

The Perl Journal

Creating RSS Files with XML::RSS

Derek Vadala • 3

Making a Cross-Platform Installer with Perl

Max Schubert • 6

Perl & Rapid Database Prototyping

Tim Kientzle • 10

Ruby vs. Perl

brian d foy • 13

Mining Mail

Simon Cozens • 15

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Cameron Laird:

Programming Perl in the .NET Environment • 19

Source Code Appendix • 20

LETTER FROM THE EDITOR

The Installer Question

Writing Perl modules is easy, as witnessed by the nearly 4000 modules available on CPAN (the Comprehensive Perl Archive Network). However, writing Perl apps that depend on multiple modules and that must be installed on multiple platforms is not so easy. Sure, Perl is a platform-neutral language (mostly). And sure, we have a great resource in CPAN and its CPAN module that, in theory at least, installs any module you need.

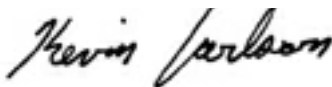
But theory and practice quickly diverge when you try to distribute a Perl app of any significant complexity to a varied group of users. Max Schubert, whose article "Making a Cross-Platform Installer with Perl" appears on page 6, can attest to this. The complexities of third-party module dependencies, platform-specific Perl behavior, and even platform-specific module bugs all became significant hurdles for him on the way to delivering an app.

The reality for most complex Perl applications is that, unless the app is distributed in multiple OS-specific forms, users need some sort of Perl expertise to troubleshoot the installation. Perl's open, extensible, modular form gives you huge flexibility. I don't know about you, but I love knowing that I don't have to reinvent many wheels in my Perl code. But the price we pay for this flexibility is module-dependency complexity. This is why some people have argued that such complexity holds Perl back from true success in the enterprise.

Of course, there have been attempts to address this. James Tillman's PAIX project (<http://www.sourceforge.net/projects/paix/>), for instance, is a cross-platform installer designed to cope with the installation hassles he encountered in his work on the Guido Tk app builder project (<http://www.sourceforge.net/projects/guido/>). PAIX is a fine start, but what we really need is a seamless installation facility built into Perl itself. There are some hints that Perl 6 may provide this. In a May 2002 posting on use Perl, Larry Wall and Damien Conway were asked what modules were likely to be included in the base Perl 6 distribution. Their response was tantalizing:

It's impractical to ship Perl 6 with a thousand modules, so we're seriously considering shipping it with almost none, and instead making the process of downloading and installing appropriate SDKs highly safe and transparent.

Since then, there doesn't seem to have been a lot of discussion about the issue, perhaps because it has gotten lost in larger issues. There are, after all, other criticisms being laid on Perl's doorstep—it's not readable enough, it's too tightly coupled to UNIX, it's too hard to integrate C/C++ libraries. All of these problems limit Perl's ability to scale, and all could potentially derail Perl's adoption. Some people say these issues already have hurt Perl, and that it is losing out to languages like Python because of it. Whether this is true and whether the creators of Perl 6 find a wise and practical solution to the installation question, we know that they face a balancing act—making Perl scalable on one hand, while on the other keeping the flexibility of the "more than one way to do it" approach that has been such an integral part of Perl's success.



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: \$12.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2002 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

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

NATIONAL SALES MANAGER

Caroline Oidem

(650) 513-4491

coidem@cmp.com

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

Perl 6 Documentation List Launches

A new mailing list has been launched to provide user documentation for Perl 6. The team is tackling the Apocalypses in order, defining semantics and providing test cases for Perl 6 functionality as it is decided. As Michael Lazarro, who suggested the list and has been guiding its evolution, wrote: "Project Rule #1: If it isn't documented, it isn't done."

Early discussion on the list centered around style and format. The team is aiming for an informal tone and a cohesive, book-like style as opposed to a "recipe" format. However, it's hoped that the layout will be somewhat more skimmable than the existing Perl 5 documentation: Most pages will be brief and focus on a single subtopic.

You can subscribe to the documentation list by e-mail to perl6-documentation-subscribe@perl.org. The list archives are also online at <http://archive.develooper.com/perl6-documentation@perl.org/>.

Perl Conferences Organized

Yet Another Society has appointed a committee to organize the Yet Another Perl Conferences in Europe. The committee's first announcement was that Paris has been chosen as the venue for YAPC::Europe 2003.

The new committee consists of nominees from past YAPC::Europe conferences: Leon Brocard and Greg McCarroll from London, Ann Barcomb from Amsterdam, and Richard Foley and Norbert Gruener from Munich. Norbert Gruener has been appointed chair of the committee. Questions or comments can be e-mailed to committee@yapceurope.org.

In other conference news, a Geek Cruises ship will be launching for Hawaii on June 1, carrying both the Perl Whirl '03 and the Mac Mania II assemblages. And a month later, OSCON 2003 will be held in Portland, Oregon, from July 7–11.

Win32 5.8 Repositories

ActivePerl 5.8 has been released in beta, and ActiveState is maintaining a repository of modules at <http://www.activestate.com/ppmpackages/5.8-windows/>. Realizing that a few xml-related modules were missing from that list, Randy Kobes announced the creation of an auxiliary repository at <http://theoryx5.uwinnipeg.ca/ppms/>. "Using this repository may not work with self-built perl-5.8s," Kobes warns, "as ActiveState sets the NAME of ARCHITECTURE in the ppd file to MSWin32-x86-multi-thread-5.8, to distinguish it from that of perl-5.6, which isn't binary compatible with 5.8. In such cases, what you could do is grab the .ppd and .tar.gz files, adjust the architecture name to match your own, and then adjust the path to the local .tar.gz file."

Apache Benchmarks Updated

The Apache Hello World Benchmarks, presented by the consultants from Chamas Enterprises, compare the execution times

and memory requirements of various application software running on an Apache server. The first benchmarks were posted in July and included `mod_perl`, `Embperl`, `PHP`, `Python`, `Mason`, `AxKit`, and various other systems; later updates added data for the C Apache APIs to show the maximum execution speed of the benchmarks.

In the first test, an emulation of a heavy web page template "with some application logic and loops, some HTTP parameter passing, and much variable interpolation in the output stream," `mod_perl` was outperformed only by the Apache C API. In a simple "Hello World" test, `mod_perl` came in fourth of 22 platforms tested; it was beaten by the Apache C API, static HTML, and the `mod_include` SSI module. `Mod_perl` led the pack once again in a database benchmark. The final two tests were XSLT benchmarks, in which Resin (<http://www.caucho.com/resin/>) was the best performer. The benchmarking suite, along with a breakdown of the results, can be found at <http://chamas.com/bench/>. Josh Chamas is currently working on benchmarks comparing `mod_perl1` and `mod_perl2`.

Mac::Carbon Released

Mac::Carbon, a collection of modules for accessing the Carbon API under Mac OS X, is now available on CPAN and SourceForge. The modules are an OS X port of the Toolbox collection used in MacPerl. The current release is version 0.01, "but a lot of it works," writes Chris Nandor (Pudge), the MacPerl maintainer. Forthcoming releases will include ports of the Mac::AppleEvents and Mac::OSA modules. More information is available from the macosx@perl mailing list (<http://lists.perl.org/showlist.cgi?name=macosx>).

Parrot BASIC Rewritten

Clinton Pierce has released an update of his BASIC for Parrot, this time using QuickBASIC rather than GW-BASIC as a model. Version 2.0 boasts richer syntax; support for user-defined types; and performance improvements. "The compiler itself is rather hideous Perl, but the output and the runtime libraries are all pure PASM," Pierce wrote in his announcement to the `perl6-internals` list. The 2.0 release is available at <http://geeksalad.org/basic/basic2.zip>.

Comparing Skill Requirements

Keith Barrett decided to compare the relative frequency with which various programming skills are listed as requirements by prospective employers. He searched by keyword on the job listing sites Monster.com and Dice.com, and found that "Perl" usually turned up between 600 and 800 hits. The most popular keywords (UNIX, SQL, and Oracle) turned up about 3000 to 4000 hits, varying by site. "Python" returned only 46 results on each page. The full results are listed at <http://www.advogato.org/article/584.html>.

Creating RSS Files with XML::RSS

In my article, “Parsing RSS Files with XML::RSS,” (*TPJ*, Fall 2002), I covered using XML::RSS to locate, parse, and reuse dynamic content found on the World Wide Web. But what if you want to provide original material for others? XML::RSS can also be used to generate a properly formatted RSS file. You can create feeds on the fly using a database back end, or generate a static RSS file that gets updated at regular intervals.

The <channel> Element

RSS files are composed of various elements that describe a channel (or feed) and its dynamic content. Each item contained within a channel should contain a <title> and <link> element and may also contain the optional <description> element. Likewise, the <channel> element, which stores metadata about the channel, has its own <title>, <link>, and <description> subelements. In addition, the <channel> structure also contains an <items> subelement that provides a table of contents for the RSS document. So a bare-bones RSS <channel> element might look something like Example 1.

In the aforementioned example, our RSS feed contains three items (typical feeds contain about 10 items) and they are indexed by their URL, or in RSS-speak, they are indexed by each item’s <link> element. Begin creating your RSS file by initializing a new RSS object and using the *channel()* method to populate the required items within the <channel> element, as in Example 2.

Only a title, link, and description are required, but other elements are also available. For example, <image> and <textinput> may be used to provide links to a site logo or newsletter subscription form. In addition to these attributes, several modules that extend the base RSS schema are available. These extensions provide elements that can include metadata about a site’s topic, authors, and update frequency, and are categorized as modules that are part of the RSS specification. The Dublin Core module (<http://web.resource.org/rss/1.0/modules/dc/>), for example, includes provisions for information about copyright, publisher, publication date, and language. The Syndication module (<http://web.resource.org/rss/1.0/modules/syndication/>) provides elements that describe how often a feed is updated. I’ll cover a few elements from each module. The specification for each module contains a

comprehensive list of options. A complete list of modules is available from <http://web.resource.org/rss/1.0/>.

Use second-level hashes to compartmentalize RSS module metadata. In Listing 1, I have added a few elements from the Dublin Core and Syndication modules to my channel element.

The Dublin Core elements that I have added are straightforward, but the Syndication elements require a short explanation. The <syn:updatePeriod> specifies a time interval in which to measure the number of updates. In this case, like many RSS feeds, I have chosen one hour. Possible choices are hourly, daily, weekly, monthly, and yearly. The <syn:updateFrequency> specifies how many times the feed is updated during each period. So in this example, the feed is updated four times per hour, or every 15 minutes. The <syn:updateBase>, though it looks a bit confusing, simply represents the first time the feed was published. In this case, November 5, 1999 at 9:00AM Eastern Standard Time.

```
<channel rdf:about="http://www.azurance.com">
  <title>azurance.com</title>
  <link>http://www.azurance.com</link>
  <description>Open Source and Security Consulting</description>
  <items>
    <rdf:Seq>
      <rdf:li rdf:resource=
        "http://www.theregister.co.uk/content/55/27734.html" />
      <rdf:li rdf:resource="http://www.vmunet.com/News/1136204" />
      <rdf:li rdf:resource=
        "http://www.internetnews.com/infra/article.php/1486121" />
    </rdf:Seq>
  </items>
</channel>
```

Example 1: A bare-bones RSS <channel> element.

```
1 use XML::RSS;
2
3 my $rss = new XML::RSS;
4
5 $rss->channel(
6
7     title => 'azurance.com',
8     link => 'http://www.azurance.com',
9     description => 'Open Source and Security Consulting',
10 );
```

Example 2: Using the channel() method to populate the <channel> element.

Derek lives in New York City and works for azurance.com, an open source and security consulting firm that he cofounded. He is the author of the upcoming book, Managing RAID on Linux (O'Reilly and Associates, 2003), and can be contacted at derek@azurance.com.

This information, combined with the update frequency and update period, allows users and applications to determine a publishing schedule.

Some module extensions may be applied to individual items in addition to the `<channel>` element. For example, specifying a `<dc:creator>` for each item is useful for sites that have articles written by more than one author. Add a second-level hash to each item for the modules and subelements that you want to include. Just follow the same examples I used for the `<channel>` element.

Adding Items

Next, I'll add some `<item>` elements using the `add_item()` method (see Listing 2).

