

# *The Perl Journal*

## Encrypting Web Pages on a Server

Craig Riter • 3

## Determining List Relationships With *List::Compare*

James E. Keenan • 6

## Automating Distributions With *scriptdist*

brian d foy • 11

## Eight Million Ways to *die*

Randal L. Schwartz • 13

## PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Jack J. Woehr:

*Perl Medic: Optimizing Legacy Code* • 16

Source Code Appendix • 17

## LETTER FROM THE EDITOR

### Template Toolkit is Too Cool

I love tools that seem to do their work magically. You know the ones I mean—those that just make all the right assumptions about what you want. The Template Toolkit (<http://template-toolkit.org/>) is one such tool. It's an extraordinarily popular system, so it's a good bet you already know about it. But that won't deter me from singing its praises. The Template Toolkit (TT) is, well, a templating mechanism. At its simplest, it can be used like this:

```
use Template;
$var = {
    greeting => 'Hello',
};
$tt = Template->new();
$tt->process("mytemplate.tt", $var);
```

where "mytemplate.tt" is a template file. This code writes the contents of mytemplate.tt to STDOUT, replacing any [% greeting %] template directives with the string "Hello."

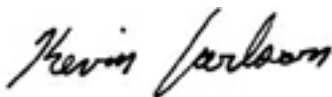
Of course, instead of \$var in this example, you could pass in just about anything in your code, including object references. This is a godsend. Hash elements and array elements can be accessed in template directives with a simple dot notation, such as [% foo.0.bar.1%].

You can almost forget about how your data will need to be formatted on output. Just store your data in a place where you can get to it easily, build your templates wisely, and TT will take care of the rest.

TT also provides some built-in functions (or *filters*, in TT parlance) that let you munge your output data in lots of useful ways. You can even write your own filters. Taken together, all the parts of TT represent an entire language for writing templates, complete with *if* statements and looping constructs. It's so rich, in fact, that it's tempting to offload much of your data processing to your templates, which can make for messy templates. Like any good Perl tool, TT doesn't dictate where that balance should lie—it leaves up to you the decision of whether or not to get carried away.

Another, less obvious virtue is that TT promotes good design. While templating in general is good practice because it separates data from presentation, and allows your applications to be more easily customized by users, TT's particular brand of template mechanism goes a step further and promotes good *object* design. The cleaner and more sensibly designed your data structures are, the easier they are to reference with TT directives. To simplify my own TT directives, I have frequently gone back to the drawing board and come up with a better way of internally representing data. It's an example of good form promoting good form.

If you haven't looked at the Template Toolkit, do yourself a favor and try it out. I guarantee you'll find a use for it.



Kevin Carlson  
Executive Editor  
kcarlson@tpj.com

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

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



### The Perl Journal

#### EXECUTIVE EDITOR

Kevin Carlson

#### MANAGING EDITOR

Della Song

#### ART DIRECTOR

Margaret A. Anderson

#### NEWS EDITOR

Shannon Cochran

#### EDITORIAL DIRECTOR

Jonathan Erickson

#### COLUMNISTS

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

#### CONTRIBUTING EDITORS

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

#### INTERNET OPERATIONS

##### DIRECTOR

Michael Calderon

##### SENIOR WEB DEVELOPER

Steve Goyette

##### WEBMASTERS

Sean Coady, Joe Lucca

#### MARKETING / ADVERTISING

##### PUBLISHER

Timothy Trickett

##### MARKETING DIRECTOR

Jessica Hamilton

##### GRAPHIC DESIGNER

Carey Perez

#### THE PERL JOURNAL

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

#### CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND

MARKETING Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

PRESIDENT, TECHNOLOGY SOLUTIONS Robert Faletta

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED

TECHNOLOGIES Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER ENTERPRISE

ARCHITECTURE GROUP Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE

DEVELOPMENT MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING

SOLUTIONS Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES

Marie Myers

# Perl News

## Perl 5.8.4 Released

As expected, Perl 5.8.4 followed closely on the heels of RC2. The `suidperl` issues that caused problems in RC1 were solved by removing the `setuidperl` executable; instead “to preserve backwards compatibility with scripts that invoke `#!/usr/bin/suidperl` the only set `uid` binary is now `sperl5.8.n` (`sperl5.8.4` for this release). `suidperl` is installed as a hard link to Perl; both `suidperl` and Perl will invoke `sperl5.8.4` automatically as the set `uid` binary, so this change should be completely transparent,” explained the release documentation.

