning. We'll come back to this in a moment. To finish off, we now have an array of *MIDI::Track* objects.

Turning these into a MIDI file is now easy:

```
$song = MIDI::Opus->new(
    { 'format' => 1, 'ticks' => 192, 'tracks' => \@midi }
);

warn "Writing on $output";
$song->write_to_file($output);
```

This creates a MIDI format 1 file, with a rather arbitrary tempo, and fills it with our tracks.

I ran it on one of my old compositions, and out popped a working MIDI file. After a few minutes, fiddling around with instrumentation on a rather more modern sequencer, my 10-year-old music file was playing in all its glory.

Later Improvements

Well, most of its glory. I got quite a few of those warnings telling me it had skipped over some events. Thankfully, it was now much, much easier to work out what those events should be.

So when the bass guitar was supposed to slide up and down but instead we got a load of warnings about an unknown event 224, we can guess that 224 means "pitch wheel." I used the original dumper and grepped for events with a code of 224, and found that the velocity moved around, centering on 128. MIDI files, on the other hand, encode pitch wheel changes not on a scale of 0 to 255 but a scale of -8192 to 8192, so I needed to do a bit of scaling. It was then easy enough to drop another stanza into *data2event*.

```
elsif ($status == 224) {
    my $pitch = ($vel - 129) * (8192 / 128);
    my $pos = $pos1 * 256 + $pos2;
    my $delta = $pos - $track->{counter};
    $track->{counter} = $pos;
    return [ "pitch_wheel_change", $delta, $track->{channel}, $pitch ]
}
```

Now there was one final problem. Songs longer than 85 measures were getting truncated, and I was getting lots of warnings about event 145. I thought about this, and realized that the 2-byte position counter could only go up to 65535 ticks, and 65535 ticks was just over 85 measures. Then we flip over from a note being event 144 to being 145—it seems that Notator used some bits in the "event type" byte to extend the position counter. This is pretty hateful, but I suppose it's better than restricting everyone to short songs. The conversion program had to change, like so:

```
my ($note, $status, $pos1, $pos2, $vel, $arg3) = map ord, split//, $line;
if ($status == /(14[45])/) {
    $pos1 += 256 * ($1 - 144);
```

This still doesn't cover everything that can happen inside a Notator file, but at least it gets the notes out, and it's enough for me to play around with those old songs again. If, by some strange chance, you have a bunch of Notator songs and a sense of nostalgia, you can get my converter from <http://simon-cozens.org/programmer/releases/son2midi.pl>. But beware—next time you get stuck in the early nineties, Perl might not be able to drag you back...



Mac OS X Panther Hacks

Kevin Carlson

Technical book publishers must have some market research that tells them that books with the word “hacks” in the title sell well. There are a lot of “hacks” books out there right now, including *Mac OS X Panther Hacks* by Rael Dornfest and James Duncan Davidson.

Whether or not the tips in this book qualify as “hacks” depends on how you define a “hack.” The book seems to (mostly) define a hack as an add-on program, or a tweak to a hidden or otherwise nonobvious system setting. And if that’s what you’re looking for—a way to enhance your day-to-day interaction with Mac OS X one quick and easy step at a time—you will probably get something out of this book.

The danger in buying one of these “tips and tricks” books is that you won’t get much bang for the buck in the short term—that maybe one or two of the items in it will interest you, and the rest will turn out not to be useful. And at \$29.95, this book could be a real risk in that department. But I found enough gems in this book to justify the price, and it’s important to remember that over time, you may come back to this book to investigate things that didn’t grab you the first time around.

The tips presented in *Mac OS X Panther Hacks* are contributed by a host of people—the authors really act more as editors than writers. This leads to a slightly discontinuous style from section to section, but also means that there will be something useful in the book for almost anyone using OS X. And this isn’t the sort of book you’re likely to read in a continuous fashion, anyway. It’s a book you’ll skim until you see something that looks useful.

The “hacks” are divided into nine chapters. Some of these cover interface enhancements, like launchers and desktop pagers, some cover scripting with Applescript, Perl and Python, and some cover interfacing with gadgets such as PDAs and cell phones. In each of these chapters, I found at least a couple things that I ei-

Kevin is TPJ’s Executive Editor. He can be reached at kcarlson@tpj.com.

***Mac OS X Panther Hacks: 100
Industrial-Strength Tips & Tools***
*Rael Dornfest and
James Duncan Davidson*
O’Reilly Media, 2004
566 pp., \$29.95
ISBN 0596007183

ther didn’t know about, or just hadn’t considered doing before. Sometimes, the book will simply point out a useful freeware or shareware system utility. You might ask why you need a book to point these out for you. It’s true that these sorts of programs are easy enough to find on your own, but I confess, I never would have considered using most of these without the book’s thoughtful pro-and-con analysis. Almost everything in the book can be found on the Web, but there is real value in the authors’ filtering and aggregating of these items. Every tip in the book seems to have been tested—everything I tried worked, unlike the “tips” you often find on the Web.