In compliance with the RSS specification, the `add_item()` method requires a title and link, but may also include an optional description. Notice how each item corresponds to an entry in the `<rdf:Seq>` metadata from my previous examples. That information is automatically generated by `XML::RSS` as items are added to the RSS object.

Combining DBI and XML::RSS

At this point, it's probably obvious that generating an RSS file using Perl is not much easier than creating one by hand using a text editor. Therefore, creating a reusable program that can automatically generate and update your RSS feed is desirable. While you could use nearly anything on your system as the data source (like text files, DB files, an LDAP server, or a combination of sources), using a back-end SQL database is a popular choice. My `rss_items` function queries a SQL back end (MySQL in my case) and calls `XML::RSS's add_item()` method to populate the RSS object (see Listing 3).

`rss_items` takes a positive integer as input. This number is used to determine how many entries are added to the RSS object. Since

I want to extract rows from the end of the table, I count the number of rows in the table and use this number to generate an offset (lines 20–23) for the first row that I want to return. The `LIMIT` portion of my second query (line 24) uses that offset and returns rows between that number (`$offset`) and the end of the table (`-1`). Depending on which back-end RDBMS you are using and what your schema looks like, you might need to perform some different steps to achieve this effect.

Finally, the `while` loop (line 27) iterates through each row of data returned, and calls the `XML::RSS add_item()` method using the title and link that was returned from my database.

Generating the File

Now I can call the `as_string()` or `save()` functions to output the data to standard out or to a file. For example:

```
1 print $rss->as_string;
2 save(" /var/www/azurance.rss");
```

Calling either `as_string()` or `save()` results in the output shown in Listing 4.

Checking Your Work

After you're done creating a feed, you might want to check whether it complies with current RSS standards. Mark Pilgrim and Sam Ruby have made available an RSS validator (<http://feeds.archive.org/validator/check>). Quite an invaluable tool, the validator allows you to enter the URL of an RSS file to be checked for errors.

TPJ

Listing 1

```
<channel rdf:about="http://www.azurance.com">
  <title>azurance.com</title>
  <link>http://www.azurance.com</link>
  <description>Open Source and Security Consulting</description>
  <dc:language>en-us</dc:language>
  <dc:rights>Copyright &copy; 1999-2002, Azurance.com</dc:rights>
  <dc:publisher>Azurance</dc:publisher>
  <dc:creator>derek@azurance.com</dc:creator>
  <dc:subject>Open Source, Security</dc:subject>
  <syn:updatePeriod>hourly</syn:updatePeriod>
  <syn:updateFrequency>4</syn:updateFrequency>
  <syn:updateBase>1999-11-05T09:00:00-05:00</syn:updateBase>
  <items>
    <rdf:Seq>
      <rdf:li rdf:resource="http://www.theregister.co.uk/content/55/27734.html" />
      <rdf:li rdf:resource="http://www.vnunet.com/News/1136204"/>
    </rdf:Seq>
    rdf:resource="http://www.internetnews.com/infra/article.php/1486121"/>
  </rdf:Seq>
</items>
</channel>
```

Listing 2

```
1 $rss->add_item(
2
3   title => "Baltimore launches Trusted Business apps",
4   link  => "http://www.theregister.co.uk/content/55/27734.html"
5 );
6
7 $rss->add_item(
8
9   title => "FBI investigates major web slowdown",
10  link  => "http://www.vnunet.com/News/1136204"
11 );
12
13 $rss->add_item(
14
15  title=> "Cisco Boosts Security, Caters To Small Business",
16  link => "http://www.internetnews.com/infra/article.php/1486121"
17 );
```

Listing 3

```
1 sub rss_items {
2
3   use DBI;
4
5   my $itemCount = shift @_;
6   my ($dsn, $dbh, $sth, $rv, @row);
7
8   my $driver      = "mysql";
9   my $database    = "rss_news";
10  my $hostname    = "localhost";
11  my $port        = "3306";
12  my $user        = "username";
13  my $pw          = "password";
14  my $table       = "news";
15
16  $dsn = "DBI:$driver:database=$database:host=$hostname:port=$port";
17  $dbh = DBI->connect($dsn, $user, $pw);
18  $dbh->{PrintError} = 1; # turn off errors, we'll deal with it
                           # ourselves
19
20  $sth = $dbh->prepare("SELECT COUNT(*) FROM news");
21  $rv = $sth->execute;
22  @count = $sth->fetchrow_array;
23  $offset = $count[0] - $itemCount;
24  $sth = $dbh->prepare("SELECT title, link FROM news
                       LIMIT $offset, - 1");
25  $rv = $sth->execute;
26
27  while (@row = $sth->fetchrow_array) {
28
29      my ($title, $link) = @row;
30      $rss->add_item(
31
32          title => "$title",
33          link  => "$link"
34      );
35  }
36 }
37 }
```

Listing 4

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://purl.org/rss/1.0/"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:taxo="http://purl.org/rss/1.0/modules/taxonomy/"
  xmlns:syn="http://purl.org/rss/1.0/modules/syndication/"
>
```

```
<channel rdf:about="http://www.azurance.com">
  <title>azurance.com</title>
  <link>http://www.azurance.com</link>
  <description>Open Source and Security Consulting</description>
  <dc:language>en-us</dc:language>
  <dc:rights>Copyright &copy; 1999-2002, Azurance.com</dc:rights>
  <dc:publisher>Azurance</dc:publisher>
  <dc:creator>derek@azurance.com</dc:creator>
  <dc:subject>Open Source, Security</dc:subject>
  <syn:updatePeriod>hourly</syn:updatePeriod>
  <syn:updateFrequency>4</syn:updateFrequency>
  <syn:updateBase>1999-11-05T09:00:00-05:00</syn:updateBase>
  <items>
    <rdf:Seq>
      <rdf:li rdf:resource="http://www.infoworld.com/articles/hn/xml/02/10/24/021024hnnpcwest.xml?s=IDGNS" />
      <rdf:li rdf:resource="http://www.idg.net/ic_959380_1794_9-10000.html" />
      <rdf:li rdf:resource="http://www.infoworld.com/articles/hn/xml/02/10/25/021025hnsecurelinux.xml?s=IDGNS" />
      <rdf:li rdf:resource="http://www.cnn.com/2002/TECH/internet/10/23/net.attack/index.html" />
      <rdf:li rdf:resource="http://www.infoworld.com/articles/hn/xml/02/10/23/021023hnapteron.xml?s=IDGNS" />
      <rdf:li rdf:resource="http://www.businessweek.com/technology/cnet/stories/963054.htm" />
      <rdf:li rdf:resource="http://www.itweb.co.za/sections/internet/2002/0210240947.asp?A=HOME&O=FPIN" />
      <rdf:li rdf:resource="http://www.internetwk.com/security02/INW20021023S0001" />
      <rdf:li rdf:resource="http://www.pcw.co.uk/News/1136211" />
      <rdf:li rdf:resource="http://zdnet.com.com/2100-1105-963087.html" />
    </rdf:Seq>
  </items>
</channel>
```

```
<item rdf:about="http://www.infoworld.com/articles/hn/xml/02/10/24/021024hnnpcwest.xml?s=IDGNS">
  <title>Network chip makers focus on security</title>
  <link>http://www.infoworld.com/articles/hn/xml/02/10/24/021024hnnpcwest.xml?s=IDGNS</link>
</item>
```

```
<item rdf:about="http://www.idg.net/ic_959380_1794_9-10000.html">
  <title>'The Golden Age of Hacking rolls on'</title>
  <link>http://www.idg.net/ic_959380_1794_9-10000.html</link>
</item>
```

```
<item rdf:about="http://www.infoworld.com/articles/hn/xml/02/10/25/021025hnsecurelinux.xml?s=IDGNS">
  <title>Secure Linux maker teams with IBM in U.S.</title>
  <link>http://www.infoworld.com/articles/hn/xml/02/10/25/021025hnsecurelinux.xml?s=IDGNS</link>
</item>
```

```
<item rdf:about="http://www.cnn.com/2002/TECH/internet/10/23/net.attack/index.html">
  <title>FBI seeks to trace massive Net attack</title>
  <link>http://www.cnn.com/2002/TECH/internet/10/23/net.attack/index.html</link>
</item>
```

```
<item rdf:about="http://www.infoworld.com/articles/hn/xml/02/10/23/021023hnapteron.xml?s=IDGNS">
  <title>RSA, AMD team up on security for Opteron chips</title>
  <link>http://www.infoworld.com/articles/hn/xml/02/10/23/021023hnapteron.xml?s=IDGNS</link>
</item>
```

```
<item rdf:about="http://www.businessweek.com/technology/cnet/stories/963054.htm">
  <title>Encryption method getting the picture</title>
  <link>http://www.businessweek.com/technology/cnet/stories/963054.htm</link>
</item>
```

```
<item rdf:about="http://www.itweb.co.za/sections/internet/2002/0210240947.asp?A=HOME&O=FPIN">
  <title>Internet banking security revolutionised with SMS-based cross-checking</title>
  <link>http://www.itweb.co.za/sections/internet/2002/0210240947.asp?A=HOME&O=FPIN</link>
</item>
```

```
<item rdf:about="http://www.internetwk.com/security02/INW20021023S0001">
  <title>Vendor Warns Of New IE Holes; Microsoft Calls Reports Irresponsible</title>
  <link>http://www.internetwk.com/security02/INW20021023S0001</link>
</item>
```

```
<item rdf:about="http://www.pcw.co.uk/News/1136211">
  <title>PGP poised for major comeback</title>
  <link>http://www.pcw.co.uk/News/1136211</link>
</item>
```

```
<item rdf:about="http://zdnet.com.com/2100-1105-963087.html">
  <title>P2P hacking bill may be rewritten</title>
  <link>http://zdnet.com.com/2100-1105-963087.html</link>
</item>
```

```
</rdf:RDF>
```



**HURRICANE ELECTRIC
INTERNET SERVICES**

Full Cabinet Colocation

Holds up to 42 1U servers

\$400/month

Colocate Your Server

Up to 7U or 12 inches with 256kpbs~70Gb transfer

\$200/month

Order Today!

email sales@he.net
or

call **510.580.4190**

TPJ

Making a Cross-Platform Installer with Perl

I recently wrote a multimodule, cross-platform, Perl-based installer. I encountered many unexpected hurdles while creating this piece of a larger software package, including problems with third-party module dependencies, unexpected platform-specific Perl behavior, CPAN and PPM autoinstallation complexity, and even platform-specific module bugs. In this article, I will discuss these issues and show how my coworker and I overcame them.

The installer is part of a midsized, CGI-based package. We had just five weeks to deliver the software, and this install tool was one of the deliverables. I naively told our customer that I could easily create a script that would automate the configuration and code installation process and also install any third party modules needed by our package that were not present on the target system.

I did my best to choose modules for this project that appeared to be both stable and available on both the CPAN and Activestate PPM repositories. Coding the system went very quickly and I felt confident that the delivery would be on time and relatively stable. After the coding was complete, I began writing the installer.

I started with the code that would install any third-party modules onto the target system. I felt this would be the most interesting, fun, and easy part of the installer. I soon discovered otherwise.

Dealing with Dependencies

My troubles began with the XML modules I chose. The project needed a fast and easy way to select elements or subsets of a complete XML document. *XML::XPath* came to mind immediately. I had used XPath in conjunction with Cocoon, in Java, at my last job. This module depends on *XML::Parser* (almost all XML namespace CPAN modules depend on *XML::Parser* directly or indirectly) and *XML::RegExp*. I also chose *XML::EasyOBJ* for programmatically writing XML documents, as it is very easy to use and has a clean syntax. *XML::EasyOBJ* depends on *XML::DOM*, which depends on *XML::Parser*. *XML::Parser*, for those of you

who haven't used it, is a glue module for the very fast (in my experience), very functional C-based *expat* library.

I encountered no problems in Windows with the *XML::Parser* installation. The PPD (ActiveState package file) for *XML::XPath* comes with a precompiled *expat* DLL. On *nix-based systems, no binary distributions were available that I could find, so I had to do the following to autoinstall the module and the *expat* library:

1. Hard-code the latest stable *expat* distribution in our distribution to avoid writing code to download it dynamically.
2. Write a shell script to explode the distribution, call the included configure script to ready the source for compilation, then call *make* to make and install it.
3. Hard-code the latest *XML::Parser* distribution in the source-code tree, and add code to the shell script to build and install it.

The script I used to do the install is shown in Listing 1.

I used the Bourne shell for this because it was quicker than writing it in Perl, and the Bourne shell is available on all flavors of UNIX/Linux. I called this script from the *setup.pl* script using the *system()* function (see Listing 2).

XML::EasyOBJ seemed to be a no-brainer; both the CPAN and Activestate repositories had it listed for download. I was wrong. I tried to install the module using ppm and discovered that the PPD file for this package was not installable. The quick workaround for this was to hard install the module in our distribution. So much for autoinstalling all third-party modules.

The system we wrote also depends on *Crypt::SSLeay* and LWP for posting XML to two gateways over HTTP using secure sockets (SSL). *Crypt::SSLeay* depends on the OpenSSL C library. PPM includes the OpenSSL DLL with the *Crypt::SSLeay* PPD, but CPAN does not. Our solution was to write another shell script to drive the install after adding the latest *openssl* and *Crypt::SSLeay* distributions in our package.

Problem solved? Not quite. The *Crypt::SSLeay* Makefile.pl looked for OpenSSL in directories usually only writable by root. We had to be able to run in a shared hosting environment with potentially very limited permissions. Our solution was to edit Makefile.pl so that it would find our private, nonroot OpenSSL

Max has worked as a developer and systems administrator since 1995, and has run a web hosting business (<http://webwizarddesign.com/>) for the last three years. He is currently working as a consultant with Vanward Technologies. He can be reached at max@vanwardtechnologies.com.

install and build against that. The shell script to do the install is shown in Listing 3.

Here is the altered portion of *Crypt::SSLeay* Makefile.pl that lets it find our local install of OpenSSL:

```
if (exists $ENV{'OPENSSL_DIR'}) {
    unshift(@POSSIBLE_SSL_DIRS, $ENV{'OPENSSL_DIR'});
}
```

For session management, I chose the *CGI::Session* module with the file-based backend for persistence, as I felt that would be the most portable and easiest to set up of the available *CGI::Session* backends. *CGI::Session* is a wonderful module—I really enjoyed integrating it into this system. However, on Windows, calls to clear session objects of all data or delete data from a session would mysteriously hang. My first thought was that I had encountered a file-locking issue since I was developing this under cygwin on Windows. I reviewed the *CGI::Session::File* source, removed the file-locking code, and tried again. No luck. I then went to the *CGI::Session* online mailing list archive and found the issue addressed.

CGI::Session uses *Autoloader* and has two sets of clear and delete routines, the names of which only differ in case: *CLEAR()/clear()* and *DELETE()/delete()*. Under Windows, one of the routines gets overwritten due to the case-insensitive nature of the platform; this causes calls to the like-named methods to fail. To fix this, we moved the lowercase named pair above the *__END__* tag in the *Session.pm* module so *Autoloader* never bothers with it. The result was that we had to hard install yet another module into our software package (this issue has since been fixed in the *CGI::Session* distribution).

It was my understanding that any module accepted into CPAN had to detect and execute CPAN installation for any missing dependent modules. It turns out this isn't entirely true. The XML modules I picked did not autoinstall all of their needed dependencies, and I found that if *Crypt::SSLeay* and *XML::Parser* were not on the system, I had to install them first so any dependent modules would find them. I remembered that *Bundle::LWP* solved my dependency tree problem for *LWP* on *nix, but I then discovered that the bundle doesn't exist for Windows and isn't needed, as *LWP* is installed as a standard module with ActiveState Perl. The list of modules needed by the software this script installs is stored as a hash table in the script. It looks like this:

```
my %modules = qw(
    CGI                2.49
    Storable           2.00
    CGI::Session       2.94
    DB_File            1.73
    Crypt::CBC         2.00
    Crypt::SSLeay      0.17
    Crypt::Blowfish_PP 1.10
    HTML::Template     2.4
    Bundle::LWP        1.6
    LWP                5.44
    Test::More         0.47
    XML::Parser        2.27
    XML::RegExp        0.03
    XML::DOM           1.06
    XML::XPath         1.04
    XML::EasyOBJ       1.12
);
```

I thought it would then be easy to just edit the list in the script and let the generic installer code install every module. Due to the *XML::Parser* and *Crypt::SSLeay* dependency issues, I had to add this code deep in the install routines themselves to make sure

XML::Parser/expat and *openssl/Crypt::SSLeay* were installed before any other needed modules:

```
if (grep(/XML::Parser/, @need)) {
    @need = grep(!/XML::Parser/, @need);
    install_xml_parser($perl, $lib);
}

if (grep(/Crypt::SSLeay/, @need)) {
    @need = grep(!/Crypt::SSLeay/, @need);
    install_crypt_sslay($perl, $lib);
}
```

@need is a list of modules the target system does not have, and is populated using the code below. Notice the code that deletes the *LWP* package is not needed for a given platform and also notice I have to use *use lib* in the system call so that Perl will be able to see the modules we had to hard install into the software distribution; the Perl binary the user picks for the CGI scripts may not be the same one that is running the setup script. Finally, the variable *\$lib* holds the absolute pathname to the lib directory in the install tree and is set earlier in the script, as shown in Listing 4.

Up to this point, I had been testing the install process only under cygwin with CPAN. The first time I ran the script under a DOS prompt on Windows NT, ppm2 complained that some of the PPDs I tried to install were corrupt, and “force install” failed for some reason when called through the install script. I debugged my own ppm2 automation code and searched for information on PPM and answers online for several hours. No luck. I then searched the Windows Perl installation tree, hoping to find something to help me out. I found a ppm3 batch file. I hadn't been aware of an upgrade. (Of course, the install notes talked about it in depth, but

Perl2Exe™



Convert your Perl scripts to stand alone exe files.

Ship your exe file without having to ship your Perl source code or install Perl.

Supports Windows and Unix.

Free non-expiring trial version.



www.indigostar.com

unfortunately, I had skipped them.) This worked, but I was in a huge time crunch, and the method of setting the root directory for installs had changed between ppm2 and ppm3. I was having a hard time figuring out how to set this option with ppm3, partially because I was anxious about the looming deadline. I recently revisited this, and it turns out to be easy to do.

Setting the root under ppm2:

```
C:> ppm set root <directory name>
Root is now <directory name> [Was <previous
  directory>]

open(S,"$perl -S ppm.bat set root $dir |") || die
  "Can't read from PPM: $!\n";
my $line = <S>;
close(S);
$orig_root = ($line =~ m/[was\s+(.+)\/i][0];
```

Under ppm3:

```
ppm3 target set root <directory name>
```

If I hadn't been in a rush, I would have used ppm3 to do nonadministrator installs; since delivery day was very near and I couldn't afford more time to learn ppm3, I decided instead to just document that the person installing this package under Windows would have to have administrator privileges also or have an administrator install it for them. I plan to change this in the next release.

Cross-Platform Woes

For *nix environments, I stayed with doing either a local (non-root) install or a root install based on the UID of the person running the script. QA started testing this part of the installer in different *nix environments. The first error they uncovered involved installing the package as a nonroot user on a machine that already has a systemwide CPAN Config.pm, but no local CPAN configuration for the user running the install. The install failed because the installer couldn't write to system directories such as /etc. Our fix was to set the environment variable *PERL_MM_OPT* to *LIB=<path to distribution lib directory>* and start CPAN initialization for the nonroot user. This is shown in Listing 5.

QA then found another case: a root user who has *su'd* without "*su -*" so that their username variable still points to their real user ID. This would cause the CPAN installer to try and use their local CPAN configuration over the systemwide configuration. To fix this, we manually set the username variable in the script to "root" and set their effective UID to be the same as their real UID in the script:

```
$ENV{USER} = 'root';
$< = $>;
```

On Solaris, the bootstrap shell script that calls setup.pl failed because the Bourne shell on Solaris does not have a built-in / operator; instead it calls the shell utility *test*, which is symlinked to the character /. I had left the quotes out of the right-hand side of a conditional I used to determine if the script was using a BSD or SysV flavor of *echo*. This caused the conditional to be malformed when it was passed to *test* on Solaris, which in turn caused the bootstrap script to abort. This was an easy fix—I just quoted the right-hand side of the expression. Here is the fixed code I used to detect which *echo* was in use in the bootstrap script.

```
# To get SysV/BSD echo sans-newline right

vnl=
```

```
bnl="\c"

if [ "'echo \"\c\"'" = "\c" ]
then
    vnl=-n
    bnl=

fi

echonl() {
    echo $vnl "$*$bnl"
}
```

QA then moved on to test the install under Windows. They found that when my setup script tried to find all existing Perl binaries on a Windows 2000 system (the second step in the install process), the install died with an error about a unicode module not being found. This was puzzling, as we were not doing anything with Unicode in this release. I discovered that when I used *glob()* on Windows 2000 to get file names, I would get a list of long file names. When I used a long Windows-style file name in a regular expression, Perl would try to interpret some backslash followed by a letter sequences, such as

```
\l
```

as unicode characters. It would find that it did not have the unicode decoder loaded for that character, and die. We wrote our own mini *glob()* that handles this case correctly by quoting metacharacters in the regular expression matcher:

```
sub our_glob {
    my $dir = shift;
    my $pat = shift;
    local(*D);
    local($_);
    opendir(D,$dir);
    my @files = grep(/\Q$pat/, readdir(D));
    return map {File::Spec->catfile($dir, $_);}
        @files;
};
```

With the install then working correctly on both *nix and Windows, QA discovered that even after the install succeeded, the CGIs would fail to find the modules we installed. I forgot to include system-dependent library directories, for example 'i386-linux', in my *use lib* statements in all the CGIs for the modules the script installs locally. The question was how to do this without knowing the system type ahead of time (scramble for *Perl in a Nutshell*). A combination of FindBin.pm and Config.pm does that for us:

```
use Config;
use FindBin;

use lib "$FindBin::Bin/lib";
use lib "$FindBin::Bin/lib/$Config{archname}"
```

Conclusion

I haven't related all of the problems I encountered in creating this script, which also has six additional installation steps, each with character-based screens and a fair amount of user interaction. The problems I encountered haven't dampened my enthusiasm for Perl, and I think CPAN and PPM are wonderful repositories. But I also see that one of the downsides of having such an open, do-it-how-you-want-to system is that it allows many implementation differences among modules and even Perls.

Writing and debugging the installer took approximately 30 percent of the time allocated to this project. I encountered problems with autoinstalling modules, issues with bugs in the third-party modules I used, issues with my own lack of awareness of Windows long path names, and regular expression interaction. I also failed to imagine the full variety of possible installation environments in which this script would be run, given that it is supposed

to run “anywhere.” I hope I have helped others avoid some of the same mistakes.

The online documentation for this project (which was also part of the five-week project), is viewable at <http://vrpp.support.netsol.com/docs/>.

TPJ

Listing 1

```
#!/bin/sh

PATH=$PATH:/usr/local/bin:/usr/local/gnu/bin:/usr/gnu/bin:
/opt/bin:/opt/gnu/bin
PATH=$PATH:/usr/bin:/bin:/usr/ccs/bin
export PATH

CC=gcc
export CC

PERL=$1
DIR=`pwd`

gzip -d -c ./expat-1.95.5.tar.gz | tar xvf -
cd ./expat-1.95.5
./configure --prefix=$DIR
make
make install
cd ..
rm -rf ./expat-1.95.5

gzip -d -c ./XML-Parser-2.31.tar.gz | tar xvf -
cd ./XML-Parser-2.31/
$PERL Makefile.PL EXPATLIBPATH="$DIR/lib" EXPATINCPTH=
"$DIR/include" CC=gcc LD=gcc
make
make install
cd ..
rm -rf ./XML-Parser-2.31
```

Listing 2

```
sub install_xml_parser {
    my $perl = shift;
    my $dir = shift;

    Screen::clear();
    print "Installing expat library and XML::Parser .. \n";
    Screen::pause();

    if (exists $ENV{'LD_LIBRARY_PATH'}) {
        $ENV{'LD_LIBRARY_PATH'} .= ":$dir";
    } else {
        $ENV{'LD_LIBRARY_PATH'} = "$dir";
    }

    system("cd $dir; sh ./make_expert $perl");

    print "XML::Parser installed.\n";
    Screen::pause();
}
```

Listing 3

```
#!/bin/sh

PATH=$PATH:/usr/local/bin:/usr/local/gnu/bin:/usr/gnu/bin:
/opt/bin:/opt/gnu/bin
PATH=$PATH:/usr/bin:/bin:/usr/ccs/bin
export PATH