## Make an Event of It

The schedule has been posted for YAPC North America, which will take place in Buffalo, New York, June 16–18. The three-day conference will feature Allison Randal delivering the keynote address; Damian Conway speaking on Perl 6 and “a series of useful modules whose interface is...nothing;” Mark Fowler explaining how to prepare a CPAN distribution; as well as many other well-known speakers covering topics that range from career planning to writing Perl without a text editor. (See [http://yapc.org/America/talk\\_desc.shtml](http://yapc.org/America/talk_desc.shtml) for the full list of talks.) Evening events will include an IMAX showing of *Harry Potter and the Prisoner of Azkaban*, sponsored by O'Reilly, and a tour of the University of Buffalo's Center for Computational Research. Proposals for lightning talks at YAPC::NA are still being accepted; you can submit ideas at <http://justanotherperlhacker.org/cgi-local/lightning/submit.pl?conference=yapcna2004> until June 8.

## Perl Mongers Spread to New Orleans

Things just got easier for Perl hackers in the Big Easy: NewOrleans.pm has been officially launched, with a wiki at <http://neworleans.pm.org/> and a mailing list archived at <http://mail.pm.org/mailman/listinfo/neworleans-pm>. The first meeting was held May 14 at the Fair Grinds Coffeehouse, and subsequent meetings are scheduled for the second Friday of each month. The group is already making plans to tackle new Perl projects such as “a clean, easy-to-use, object-oriented CPAN package that would allow for the merging of record data into a printable output,” or “a module that makes it easy to build interpreters for XML-based languages.”

## That's Amore

The Italian Perl Mongers have finished a labor of love: They've translated the entire Perl functions documentation (`perlfunc.pod`) into Italian. It's all available now at <http://www.perl.it/documenti/perlfunc/index.html>, along with an Italian translation of the Perl FAQ documents (`perlfaq*.pod`). The group plans to post a POD2::IT

module to CPAN that will contain the translated documentation. Their work is part of the Italian PerlDoc Translation Project, hosted on SourceForge at <http://pod2it.sourceforge.net/>.

## Polly Want a Compiler

Vishal Vatsa, a graduate student at the National University of Ireland, has undertaken porting GCC to the Parrot architecture as his Master's research project. While the project is still in its opening stages—“I have managed to create a token backend for Parrot so far, nothing really great (i.e. does not work at the moment),” Vatsa posted on the `perl6-internals` list—a successful implementation would allow Java code to compile to Parrot bytecode. Vatsa tracks his progress at <http://www.parrot.cs.may.ie/>.

For others looking to help out with Parrot and Perl 6, chromatic on the `perl6-language` list suggested resurrecting the P6 Stories project. Initially conceived as a month-long experiment “to see how far we could get in describing the current implementation requirements for Perl 6 in terms of XP-style story cards,” the wiki has been languishing recently. It awaits new contributors at <http://p6stories.kwiki.org/>.

Meanwhile, Dan Sugalski threw down a challenge for all the Forth lovers out there: “Your task, if you choose to accept it, is to turn `languages/forth/forth.pasm` into a loadable compiler module that you can compile workable Forth code with. (Adding new base words are optional, though that certainly won't hurt). There's some stub code in there as it is to wrap a sub `pmc` around a Forth word, but being stub it doesn't actually work. This'd be a good opportunity to fix that.”