Most of the book is aimed at the nonprogrammer. Where there is code, there are also exact step-by-step instructions for making that code work. The authors have also ensured that any command-line invocations are clearly spelled out for people who don’t spend much time in a terminal window.

Some of the hacks are utterly simple, one-line commands that change default settings on your Mac (such as changing the page-rendering behavior of Safari), and others are multi-step processes that really would be better termed “projects”

(like creating an automated web photo gallery using Perl and your iPhoto library). Some hacks serve a real purpose, like getting your Bluetooth phone to use your Mac's Internet connection, rather than the phone's decidedly pricier web access. Others are almost totally frivolous, like covering your desktop with album-cover images from iTunes. Some of the hacks are just convenient starting points for larger explorations of your system's capabilities. For instance, PHP is there on your Mac, waiting to be turned on. Apache-savvy readers already know this and know how to turn it on. But for others, simply being shown how to add the proper lines to httpd.conf and restart Apache can open up a whole world of possibilities that they didn't know existed.

The chapters on networking, servers, and system administration contain some of the more advanced techniques in the book. There are some very useful things here for anyone who is in charge of a growing network of Macs. Experienced IT personnel will already know most of this material, but folks setting up a collocated web server machine for the first time, or who are just beginning to expand their home network to manage a small office network of Macs, will benefit from the targeted approach to topics like accessing remote desktops, or setting up a Postfix mail server.

The book also has lots of advice about how to do effective backups on your Mac. It briefly covers some of the commercial solutions, but then quickly dives into the many ways to accomplish backups for free, using the tools already installed by default. It turns out many commercial and shareware backup solutions are just window dressing—under the hood, they actually just use the default system tools. So it pays to learn how your Mac's built-in utilities like ditto and hdiutil work. With a tiny bit of scripting and a crontab entry, you've got a fully automated backup system for no money at all.

Perhaps the best and least tangible benefit of *Mac OS X Panther Hacks* is that it sparks interest in exploring areas of OS X that you might not have previously considered. OS X's GUI hides the fact that there's a full BSD subsystem waiting to be put to work, and even if you know this, it's easy to forget that accessing that capability can be very easy if you know where to look.

TPJ

Every tip in the book seems to have been tested—everything I tried worked, unlike the "tips" you often find on the Web

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.
- ✓ **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 Math Power Newsletter.** Join Homer B. Tilton and expand your base of math knowledge.
- ✓ **Dr. Dobb's Active Scripting Newsletter.** Find out the most clever Active Scripting techniques from Mark Baker.

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

Source Code Appendix

brian d foy "HTML Calendars"

Listing 1

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

use Date::Simple qw(:all);
use HTML::Calendar::Simple;

my $npr = sub {
    "http://www.npr.org/dmg/dmg.php?prgCode=" .
    $_[0] .
    "&showDate=" .
    $_[1]->format( "%d-%B-%Y" ) .
    "&segNum=&mediaPref=RM"
};

my $pri = sub {
    "http://www.panix.com/~comdog/marketplace.cgi/marketplace.smil?" .
    "http://www.marketplace.org/play/audio.php?media=/" .
    $_[1]->format( "%Y/%m/%d" ) .
    "_mmp"
};

my $weekdays = 0b0_11111_0; # read from right to left. low
my $saturday = 0b1_00000_0; # bit it sunday, high bit is
my $sunday = 0b0_00000_1; # saturday

my @shows = (
    [ ATC => $weekdays, $npr ], # All Things Considered
    [ ME => $weekdays, $npr ], # Morning Edition
    [ FA => $weekdays, $npr ], # Fresh Air
    [ WESAT => $saturday, $npr ], # Weekend Edition Saturday
    [ WESUN => $sunday, $npr ], # Weekend Edition Sunday
    [ MP => $weekdays, $pri ], # Marketplace
);

my( $year, $month ) = map { today()->$_ } qw(year month);

my $calendar = HTML::Calendar::Simple->new(
    { year => $year, month => $month } );

DAY: foreach my $day ( 1 .. days_in_month( $year, $month ) )
{
    my $date = ymd( $year, $month, $day );
    my $day_of_week = 1 << $date->day_of_week;

    SHOW: foreach my $show ( @shows )
    {
        next SHOW unless $show->[1] & $day_of_week;
        my $link = $show->[2]( $show->[0], $date );

        $calendar->daily_info( {
            day => $day,
            $show->[0] => qq|<a href="$link">$$show[0]</a>|,
        } );
    }
}

print <<"HERE";
<html><head><title>brian's NPR Listening Calendar</title>
<link rel="stylesheet" type="text/css" href="npr_calendar.css" />
</head><body><h1>brian's NPR Listening Calendar</h1>

@{ [ $calendar->calendar_month ] }

</body></html>
HERE
```

TPJ