CC=gcc
export CC

PERL=$1
DIR=`pwd`

gzip -d -c ./openssl-0.9.6g.tar.gz | tar xvf -
cd ./openssl-0.9.6g
./config --prefix=$DIR
make
make install
cd ..
rm -rf ./openssl-0.9.6g

gzip -d -c ./Crypt-SSLeay-0.45.tar.gz | tar xvf -
```

```
cd ./Crypt-SSLeay-0.45/

OPENSSL_DIR=$DIR
export OPENSSL_DIR

$PERL Makefile.PL CC=gcc LD=gcc
make
make install
cd ..

rm -rf ./Crypt-SSLeay-0.45

exit 0
```

Listing 4

```
my $perl = $pool->get('PERL');

# Close standard error so user doesn't see failure messages for modules
# that don't exist.

close(STDERR);

my $cmd;

# On windows, no such thing as Bundle::LWP, on UNIX/Linux,
# just want to test for Bundle so dependencies are done correctly.

if (Util::iswin()) {
    delete $modules{'Bundle::LWP'};
} else {
    delete $modules{'LWP'};
}

for (sort keys %modules) {
    $cmd = "$perl -e \"use lib q!$lib!; use $_ $modules{$_};\"";
    if (system($cmd)) {
        push(@need, $_);
    } else {
        push(@have, $_);
    }
}

open(STDERR, '>&0');
```

Listing 5

```
# For local, non-root installs
$ENV{'PERL_MM_OPT'} = " LIB=$dir" if defined $dir;

my $cpanrc = "$ENV{'HOME'}/.cpan/CPAN/MyConfig.pm";
unless (-f $cpanrc) {
    initialize_cpan($cpanrc, $perl);
}

sub initialize_cpan {
    my $cpanrc = shift;
    my $perl = shift;

    use File::Basename;

    Screen::clear();
    print "Initializing CPAN (follow on-screen instructions:\n";
    Screen::pause();

    my $dir = File::Basename::dirname($cpanrc);
    DirCopy::make_path($dir, 0700);

    system("$perl -MCPAN::FirstTime -e
'CPAN::FirstTime::init(q!$cpanrc!)'");

    print "Initialization complete! On to module install.\n";
    Screen::pause();
}
```

TPJ

Perl & Rapid Database Prototyping

Even though the bulk of my production development is in C, Java, and assembly language, I often turn to Perl to quickly prototype new ideas. That's especially true with database applications. Perl's DBI module for database access provides convenient access to the most popular relational databases, and its strong text-processing abilities make it easy to synthesize complex queries.

Over the last several years, I've refined a simple framework that handles most routine database access automatically and significantly accelerates the development of simple database applications. This framework takes advantage of Perl's advanced object-oriented features to pack a lot of power into a small package.

Accessors

Accessor functions are the easiest way to abstract database access. By convention, accessors in Perl are subroutines that can either set or return a value, depending on how they are called. Listing 1 illustrates a simple accessor function. The accessor always returns the old value; if a new value is specified, the function also sets the value.

The Class-Per-Table Pattern

Accessors are a natural way to manipulate database data. A simple object-oriented design pattern uses a single class for each table in your database. An object of such a class represents a single row, and you use accessor methods to read or change individual fields. Such a system could be used as in Listing 2. The details of the data storage are completely hidden; this design makes it easy to change the details of the low-level storage without any impact on the higher level application code.

Exploiting AUTOLOAD

The class-per-table pattern can involve a lot of highly repetitive code. For each column in the database, you need a method that can read/write that column. Often, programmers build simple code generators to create reams of code for such classes.

Perl's *AUTOLOAD* capability lets you eliminate almost all of the explicit accessor methods. If there is no "phone" method

defined, then an expression such as *\$e->phone* invokes the *AUTOLOAD* method instead. The special *\$AUTOLOAD* variable contains the full name (including package name) of the method that was requested. Using this, the *AUTOLOAD* method can load the requested method from a disk or database or simply emulate the call directly.

My database access framework uses *AUTOLOAD* to provide a default accessor method for any column of a database. I simply use the name of the column as the name of the accessor. For example, if there is no explicit implementation of the phone method, *\$e->phone('555-5555')* invokes *AUTOLOAD* to store the specified value into the phone column. This gives you convenient access to any database column without having to write any code.

Packaging It Up

Since real-world projects have lots of tables, I've created a single *DBTable* class that contains the *AUTOLOAD* method just described and a number of other convenient methods. The per-table

classes simply inherit most of their functionality from *DBTable*. You only need to write one or more constructors that let you access rows in your tables.

To illustrate, consider the schema in Listing 3. Listing 4 is the complete Perl code needed to use this schema. In particular, note that I've defined no accessors; the *AUTOLOAD* method in *DBTable* will handle them automatically.

Defining New Tables

The first three lines of Listing 4 are just standard boilerplate for any Perl class. The only interesting part is the *byName* constructor. The *_fetchOrCreate* method accepts a hash that identifies this row and returns a hash with the values of one or more columns. If the row doesn't exist, it is created.

The *DBTable* methods rely on the *TABLE* and *KEY* entries in the hash. The *TABLE* entry specifies the name of the database table. The *KEY* entry is a hash providing a primary key that identifies the particular row. In Listing 3, the primary key is the value of the *employee_id* field. The first argument to the *_fetchOrCreate* method specifies a condition for identifying or creating the necessary row; the second argument is the column whose value is to be returned.

*This framework has helped
me to very rapidly build
new database applications*

Tim is a freelance software developer and consultant. He can be reached at kientzle@acm.org.

If you now look carefully at Listing 2, you can see how the entire system works. The `$e` variable in that example contains everything needed to identify a particular row in a certain table. The call `$e->phone('555-5555')` invokes the `AUTOLOAD` method in the `DBTable` class, which can use the table name (from the `TABLE` key), the primary key (from the `KEY` key), and the column name (from the method name) to access the database.

Inside DBTable

The full implementation of `DBTable` is shown in Listings 8 and 9, available in the Source Code Appendix, pages 24–26. There are only four places where SQL statements are actually constructed: `_fetch` reads one or more columns from a row, `_create` creates a new row, `_set` sets the values of columns, and `_columns` obtains the names of the columns in this table. The string manipulations that convert hashes into SQL statements are admittedly a bit convoluted, but they are hidden inside just a couple of small functions. Also available electronically is an example program that uses this code. You can use this as a starting point for your own database applications.

Advanced Techniques

Sometimes you will want accessors that return objects rather than raw database fields. Listing 5 is a *manager* method that returns an object. With this addition, you can refer to `$e->manager_id` to obtain the numerical ID for the *manager*, or `$e->manager` to obtain an object. Such methods are especially useful when you have complex schemas with many interrelated tables. In like fashion, you can define methods that simulate nonexistent database fields (by transparently accessing joined tables, for instance).

The *byID* constructor in Listing 5 could have simply used `$self->{'KEY'} = {'employee_id'=>$id}` to set the *KEY* without any database access. Although that would be faster, it doesn't provide any guarantees that the requested row actually exists.

Finally, remember that `$e->dbHandle` returns the underlying DBI database handle to your application so that you can use SQL statements directly if necessary.

Performance

This framework makes it easy to prototype database applications. You simply define a package for each table in your schema, and those packages need only contain one or two simple constructors each. You can then begin to build your application, accessing database columns through accessor functions.

Of course, such applications are going to be slow. Every time you read or write a field, at least one database access is going to occur. As your application evolves, you can improve performance by overriding specific accessors. For example, if you access the primary key fields frequently, you can override them using code like that in Listing 6.

Listing 7 takes this idea a step further by modifying the constructor to store the entire row in memory, and overrides the `_fetch` and `_set` methods to read and update the in-memory versions, dramatically reducing database traffic. Such optimizations are not always appropriate, however, and are generally best delayed until you understand your application better.

Future Directions

Currently, `DBTable` stores the DBI database handle in a global variable, which limits you to a single database handle at a time. Applications that must access multiple databases will need to find a way to address this.

Right now, the `_columns` method is rather wasteful. It generates a full database query once for every object. It would be more efficient to do so only once for every table. Worse, it stores a list of valid columns in every object rather than once for each table, which could become a significant memory problem.

Never Write A Makefile Again Let This New Software Tool Write Them For You

Chop your makefile errors, your makefile coding costs, and the size of your build team to zero.

Stop fixing broken makefiles. Stop worrying about whether or not your new makefile code is going to work. *Stop thinking about makefiles period.* Generate smart makefiles instead.

NEW TECHNOLOGY

Four new patent-pending technologies make it possible. *Collections* are new, universale software containers formed by adding collection specifier files to trees of source files. A *Collection Knowledge Base* holds 12 years of makefile generation experience for Fortran, C/C++, Perl, Python, and Java, for 20 Unix and Windows platforms. A *Smart Makefile Generator* emits fast, parallel makefiles that follow your site conventions every time. An *Adaptive Focus GUI* dynamically adapts itself to match collection data types.

You get a complete and multiplatform *software build infrastructure* for fast parallel builds, profiling, debugging, memory checking, fast re-linking, regression testing, file sharing, code linting and formatting, release staging, software installations, and more.

A makefile for "hello world" has 133 targets—3 to build the program, and 130 for other software

processes. You get a *software reuse infrastructure* to manage multiplatform software fragments, and a *checklist infrastructure* to help guide you through complex procedures.

SMART KNOWLEDGE BASE

You get more than *1000 template files* in the customizable knowledge base. Use *makefile services* to encapsulate, name and reuse your process fragments. Use *GUI focus variables* to pass state information into *parameterized GUI actions*.

Use *knowledge plugins* to encapsulate and share knowledge with your peers. Write your own plugins, or extend your database with expert *third-party plugins* for J2EE and database development.

Use *knowledge contexts* for complex process variations that require different software tools, process steps, and command line arguments. Contexts are named sets of *knowledge search rules*.

Use *knowledge trees* to organize and share knowledge at your site, or store it in knowledge collections for sharing with peers at other sites. Use *knowledge debugging tools* to inspect the knowledge stored in your database.

INFRASTRUCTURE MANAGERS

Smart Makefiles Now is the first of five new tools from a *12 year, multi-million*

dollar project to build smart, knowledge-based software tools. It captures your site process knowledge and carries it forward to future projects where you can reuse it automatically at zero labor cost.

It's a general tool for people that work with automated software processes, including developers, sysadmins, testers, and scientists, who all run programs, scripts and data files on multiple platforms.

It's a "glue" framework for all the various scripts, tools, files, code bits, notes and checklists at your site. Organize and implement your site policies and tools under a single, generic, adaptive GUI interface.

It scales up easily over people, projects, platforms, languages, policies, and preferences. It reduces project risk because process knowledge is stored in the database, not in people, and because generated makefiles are not sensitive to funding cuts, reorgs, and staff turnover.

Work at higher abstraction levels with collection objects. Apply operations to whole collections automatically, instead of applying hand-crafted makefile command sequences to individual files.

Improve your software processes by changing the database. One person adds a new best practice and everyone else benefits at zero additional labor costs. Incremental effort inevitably extends the power of your database.

Smart Makefiles Now has been tested on *millions of multiplatform makefiles*. At our site, it generates 6000 makefiles for 20 platforms to build 1 MLOC every day—2 million makefiles a year. It chops costs down in the *whole second half of the software lifecycle*.

AFFORDABLE PRICE

Street price is *\$0.01/hr*—two thousand times cheaper than programmers at \$20.00/hr, and more efficient. You get all platforms with each loginid license.

See for yourself. Spend an hour with the world's first smart makefile generator. *Learn new concepts.* Run the demo collections and try out some plugins. Then copy your own files into the demo collections and generate some smart makefiles for your own projects

It's simple because it's *smart*. Get yours now.

www.as2.smart-make.com

Conclusion

This framework has helped me to very rapidly build new database applications. In some cases, performance is not a serious issue, and such code can even be used in production. However, even when performance is an issue, this design is still a good place to start. You should generally be able to build functioning applica-

tions using this interface and then upgrade the underlying data access to improve performance without having to extensively modify your application.

TPJ

Listing 1

```
# Simple accessor
sub foo {
    my $a=$FOO;
    ($FOO)=@_ if @_;
    return $a;
}
# Using accessors
foo(4); # foo = 4
print foo(5); # Print 4, set foo=5
print foo(); # Print 5
```

Listing 2

```
# Access a row
$e = Employee->byName('Doe', 'John');
# Get phone number
print $e->phone();
# Set phone number
$e->phone('555-5555');
```

Listing 3

```
CREATE TABLE employee(
    employee_id INT PRIMARY KEY,
    first      CHAR(80),
    last       CHAR(80),
    phone      CHAR(80),
    manager_id INT
);
```

Listing 4

**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.