## Bricolage 1.8.0

Bricolage, the open-source content-management system written in Perl using `HTML::Mason` and `Apache/mod_perl`, is now in version 1.8.0. This release has been a year in coming and represents the work of 20 independent developers as well as contributions from various companies worldwide. New features cited in the release announcement include “performance boosts to search queries and URI uniqueness validation; e-mail distribution; a greatly simplified templating API; template sandboxes to enable template development without interfering with production templates; support for Template Toolkit templates (<http://www.template-toolkit.org/>); new “Publish” and “Recall” permissions for improved workflow management; per-user preferences; document formatting at publish time, rather than publish scheduling time; new German and Mandarin localizations; image thumbnails and icons for all media documents; and support for HTMLArea WYSIWYG editing (<http://www.interactivetools.com/products/htmlarea/>).” The Bricolage home page is <http://www.bricolage.cc/>.

# Encrypting Web Pages On a Server

Sometimes I am called paranoid, but I like to think of it as being careful. I have some web pages that I would like to be able to access from anywhere, but keep anyone else from seeing.

Having access to a file from anywhere I wish can be handled easily by hosting the file on my web site, but protecting the data is a little more difficult. Of course, the web page will be password protected, and will be on a secure (SSL) web server so that it isn't publicly accessible and no one can view the data as it traverses the Internet from the web site to the browser. But what about when the data is sitting on the server in a file? If I store it as an HTML file for the web server to access, then the permissions will be open enough for anyone on the server to view the file. I may be able to restrict the permissions so that only the administrator of the site can get to it, but I want to feel secure in the knowledge that only I can view the data, even if the server is compromised.

Just as the connection between the server and my browser is encrypted, then encrypting the file on the server will protect it from prying eyes. The file would be secure even if someone somehow obtained access to the file.

So the question, then, is how to encrypt and decrypt the file. This is where Perl and CPAN come in. I can write a script that will read the file in, encrypt the data, and save the file on the server. The script would also be able to decrypt the file, given the correct passphrase, and send the file to the browser.

Looking through CPAN led me to the Crypt modules, one of which was *Crypt::Rijndael*. I remembered reading that this algorithm was chosen by the U.S. Government as the new Advanced Encryption Standard (AES), so it should be good at protecting my data. After reading the details on usage, however, I noticed that the input data had to be exact multiples of the blocksize or the encrypt function would croak. I also had to supply a key, not just a password or passphrase. There was even an option for specifying the encryption mode: Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feed Back (CFB), and several others (see <http://www.rsasecurity.com/rsalabs/faq/2-1-4.html> for a def-

inition of the modes). While I did remember some of the differences between these modes, and that I should use CBC as was noted in the docs, I began to suspect that maybe there was a better module to use.

I then took a look at *Crypt::CBC*, which provides an interface to let me pick a cipher (encryption algorithm), uses an arbitrary length passphrase (key), and ensures that the data is of the correct block size (padding it if needed). This was certainly better than interfacing with *Crypt::Rijndael* directly, but there was one more option that I wasn't quite sure about.

The Initialization Vector (IV) could either be set or randomly generated. The IV is important with block ciphers because it is part of the input in encrypting the first block of data. I didn't realize the impact of this until I hard-coded the IV value and noticed that encrypting the same data with the same passphrase yielded the same encrypted data. Of course, this makes sense, given that the same three inputs (passphrase, data to encrypt, and IV) were used for the encryption. I did notice that the *Crypt::CBC* has an option to randomly generate the IV and then place this value at the start of the encrypted data. While this is an effective solution, I was a little concerned about putting this data into the encrypted file.

Some more searching on CPAN turned up the *Crypt::OpenPGP* module, which is a pure Perl implementation of the OpenPGP standard (RFC2440). I did not want to use the public-key aspects of the module, just the symmetric encryption, since I was both encrypting and decrypting the data file. *Crypt::OpenPGP* turned out to be the module that I needed: It was designed to encrypt individual files, and the Perl module has a very simple interface for encryption and decryption, requiring nothing more than a passphrase. It also handles the plaintext file in memory so that the data is not written to disk, thus reducing further exposure of the data.