```
package Employee
use vars qw(@ISA);
@ISA = qw(DBTable);
sub byName {
    my($class,$last,$first) = @_;
    my $self = bless {'TABLE'=>'employee'},$class;
    # _fetchOrCreate returns a hash
    # { 'employee_id' => <number> }
    $self->{'KEY'} = $self->_fetchOrCreate(
        {'last' => $last,'first' => $first}, 'employee_id' );
    return $self;
}
```

Listing 5

```
# An accessor returning an object
sub manager {
    my($self) = @_;
    return Employee->byId($self->manager_id);
}
# Another constructor
sub byID {
    my($class,$id) = @_;
    my $self = bless {},$class;
    $self->{'TABLE'} = 'employee';
    $self->{'KEY'} = $self->_fetchOrCreate(
        ({'employee_id' => $id}, 'employee_id' );
    return $self;
}
```

Listing 6

```
# An optimized employee_id accessor
sub employee_id {
    my $self = shift;
    $self->{'KEY'}->{'employee_id'};
}
```

Listing 7

```
# An optimized Employee class
package Employee
use vars qw(@ISA);
@ISA = qw(DBTable);
sub byName {
    my($class,$last,$first) = @_;
    my $self = bless {},$class;
    my $cols = $self->_columns;
    $self->{'TABLE'} = 'employee';
    # Read and cache all of the columns
    $self->{'FIELDS'} = $self->_fetchOrCreate(
        ({'last' => lc $last, 'first' => lc $first}, keys %$cols);
    $self->{'KEY'} =
        { 'employee_id' => $self->{'FIELDS'}->{'employee_id' } };
    return $self;
}
# Override _fetch to just pull the value from memory
sub _fetch {
    my ($self, $key, $col) = @_;
    return { $col => $self->{'FIELDS'}->{$col} };
}
# Override _set to modify the in-memory value
sub _set {
    my($self,$key,$set) = @_;
    $self->SUPER::_set($key,$set); # Set fields in DB
    foreach $k (keys %$set) {
        $self->{'FIELDS'}->{$k} = $set->{$k};
        # Remember to update the value in 'KEY', if it's there
        if(exists $self->{'KEY'}->{$k}) {
            $self->{'KEY'}->{$k} = $set->{$k};
        }
    }
}
```

TPJ



Ruby vs. Perl

brian d foy

Lately, I have been playing with Ruby. Occasionally, I take a vacation from Perl to try out a new language. Usually, I cannot wait to get back to Perl. I did not find Python's whitespace rules very appealing, Java is just too much work, and most people have never heard of Smalltalk. I am still playing with Ruby—maybe because it comes with Mac OS 10.2. Should I switch from Perl to Ruby?

Ruby has what most PASCAL users have wanted in Perl—a *writeln* sort of function. The “hello world” program in Ruby is just a little bit shorter than in Perl. I can leave off the trailing new-line in Ruby, which takes away one of my most common programming omissions.

```
#!/usr/bin/perl
print "Hello world!\n"
```

```
#!/usr/bin/ruby
puts "Hello world!"
```

I also like that everything is truly an object, even literals, so that I can call methods on everything. In Perl and some other pseudo object-oriented languages, I first have to get an object to treat something like “0” as an object. In Ruby it already is an object, and I use it this way later in this article. Since Ruby came after Perl and Python, it simplified some things that those languages had to continue to support (although Perl 6 may do even better).

A couple of weeks ago, I started to port my Business::ISBN module, which lets me deal with International Standard Book Numbers (ISBN), to Ruby. The module is fairly simple and does not have that many advanced concepts. It mostly handles simple string processing along with simple arithmetic. It was one of my first real Perl modules, so maybe it should be my first Ruby module, too.

Although Ruby has been around since 1995, it still does not have many modules. One of the reasons that Perl is so useful and always draws me back is that it is easy to get things done because it has so many modules. The Comprehensive Perl Archive Network (CPAN) has about 2500 registered modules, and many more

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. He can be contacted at comdog@panix.com.

unregistered ones—all of which I can easily find on CPAN Search (<http://search.cpan.org/>). The Ruby Application Archive (RAA) (<http://www.ruby-lang.org/en/raa.html>) has only a fraction of that total.

The ISBN is a number that publishers assign to books and similar items. Each binding of a book should get its own ISBN. For instance, *Programming Perl*, the first edition, has the ISBN 0-937-17564-1. The first part of the ISBN, “0” in this example, specifies the language, and the second part the publisher. The third part uniquely identifies the book, and the last part is the checksum. These might show up in data with or without the hyphens, or with spaces or other characters. I created the Business::ISBN module so I could easily go through tens of thousands of ISBNs, verify their checksums, and sort them by publisher.

Part of the Business::ISBN module turns ISBNs into European Article Numbers (EAN), which do the same thing as ISBNs but for more than just books. The EAN prepends three digits to the start of the ISBN and computes a new checksum. In Perl, I use fairly basic coding structure to do the job (see Example 1). In the Perl version, line 5 gets the ISBN as a string with no extra characters and in line 7 returns, unless the ISBN is in the expected format. After that, I go through the EAN algorithm. The *foreach* loop adds up the necessary digits of the ISBN so it can compute the final EAN checksum.

```
1  sub as_ean
2  {
3      my $self = shift;
4
5      my $isbn = ref $self ? $self->as_string([])
6                :_common_format $self;
7
8      return unless ( defined $isbn and length $isbn == 10 );
9
10     my $ean = '978' . substr($isbn, 0, 9);
11
12     my $sum = 0;
13     foreach my $index ( 0, 2, 4, 6, 8, 10 )
14     {
15         $sum += substr($ean, $index, 1);
16         $sum += 3 * substr($ean, $index + 1, 1);
17     }
18
19     $ean .= ( 10 * ( int( $sum / 10 ) + 1 ) - $sum ) % 10;
20
21     return $ean;
22 }
```

Example 1: The *as_ean* subroutine.

```

def to_ean
  ean = '978' + @isbn[0..8]
  sum = 0
  0.step( 10, 2 ) { |n|
    sum += ean[n..n].to_i
    sum += ean[n+1..n+1].to_i * 3
  }
  ean += "#{ ( 10 * ( ( sum / 10 ) + 1 ) - sum ) % 10 }"
  return ean
end

```

Example 2: Ruby version of Example 1.

In Ruby, I can simplify a lot of the EAN checksum code with a specialized iterator (see Example 2). Since everything is an object, I can call methods on anything, including 0. In this case I call the *step()* method on 0. It goes up to 10 in steps of 2. That little bit of code represents the *foreach* loop in the Perl version.

People complain that Perl ranges only go in one direction. In the Perl version of the *Business::ISBN::_checksum* routine (Example 3), I need to multiply the first ISBN digit by 10, the second digit by 9, and so on for the first nine digits. I have one sequence that is ascending, the position in the string, and another descending, the factor. That is not a big problem in Perl since I can use the *reverse()* function to switch around 2..10.

Perl needs to know the whole range to reverse it. It cannot give you the last element as the first element until it knows what that last element is. I could have converted the *foreach()* to a *for()* or a *while()*, but that is not very Perly. Ruby handles this naturally because the *step()* method can go backwards, as in Example 4.

So far Ruby is looking pretty good, at least for the things I have pointed out, but it misses one of Perl's greatest strengths that, until now, I had taken for granted. In the Ruby version of the *_checksum* routine, I have to explicitly handle the number-to-string and string-to-number conversions. In line 6 I have to turn the ISBN digit from a character into an integer so I can use it in the multiplication, and once I have the checksum, in line 11, I have to make sure that it is a string so that the other methods in the module can use it correctly.

Ruby strings are sequences, meaning that I can access parts of the string with a range (for which I would use a *substr()* in Perl). However, if I access a single index, like *@isbn[9]*, I get the ordinal value of the character instead of the character itself. If the tenth character, the checksum, were an "X," a valid ISBN checksum character, *@isbn[9]* returns not the character "X," but its ordinal value 88 (decimal). This must be useful for something, but not anything that I do frequently. It does make it easy to process a string as binary data, though. If I want to access part of a string as a string, I need to use a range, even if the range is only 1. To get the tenth character of *@isbn*, I use *@isbn[9..9]*. Although this is annoying, it is better than Perl's *substr(\$isbn, 9, 1)*.

In the end, I will probably keep my eye on Ruby, and when I have free time, I'll play with it. I might even be able to use small Ruby scripts to perform specific tasks. I cannot stop using Perl though, because it is just too useful. No other language can beat the utility of CPAN. Perl has some rough edges, but it gets the job done faster and just as well as a more purist approach.

References

- EAN International and the Uniform Code Council, <http://www.ean-ucc.org/>
- ISBN.org, <http://www.isbn.org/standards/home/index.asp>
- Business::ISBN, <http://search.cpan.org/author/BDFOY/Business-ISBN-1.70/>

```

sub _checksum
{
  my $data = _common_format shift;

  return unless defined $data;

  my @digits = split //, $data;
  my $sum = 0;

  foreach( reverse 2..10 )
  {
    $sum += $_ * (shift @digits);
  }

  my $checksum = (11 - ($sum % 11))%11;

  $checksum = 'X' if $checksum == 10;

  return $checksum;
}

```

Example 3: The Business::ISBN::_checksum routine.

```

def _checksum
  sum = 0
  10.step( 2, -1 ) {
    |n|
    m = 10 - n
    sum += n * @isbn[m..m].to_i
  }
  checksum = ( 11 - ( sum % 11 ) ) % 11
  checksum = 'X' if checksum == 10
  return checksum.to_s
end

```

Example 4: Ruby version of the _checksum routine shown in Example 3.

Subscribe Now To The Perl Journal

For just \$1/month, you get
the latest in Perl program-
ming techniques in e-zine
format from the world's
best Perl programmers.

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



Mining Mail

Simon Cozens

In my article, “Filtering Mail with Mail::Audit and News::Gateway” (*TPJ*, Summer 2000), I discussed a relatively simple way to help manage e-mail by filtering into mail folders and gatewaying to private news groups. In this article, I’ll discuss the “next generation” of mail handling and mail mining, and demonstrate the utility of a Mail::Miner set up.

Data Mining

The “formal” definition of data mining from the *Free Online Dictionary of Computing* states that it is “analysis of data in a database using tools which look for trends or anomalies without knowledge of the meaning of the data.” However, when we use it in this article, we use it in a much looser sense—data mining is the automated extraction of core pieces of information from a mass of data, and its filing in such a way as to make querying and retrieval relatively easy.

One convenient source of a massive corpus of data suitable for data mining is the mass of e-mail that arrives at our system every day.

E-mail is a surprisingly interesting data format. It contains a lot of structured, regular data in the form of mail headers, which is easy for a computer to parse. Unfortunately, the utility of the mail headers is pretty hit and miss. While things like “To” and “Subject” are always going to be important, in the vast majority of cases, many of the headers are almost useless when the message has been delivered, filed, and read.

The structure of the body is also relatively easy to parse; there may be binary or textual attachments, which may or may not contain useful data. Finally, there’s usually one reasonably large unstructured part, the textual body of the message.

Just like the mail headers, this is pretty hit and miss, too. There could be things that we will want to remember later: phone numbers, dates, place names, addresses, snippets of code, and so on. But interspersed with that, we find line after line of small talk, signature files, flames, and all kinds of other noninformation that is almost useless when the message has been delivered, filed, and read.

The idea behind mail mining is to provide a means for separating the wheat from the chaff. I want to be able to find out where

and when I’m meeting someone, without much concern for the state of the weather in western Japan one particular Friday afternoon. So our goal, then, is to produce a mechanism for extracting useful information from both the structured and unstructured portions of a mail message, preferably without human intervention, and provide a means for retrieving that information quickly and easily.

To put it in extremely human terms, I want a tool that lets me say “Show me the mail I got around three weeks ago from Nat that had something to do with web services and had an interesting snippet of code in it.”

And this is precisely what the Mail::Miner module does.

The Mail::Miner Method

Mail::Miner, as its name implies, is a module, rather than a complete application. To be precise, it’s a collection of modules arranged in the framework shown in Figure 1.

The mail comes in at the top of the diagram and is converted by your Mail::Audit filter into a MIME::Entity, which is handed to Mail::Miner::Message by whatever your delivery process happens to be. Right at that moment, an entry is created for the e-mail in a relational database, storing the from address, subject, and other useful but trivial metadata. Any attachments are stripped off and filed separately in the database, associated with the mail message in question. A notification is added to the body of the e-mail, of the form:

```
[ text/x-perl attachment test.pl detached - use
mm --detach 821
to recover ]
```

Notice the format of this text. To retrieve the attachment, I just cut and paste the middle line onto a shell prompt, and the attachment will be dumped into the current directory. (If there’s already a *test.pl* there, *mm*, the Mail::Miner command-line utility, will prompt before overwriting.)

Then Mail::Miner locates and calls any Mail::Miner recognizers. We’ll come back to those in a second.

After this point, the mail, with its newly flattened body, can be filed into the database. Once that’s done, the message can leave the Mail Miner system, ready for delivery to the user’s inbox.

Notice that here, even with no cleverness, we have a system for managing attachments, plus a searchable database of old e-mail. However, the real power of Mail::Miner comes in its recognizers.

Simon Cozens 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, a former Parrot pumpking, and an obsessive player of the Japanese game of Go. Simon can be reached at simon@simon-cozens.org.

Recognizers and Queries

Recognizers are simply modules that look for things that may be considered interesting in an incoming message (“assets”), file them away for later, and provide an interface to query for them.

Let’s take a tour of the currently implemented Mail::Miner recognizers.

The first recognizer isn’t really a recognizer at all, but it does provide a query mechanism—the Mail::Miner::Message module itself can be used to query the From address of messages in the database. Recognizers declare command-line options that they can fulfill, so I can now say:

```
% mm --from tpj.com --summary
26 matched
766:2002-03-29:      Kevin Carlson
<kcarlson@tpj.com>: The Perl Journal
768:2002-03-29:      Kevin Carlson
<kcarlson@tpj.com>:Re: The Perl Journal
769:2002-03-29:      Kevin Carlson
<kcarlson@tpj.com>:Re: The Perl Journal
770:2002-03-29:      Kevin Carlson
<kcarlson@tpj.com>:Re: The Perl Journal
783:2002-04-03:      Kevin Carlson
<kcarlson@tpj.com>:Re: The Perl Journal
...
```

If I hadn’t specified the summary option, *mm* would have returned a dump of all of the above messages in a UNIX mailbox format—this gives us virtual folders, à la Evolution. (<http://www.ximian.com/products/evolution/>)

Let’s add some more intelligent recognizers. Now, I’m a firm believer in 80-percent solutions. Most of the time, the effort required to make an algorithm “perfect” isn’t worth it. Edge cases are, well, edge cases; and if you don’t expect the algorithm to get things right, 80 percent of the time is just fine.

So when I say “intelligent” recognizers, I’m not referring to artificial intelligence; in fact, what the recognizers do is more along the lines of artificial stupidity, leaving the human (who is supposed to have some sort of natural intelligence) to do some elementary top-level filtering. Computers are good at grinding data, so we’ll leave them to do that, and humans are good at top-level filtering, so we’ll leave them to do that.

Think of the recognizers, then, as a production line of trained monkeys. When these monkeys find something interesting or shiny, they file it away in the database, as an “asset.” Assets know which mail message they were found in, and which monkey discovered them.

For instance, when a recognizer attempts to discover any phone numbers in a message, it throws up anything that it can find that looks even remotely like a phone number. This naturally produces one or two false positives—although not that many, since most of the long sequences of numbers, parentheses, and hyphens found in mail messages turn out to be phone numbers anyway—but that’s “Officially OK” by the Mail::Miner design philosophy. After all, if you can very quickly scan through 500 MB of e-mail and produce three candidate phone numbers for someone, two of which are obviously bogus, I’d call that a sufficiently big win.

The phone number recognizer is actually a slightly interesting example, because using it alters the output format of *mm*. Essentially, there are two types of recognizer: those that help to find particular messages, and those that actually store “hard” information. The former type of recognizer produces a mailbox full of messages that match; the latter type of recognizer just dumps out the asset in question.

What does this mean in practice? Well, when you’re asking *mm* about phone numbers, it’s more than likely that you don’t want messages containing phone numbers, but you actually want to get

at the numbers themselves. So, for instance, if I ask Mail::Miner for Tim O’Reilly’s phone number:

```
% mm --from "Tim O'Reilly" --phone
Phone numbers found in message 2863 from "Tim
O'Reilly" <tim@oreilly.com>:
(555) 123-4567
```

(Notice there that I’ve used both the simple “From” query tool and the query tool provided by the phone-number recognizer to form an additive filter.)

Of course, if I want to check this out, I can get a copy of the message in question:

```
% mm --id 2863
From mail-miner-2863@localhost Thu Oct 31 18:58:53
2002
Received: from rock.oreilly.com ([209.204.146.34]
helo=smtp.oreilly.com)
...
```

Or just find out what the mail was actually about:

```
% mm --id 2863 --summary
1 matched
2863:2002-08-31:      "Tim O'Reilly"
<tim@oreilly.com>:Re: Mail::Miner update
```

And speaking of what mails are about, let’s move on to another recognizer, the keywords recognizer.

One interesting aspect of Mail::Miner from my point of view is that it has sparked the development of a few other neat little modules. The first stemmed from the fact that, as discussions tend to drift, a once-relevant “Subject:” line is now completely irrelevant to a particular e-mail. How do I find important details about a forthcoming business trip if they’re hidden in a thread entitled “Return from caller”?

To solve this problem, I came up with the amazingly simple Lingua::EN::Keywords module to extract a set of salient keywords from a block of text. The keywords recognizer runs this module over a mail message, files each keyword returned from Lingua::EN::Keywords as an asset attached to the message, and provides the (synonymous) *--about* and *--keyword* command-line query functions.

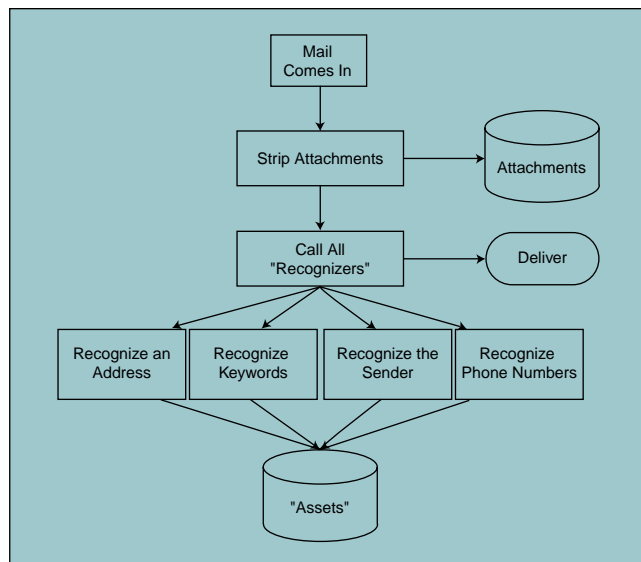


Figure 1: The Framework of Mail::Miner.

The other recognizer that is currently implemented is Mail::Miner::Recognizer::Address. This recognizes physical addresses by looking for something that looks a bit like a postcode or U.S. zip code and state, and filing the entire paragraph.

Database Digression

There now follows a technical digression that you may skip if you're not particularly interested in reading a rant about relational databases.

As we've seen above with `--from ... --address`, queries can be combined. As an interesting technical point, I wanted each query to be represented by a single SQL statement for efficiency. How this works in practice is that we generate `SELECT * FROM messages WHERE` and each recognizer takes its argument, generates a suitable `WHERE` clause, and all the `WHERE` clauses get `AND`-ed together.

When the `SELECT` statement returns a set of messages, another function in each of the recognizers is called to allow them to post-process and filter out inapplicable messages on a more fine-grained basis than can be achieved with raw SQL alone. This is an elegant and efficient design.

Unfortunately, most of the interesting recognizers are looking for messages that contain assets of a particular form. Hence, the `WHERE` clause that they return is actually a subselect along the lines of

```
EXISTS (
  SELECT * FROM assets
  WHERE message = message.id
    AND recogniser = "me"
    AND asset LIKE '%something%'
)
```

Of course, the most popular open source relational database does not support subselects. Hence, for the moment, Mail::Miner requires the second most popular open source relational database, PostgreSQL, until either MySQL gets its act together or I am persuaded that there's an equally elegant design that doesn't use subselects.

It's About the User, Stupid

In the grand tradition of this sort of article, I shall talk at length about currently unimplemented features as though they were up and running.

The other module sparked by the Mail::Miner project came out of the need to express timeframes in a human manner. As well as being a big fan of 80 percent solutions, I'm also a big fan of fuzzy input. Fuzzy input means that the computer, which has an awful lot of processing power when it comes to precise operations, tries its best to understand the human, who isn't all that great at precise operations. This requires admitting that computer programs are there for the user's benefit and not the programmers, but that's a different story and will be told a different time.

So, given that the whole premise of Mail::Miner is that I only vaguely know what I'm looking for, I don't want to have to specify explicit dates and times in order to narrow down searches. If I knew the date and time of the e-mail, I wouldn't need Mail::Miner in the first place!

Instead, I want to be able to say "find me the e-mail I got from Adam sometime around a week ago." The Date::PeriodParser module was written to solve this problem: Given a "fuzzy" date expressed in English, produce a pair of UNIX time values that are likely to bracket the date.

For instance, as I write this on a Thursday night:

```
% perl -MDate::PeriodParser -le 'print scalar
    localtime $_ for
```

THE ONLY TOOLS YOU NEED



Whether you need training in the most basic Perl programming, or you have to hack into the most complex code, Stonehenge Consulting Services, Inc. provides the tools that give you the ability to remain on the cutting edge. We deliver quality Perl training to corporations around the world. Our courseware and presentations are easy to follow, funny, and produce satisfied customers.

Stonehenge Consulting Services delivers top-to-bottom service in providing Perl support services, like training, documentation, and software develop-

ment including design review and code review to reduce maintenance costs and improve the quality of deliveries. Our sharp team of principal engineers, lead by Randal Schwartz, forged these tools to be a cut above the rest, and are recognized worldwide for their contribution to the Perl community.

 **Stonehenge**
stonehenge consulting services, inc.
0333 sw flower st portland, or 97201-3746 usa
tel 503.777.0095 fax 817.249.6793
perl.stonehenge.com

```
parse_period("around the morning of the day  
before yesterday")'
```

```
Mon Oct 28 22:00:00 2002
```

```
Tue Oct 29 14:00:00 2002
```

“Around the morning of the day before yesterday” translates to “between very late on Monday evening to early Tuesday afternoon.” Mail::Miner’s `--date` option will give an interface to this.

It's Not Just About the User

So far in our explorations of Mail::Miner, we’ve seen how it can be used to extract salient information from the incoming e-mail of a single user. However, if the system was to be deployed across an organization with multiple users, this would require each user to have their own database.

Or would it? The natural extension to filing information about how you communicate by e-mail is to file information about how an organization communicates internally and with its clients.

A planned future phase of Mail::Miner is to have it sit on the main mail gateway to a company and file every single incoming and outgoing e-mail. From this simple idea, we can develop a customer relationship management utility—now you know where your clients live, what they do, and more importantly, who’s been speaking to them about what.

So What Next?

The future of Mail::Miner lies in three distinct developments: The first is the development of more specific recognizers; the second, in developing this idea of Mail::Miner as an organization-wide tool; and third, the extension of Mail::Miner from a purely search and retrieval tool to an integrated part of an e-mail client.

In the first category, I intend to work on recognizers that detect and extract place names, dates and times, code snippets, human languages used in an e-mail, and much more.

In the second, I see far more possibilities. Adding a user-defined asset recognizer and asset-management tool will allow you to specify more accurately the details that you want to record from an e-mail; combine this with the idea of restructuring the assets system so that it can be applied either to a particular e-mail or a particular recipient, and you have a system that can store the fact that a particular client has expressed an interest in playing golf, or is going on holiday for the next two weeks, or many other things that a computer cannot pick out, no matter how many monkeys are involved.

Similarly, there could be recognizers that attempt to divine the relationships between correspondents on an e-mail message—if I always cc Joe when I e-mail Amy, then Mail::Miner should take notice of this.

This ties in with the third category, which combines the data-retrieval capabilities of Mail::Miner with the ordinary e-mail client. I’ve already intimated that Mail::Miner can be used to generate virtual mail folders. Imagine if your favorite e-mail client could search read mail based on language or a vague description of when you read it! On this front, I expect to work on IMAP proxies that can use an ordinary mail client to perform Mail::Miner searches, as well as integration with some of the more common UNIX mail clients.

However, for me, Mail::Miner is and always has been a way to make sure I never need to remember anything again—now, who do I have to send this article to? I’m sure I got an e-mail from someone at tpj.com a week or so ago...

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/>



Programming PerlNET

Cameron Laird

Are you an experienced Perl programmer moving into the .NET world or a Microsoft-savvy programmer looking to boost your productivity? If so, you need a copy of *Programming Perl in the .NET Environment*, by Yevgeny Menaker, Michael Saltzman, and Robert J. Oberg.

Programming Perl in the .NET Environment has a straightforward outline: a couple dozen pages explaining .NET, a competent but unremarkable introduction to conventional Perl in the next 130 pages, 250 pages on “Programming with PerlNET,” brief appendices on Visual Studio and C# for Perl programmers, and an adequate index. The quality of the book is high enough to serve everyone working with Perl and .NET, but not so extraordinary as to demand attention from those involved in only one of the two.

To judge the book for yourself, you should have at least an “executive-level” understanding of .NET. Explaining .NET is more challenging than, for instance, explaining HTML or CGI or SQL or other topics popular among Perl programmers. The difficulty is that Microsoft has constructed .NET as an idiosyncratic combination of marketing plan, business strategy, and engineered software. For an accurate technical picture, you must discount at least a portion of .NET’s marketing. It’s only valid in a figurative sense, from our perspective as developers.

Here’s the core of what is technically true: .NET is a framework; that is, an enormous collection of classes that provide functionality common in office automation, so-called “e-commerce,” and related areas, along with XML awareness and other low-level routines. In principle, the framework is language-neutral and perhaps platform-neutral, and even bigger than any one language or operating system. In practice, .NET works best under Windows, and Microsoft’s new C# language is the one best able to exploit .NET.

If you know how much more productive you can be with Perl than C# or Visual Basic, the availability of PerlNET (.NET-hosted Perl) will thrill you. Along with its familiar capabilities as a Perl processor, PerlNET both creates and uses .NET components. This means that you have all of Perl’s flexibility in accessing such key .NET dimensions as the Windows Forms GUI toolkit, the Active Server Pages (ASP) web application system, and the ActiveX Data Objects (ADO) data manager.

I like *Programming Perl in the .NET Environment* for its recognition that programmers with different backgrounds need different information, for its accuracy, and for the richness of the example code explained in the text. The prose of the book is generally well

Cameron Laird maintains the authoritative FAQ on Perl/Tk and writes frequently on languages, networking, and security for The Perl Journal and other magazines. He can be reached at claird@phaseit.net.

***Programming Perl
in the .NET Environment***
*by Yevgeny Menaker, Michael
Saltzman, and Robert J. Oberg*
Prentice Hall PTR, 2003
495 pp., \$44.99
ISBN: 0-13-065206-7

edited and readable; only isolated sections show the flab or lack of polish that suggests editors overlooked them. Small imprecisions regarding typing—dynamic versus static, and strong versus weak—will bother pedants, but neither they nor occasional typographical errors interfere with the main points of the chapters.

The introduction of *Programming Perl in the .NET Environment* includes more cheerleading for Microsoft in its creation of .NET than I prefer. While it’s important to provide clear explanations of .NET’s commercial and engineering advantages, and the authors’ enthusiasm appears genuine, I think the book would have been better with just a bit more reserve.

In principle, PerlNET has at least three distinct roles: development of new applications; reuse of legacy Perl-coded components (all of CPAN, for example); and exposure of a scripting interface to large applications. I’m not certain at all which will prove to be most important. *Programming Perl in the .NET Environment* is aimed at programmers with the Perl lust for high productivity. My impression of large information technology (IT) shops is that Perl’s culture of flexibility and clever concision interests them little. One way to think about .NET and such ancestral technologies as COM and Visual Basic is that they’re designed to lift up masses of mediocre coders, not expand the horizons of elite wizards. IT managers who favor “.NET’s power” generally are talking about things such as business-ready report generators or colorful navigational aids. The large body of Perl programmers seem to regard such pieces as algorithmically uninteresting. What *Programming Perl in the .NET Environment* doesn’t tell us is how the organizational cultures of Perl and .NET will accommodate each other over the next couple of years.

TPJ

Source Code Appendix

Derek Vadala "Creating RSS Files with XML::RSS"

Listing 1

```
<channel rdf:about="http://www.azurance.com">
  <title>azurance.com</title>
  <link>http://www.azurance.com</link>
  <description>Open Source and Security Consulting</description>
  <dc:language>en-us</dc:language>
  <dc:rights>Copyright &amp;copy; 1999-2002, Azurance.com</dc:rights>
  <dc:publisher>Azurance</dc:publisher>
  <dc:creator>derek@azurance.com</dc:creator>
  <dc:subject>Open Source, Security</dc:subject>
  <syn:updatePeriod>hourly</syn:updatePeriod>
  <syn:updateFrequency>4</syn:updateFrequency>
  <syn:updateBase>1999-11-05T09:00:00-05:00</syn:updateBase>
  <items>
    <rdf:Seq>
      <rdf:li rdf:resource="http://www.theregister.co.uk/content/55/27734.html" />
      <rdf:li rdf:resource="http://www.vnunet.com/News/1136204"/>
      <rdf:li rdf:resource="http://www.internetnews.com/infra/article.php/1486121"/>
    </rdf:Seq>
  </items>
</channel>
```

Listing 2

```
1  $rss->add_item(
2
3      title  => "Baltimore launches Trusted Business apps",
4      link   => "http://www.theregister.co.uk/content/55/27734.html"
5  );
6
7  $rss->add_item(
8
9      title  => "FBI investigates major web slowdown",
10     link   => "http://www.vnunet.com/News/1136204"
11 );
12
13 $rss->add_item(
14
15     title=> "Cisco Boosts Security, Caters To Small Business",
16     link => "http://www.internetnews.com/infra/article.php/1486121"
17 );
```

Listing 3

```
1  sub rss_items {
2
3      use DBI;
4
5      my $itemCount = shift @_ ;
6      my ($dsn, $dbh, $sth, $rv, @row);
7
8      my $driver      = "mysql";
9      my $database    = "rss_news";
10     my $hostname     = "localhost";
11     my $port         = "3306";
12     my $user         = "username";
13     my $pw           = "password";
14     my $table        = "news";
15
16     $dsn = "DBI:$driver:database=$database:host=$hostname;port=$port";
17     $dbh = DBI->connect($dsn, $user, $pw);
18     $dbh->{PrintError} = 1; # turn off errors, we'll deal with it ourselves
19
20     $sth = $dbh->prepare("SELECT COUNT(*) FROM news");
21     $rv = $sth->execute;
22     @count = $sth->fetchrow_array;
23     $offset = $count[0] - $itemCount;
24     $sth = $dbh->prepare("SELECT title, link FROM news
25                          LIMIT $offset, - 1");
26     $rv = $sth->execute;
27
28     while (@row = $sth->fetchrow_array) {
29
30         my ($title, $link) = @row;
31         $rss->add_item(
32
33             title  => "$title",
34             link   => "$link"
35         );
36     }
37 }
```

Listing 4

```
<?xml version="1.0" encoding="UTF-8"?>

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://purl.org/rss/1.0/"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:taxo="http://purl.org/rss/1.0/modules/taxonomy/"
  xmlns:syn="http://purl.org/rss/1.0/modules/syndication/"
>

  <channel rdf:about="http://www.azurance.com">
    <title>azurance.com</title>
    <link>http://www.azurance.com</link>
    <description>Open Source and Security Consulting</description>
    <dc:language>en-us</dc:language>
    <dc:rights>Copyright &copy; 1999-2002, Azurance.com</dc:rights>
    <dc:publisher>Azurance</dc:publisher>
    <dc:creator>derek@azurance.com</dc:creator>
    <dc:subject>Open Source, Security</dc:subject>
    <syn:updatePeriod>hourly</syn:updatePeriod>
    <syn:updateFrequency>4</syn:updateFrequency>
    <syn:updateBase>1999-11-05T09:00:00-05:00</syn:updateBase>
    <items>
      <rdf:Seq>
        <rdf:li rdf:resource="http://www.infoworld.com/articles/hn/xml/02/10/24/021024hnnpcwest.xml?s=IDGNS" />
        <rdf:li rdf:resource="http://www.idg.net/ic_959380_1794_9-10000.html" />
        <rdf:li rdf:resource="http://www.infoworld.com/articles/hn/xml/02/10/25/021025hnsecurelinux.xml?s=IDGNS" />
        <rdf:li rdf:resource="http://www.cnn.com/2002/TECH/internet/10/23/net.attack/index.html" />
        <rdf:li rdf:resource="http://www.infoworld.com/articles/hn/xml/02/10/23/021023hnopteron.xml?s=IDGNS" />
        <rdf:li rdf:resource="http://www.businessweek.com/technology/cnet/stories/963054.htm" />
        <rdf:li rdf:resource="http://www.itweb.co.za/sections/internet/2002/0210240947.asp?A=HOME&O=FPIN" />
        <rdf:li rdf:resource="http://www.internetwk.com/security02/INW20021023S0001" />
        <rdf:li rdf:resource="http://www.pcw.co.uk/News/1136211" />
        <rdf:li rdf:resource="http://zdnet.com.com/2100-1105-963087.html" />
      </rdf:Seq>
    </items>
  </channel>

  <item rdf:about="http://www.infoworld.com/articles/hn/xml/02/10/24/021024hnnpcwest.xml?s=IDGNS">
    <title>Network chip makers focus on security</title>
    <link>http://www.infoworld.com/articles/hn/xml/02/10/24/021024hnnpcwest.xml?s=IDGNS</link>
  </item>

  <item rdf:about="http://www.idg.net/ic_959380_1794_9-10000.html">
    <title>'The Golden Age of Hacking rolls on'</title>
    <link>http://www.idg.net/ic_959380_1794_9-10000.html</link>
  </item>

  <item rdf:about="http://www.infoworld.com/articles/hn/xml/02/10/25/021025hnsecurelinux.xml?s=IDGNS">
    <title>Secure Linux maker teams with IBM in U.S.</title>
    <link>http://www.infoworld.com/articles/hn/xml/02/10/25/021025hnsecurelinux.xml?s=IDGNS</link>
  </item>

  <item rdf:about="http://www.cnn.com/2002/TECH/internet/10/23/net.attack/index.html">
    <title>FBI seeks to trace massive Net attack</title>
    <link>http://www.cnn.com/2002/TECH/internet/10/23/net.attack/index.html</link>
  </item>

  <item rdf:about="http://www.infoworld.com/articles/hn/xml/02/10/23/021023hnopteron.xml?s=IDGNS">
    <title>RSA, AMD team up on security for Opteron chips</title>
    <link>http://www.infoworld.com/articles/hn/xml/02/10/23/021023hnopteron.xml?s=IDGNS</link>
  </item>

  <item rdf:about="http://www.businessweek.com/technology/cnet/stories/963054.htm">
    <title>Encryption method getting the picture</title>
    <link>http://www.businessweek.com/technology/cnet/stories/963054.htm</link>
  </item>

  <item rdf:about="http://www.itweb.co.za/sections/internet/2002/0210240947.asp?A=HOME&O=FPIN">
    <title>Internet banking security revolutionised with SMS-based cross-checking</title>
    <link>http://www.itweb.co.za/sections/internet/2002/0210240947.asp?A=HOME&O=FPIN</link>
  </item>

  <item rdf:about="http://www.internetwk.com/security02/INW20021023S0001">
    <title>Vendor Warns Of New IE Holes; Microsoft Calls Reports Irresponsible</title>
    <link>http://www.internetwk.com/security02/INW20021023S0001</link>
  </item>

  <item rdf:about="http://www.pcw.co.uk/News/1136211">
    <title>PGP poised for major comeback</title>
    <link>http://www.pcw.co.uk/News/1136211</link>
  </item>

  <item rdf:about="http://zdnet.com.com/2100-1105-963087.html">
    <title>P2P hacking bill may be rewritten</title>
    <link>http://zdnet.com.com/2100-1105-963087.html</link>
  </item>
```

</rdf:RDF>

Max Schubert “Making a Cross-Platform Installer with Perl”

Listing 1

```
#!/bin/sh

PATH=$PATH:/usr/local/bin:/usr/local/gnu/bin:/usr/gnu/bin:/opt/bin:/opt/gnu/bin
PATH=$PATH:/usr/bin:/bin:/usr/ccs/bin
export PATH

CC=gcc
export CC

PERL=$1
DIR=`pwd`

gzip -d -c ./expat-1.95.5.tar.gz | tar xvf -
cd ./expat-1.95.5
./configure --prefix=$DIR
make
make install
cd ..
rm -rf ./expat-1.95.5

gzip -d -c ./XML-Parser-2.31.tar.gz | tar xvf -
cd ./XML-Parser-2.31/
$PERL Makefile.PL EXPATLIBPATH="$DIR/lib" EXPATINCPTH="$DIR/include" CC=gcc LD=gcc
make
make install
cd ..
rm -rf ./XML-Parser-2.31
```

Listing 2

```
sub install_xml_parser {
    my $perl = shift;
    my $dir = shift;

    Screen::clear();
    print "Installing expat library and XML::Parser .. \n";
    Screen::pause();

    if (exists $ENV{'LD_LIBRARY_PATH'}) {
        $ENV{'LD_LIBRARY_PATH'} .= ":$dir";
    } else {
        $ENV{'LD_LIBRARY_PATH'} = "$dir";
    }

    system("cd $dir; sh ./make_expert $perl");

    print "XML::Parser installed.\n";
    Screen::pause();
}
```

Listing 3

```
#!/bin/sh

PATH=$PATH:/usr/local/bin:/usr/local/gnu/bin:/usr/gnu/bin:/opt/bin:/opt/gnu/bin
PATH=$PATH:/usr/bin:/bin:/usr/ccs/bin
export PATH

CC=gcc
export CC

PERL=$1
DIR=`pwd`

gzip -d -c ./openssl-0.9.6g.tar.gz | tar xvf -
cd ./openssl-0.9.6g
./config --prefix=$DIR
make
make install
cd ..
rm -rf ./openssl-0.9.6g

gzip -d -c ./Crypt-SSLeay-0.45.tar.gz | tar xvf -
cd ./Crypt-SSLeay-0.45/

OPENSSL_DIR=$DIR
export OPENSSL_DIR

$PERL Makefile.PL CC=gcc LD=gcc
make
make install
cd ..
```

```
rm -rf ./Crypt-SSLeay-0.45
```

```
exit 0
```

Listing 4

```
my $perl = $pool->get('PERL');

# Close standard error so user doesn't see failure messages for modules
# that don't exist.

close(STDERR);

my $cmd;

# On windows, no such thing as Bundle::LWP, on UNIX/Linux,
# just want to test for Bundle so dependencies are done correctly.

if (Util::iswin()) {
    delete $modules{'Bundle::LWP'};
} else {
    delete $modules{'LWP'};
}

for (sort keys %modules) {
    $cmd = "$perl -e \"use lib q!$lib!; use $_ $modules{$_};\"";
    if (system($cmd)) {
        push(@need, $_);
    } else {
        push(@have, $_);
    }
}

open(STDERR, '>&0');
```

Listing 5

```
# For local, non-root installs
$ENV{'PERL_MM_OPT'} = " LIB=$dir" if defined $dir;

my $cpanrc = "$ENV{'HOME'}/.cpan/CPAN/MyConfig.pm";
unless (-f $cpanrc) {
    initialize_cpan($cpanrc, $perl);
}

sub initialize_cpan {
    my $cpanrc = shift;
    my $perl = shift;

    use File::Basename;

    Screen::clear();
    print "Initializing CPAN (follow on-screen instructions:\n";
    Screen::pause();

    my $dir = File::Basename::dirname($cpanrc);
    DirCopy::make_path($dir, 0700);

    system("$perl -MCPAN::FirstTime -e 'CPAN::FirstTime::init(q!$cpanrc!)'");

    print "Initialization complete! On to module install.\n";
    Screen::pause();
}
```

Tim Kientzle “Perl & Rapid Database Prototyping”

Listing 1

```
# Simple accessor
sub foo {
    my $a=$FOO;
    ($FOO)=@_ if @_;
    return $a;
}

# Using accessors
foo(4); # foo = 4
print foo(5); # Print 4, set foo=5
print foo(); # Print 5
```

Listing 2

```
# Access a row
$e = Employee->byName('Doe', 'John');
# Get phone number
print $e->phone();
# Set phone number
$e->phone('555-5555');
```


Listing 3

```
CREATE TABLE employee(  
    employee_id INT PRIMARY KEY,  
    first       CHAR(80),  
    last        CHAR(80),  
    phone       CHAR(80),  
    manager_id INT  
);
```

Listing 4

```
package Employee  
use vars qw(@ISA);  
@ISA = qw(DBTable);  
sub byName {  
    my($class,$last,$first) = @_;  
    my $self = bless {'TABLE'=>'employee'},$class;  
    # _fetchOrCreate returns a hash  
    # { 'employee_id' => <number> }  
    $self->{'KEY'} = $self->_fetchOrCreate(  
        {'last' => $last,'first' => $first}, 'employee_id' );  
    return $self;  
}
```

Listing 5

```
# An accessor returning an object  
sub manager {  
    my($self) = @_;  
    return Employee->byId($self->manager_id);  
}  
# Another constructor  
sub byID {  
    my($class,$id) = @_;  
    my $self = bless {},$class;  
    $self->{'TABLE'} = 'employee';  
    $self->{'KEY'} = $self->_fetchOrCreate  
        ({'employee_id' => $id}, 'employee_id' );  
    return $self;  
}
```

Listing 6

```
# An optimized employee_id accessor  
sub employee_id {  
    my $self = shift;  
    $self->{'KEY'}->{'employee_id'};  
}
```

Listing 7

```
# An optimized Employee class  
package Employee  
use vars qw(@ISA);  
@ISA = qw(DBTable);  
sub byName {  
    my($class,$last,$first) = @_;  
    my $self = bless {},$class;  
    my $cols = $self->_columns;  
    $self->{'TABLE'} = 'employee';  
    # Read and cache all of the columns  
    $self->{'FIELDS'} = $self->_fetchOrCreate  
        ({'last' => lc $last, 'first' => lc $first}, keys %$cols);  
    $self->{'KEY'} =  
        { 'employee_id' => $self->{'FIELDS'}->{'employee_id' } };  
    return $self;  
}  
# Override _fetch to just pull the value from memory  
sub _fetch {  
    my ($self, $key, $col) = @_;  
    return { $col => $self->{'FIELDS'}->{$col} };  
}  
# Override _set to modify the in-memory value  
sub _set {  
    my($self,$key,$set) = @_;  
    $self->SUPER::_set($key,$set); # Set fields in DB  
    foreach $k (keys %$set) {  
        $self->{'FIELDS'}->{$k} = $set->{$k};  
        # Remember to update the value in 'KEY', if it's there  
        if(exists $self->{'KEY'}->{$k}) {  
            $self->{'KEY'}->{$k} = $set->{$k};  
        }  
    }  
}
```

Listing 8

```
package DBTable;  
use Carp;  
use DBI;
```

```

# Get/set the active DB handle in a global variable
my $db;
sub dbHandle {
    my $self = shift;
    my $olddb = $db;
    if(@_) { $db = shift; }
    return $olddb;
}
sub DESTROY { $db->finish }
# Query columns in the given row.
# The row is specified using a hash of column=>value pairs.
# Note: 'undef' is returned if there is no such row or the column is
null.
sub _fetch {
    my($self,$keyHash,@cols) = @_;
    my $sql = "SELECT ". join(', ',@cols)." FROM "
        . $self->{'TABLE'}
        . " WHERE "
        . join(' AND ', map {"$_=?"} keys %$keyHash);
    my $sth = $self->dbHandle->prepare($sql);
    $sth->execute(values %$keyHash);
    if(my $hr = $sth->fetchrow_hashref) {
        $sth->finish;
        return $hr;
    } else {
        return undef;
    }
}

# Create a row and return a hash with the values of columns @cols
sub _create {
    my($self,$keyHash,@cols) = @_;
    my $table = $self->{'TABLE'};
    my $sql = "INSERT INTO $table (" . join(', ',keys %$keyHash). ")"
        . " VALUES(" . join(', ',map {'?'} keys %$keyHash) . ")";
    my $sth = $self->dbHandle->prepare($sql);
    $sth->execute(values %$keyHash);
    $self->_fetch($keyHash,@cols);
}

# Try to 'fetch' from a particular row; if it fails, create the row
sub _fetchOrCreate {
    my $self = shift;
    my $result = $self->_fetch(@_);
    if(!defined $result) { $result = $self->_create(@_) }
    return $result;
}

# Set columns in the specified row to the given value; returns a count
# of rows modified.
sub _set {
    my($self,$keyHash,$updates) = @_;
    my $sql = "UPDATE ". $self->{'TABLE'}." SET "
        . join(', ', map {"$_=?"} keys %$updates)
        . " WHERE ". join(' AND ', map {"$_=?"} keys %$keyHash);
    return $self->dbHandle->do($sql,{},values %$updates,values
%$keyHash);
}

# Returns a hash ref mapping each valid column name to 1.
# Caches the result in the object under key 'COLUMNS'.
sub _columns {
    my($self) = @_;
    if(!$self->{'COLUMNS'}) {
        my $sql = "SELECT * FROM ". $self->{'TABLE'}." WHERE 1=0";
        my $sth = $self->dbHandle->prepare($sql);
        $sth->execute();
        $self->{'COLUMNS'} = { map { ( $_>1 ) } @{$sth->{'NAME'}} };
        $sth->finish();
    }
    return $self->{'COLUMNS'};
}

# Allow easy reference to $object->column for any column. With no
# argument, just returns the value; with argument also sets the value.
our $AUTOLOAD;
sub AUTOLOAD {
    my($self,$value) = @_;
    my $col = $AUTOLOAD;
    die "Cannot auto-access column $col without an object!" if ! ref
$self;
    $col =~ s/.*://; # Strip off leading package name
    if(!$self->_columns->{$col}) {
        croak("No column \"$col\" in table \"".$self->{'TABLE'}."\"");
    }
    my $oldValue = ($self->_fetch($self->{'KEY'},$col)->{$col});
    if($value) { $self->_set($self->{'KEY'},{$col=>$value}) }
    return $oldValue;
}

```

Listing 9

```
#!/usr/bin/perl
use DBI;
use DBTable;

package Employee;
use vars qw(@ISA);
@ISA = qw(DBTable);
sub byName {
    my($class,$last,$first) = @_ ;
    my $self = bless {},$class;
    $self->{'TABLE'} = 'employee';
    $self->{'KEY'} = $self->_fetchOrCreate
        ({'last' => lc $last,
          'first' => lc $first,
          'employee_id' => });
    return $self;
}

package main;
use DBI;
DBTable->dbHandle(DBI->connect("DBI:mysql:test", 'test', 'test'));

my $employee = Employee->byName('Kientzle','Tim');
print $employee->phone(),"\n";
print $employee->last , " , " , $employee->first, "\n";
print $employee->phone('555-5555'), "\n";
print $employee->phone('555-5555'), "\n";
print $employee->junk; # Force error message for non-existent column
```