## Installing *Crypt::OpenPGP*

*Crypt::OpenPGP* is not the easiest module to install due to all of the dependencies. I had some trouble when I found out that one of the dependencies, *Math::Pari*, required an external library. I got around these problems by using the ports system on FreeBSD

---

*Craig is a consultant at n-Link Corp. in Seattle, Washington. He can be contacted at [craig@riter.com](mailto:craig@riter.com).*



but you should be able to follow the instructions of the *Math::Pari* module to get everything configured properly.

## Using the Script

When first accessing the script, you are presented with two forms: one to encrypt an HTML page and store it on the server and the second to decrypt a page stored on the server. The encrypt form has input fields for the filename, passphrase, and text data (see Figure 1). The filename is used as the name of the file when storing the encrypted data on disk. This name is also used on decryption to identify the file to decrypt.

The passphrase is used to encrypt the data. While there is no minimum length on the passphrase, it is best to use one greater than six characters using letters, numbers, and punctuation. The text data is the entire file that should be encrypted. The data needs to be a complete HTML file since only this data will be returned to the browser when it is decrypted.

The encrypted file can be decrypted either by using the form on the page as shown in Figure 1 or by accessing the script with the name of the file to decrypt as the extra path info. An example of this is `https://server/cgi-bin/securepage.pl/filename/`. When decrypting this way, a form is returned so that the passphrase can be entered.

## Anatomy of the Script

There is a configuration variable for the directory that should be used when storing the encrypted files and another variable for the encryption algorithm (or cipher) to use when encrypting the file. *Crypt::OpenPGP* does not need the cipher specified when de-

crypting the file because the cipher used encrypting the file is stored in the file.

There is a check to ensure that the script is being accessed over a secure link. Because the data is being encrypted on the server, it makes sense that we ensure that the link is also secure. The HTTPS environment can be checked as shown in Listing 1. The `$HOST` and `$SCRIPT` variables are set so that a full URL can be constructed for redirecting to the secure URL. It is assumed that

---

*I want to feel secure in the  
knowledge that only I can view  
the data, even if the server is  
compromised*

---

The screenshot shows a Mozilla Firefox browser window titled "secure page - Mozilla Firefox". The address bar shows a URL starting with "https://". The browser's menu bar includes File, Edit, View, Go, Bookmarks, Tools, and Help. Below the menu bar is a toolbar with navigation buttons (back, forward, home, stop, reload) and a search bar. The main content area displays two forms. The first form, titled "Encrypt File", has three input fields: "Filename:", "Passphrase:", and "Text Data:". Below these fields is an "Encrypt" button. The second form, titled "Decrypt File", has two input fields: "Filename:" and "Passphrase:". Below these fields is a "Decrypt" button. At the bottom of the browser window, there is a status bar that says "Done".

Figure 1: The encrypt form.

the secure URL is constructed by only changing the protocol from http to https.

The input parameters to the script need to be retrieved from the CGI.pm object (Listing 2). The `$passphrase` and `$data` variables can be taken as is, but the `$filename` variable needs to be checked to ensure that there are no slashes (/) in the value. We don't want the filename to be used to backtrack up the directory chain and potentially gain access to other files on the system.

Next, the script needs to determine what the state of the request is. Did the script get called for the first time, did the user click the Encrypt button or the Decrypt button? In Listing 3, you see the tests for state control of the script. It starts with getting the values of the two submit buttons, "decrypt" and "encrypt." Remember that submit buttons set a parameter value in the request.

The first *if* condition tests whether the decrypt button was clicked and also that the filename variable has been set. If both are true, then the subroutine `PerformDecrypt_OpenPGP` is called with the passphrase and filename. The second *if* tests the `$pathinfo` variable, which was set from the `PATH_INFO` environment variable. `$pathinfo` will not be empty if the script is called with extra path info (path in the URL after the script name). With `$pathinfo` having a value, the *f* (or filename) parameter of the CGI request is set to `$pathinfo` minus the first "/" character. `ReturnDecryptForm` is called to display a form to get the passphrase to decrypt the file. The single parameter to this subroutine is described later.

The third *if* condition tests the `$encrypt` variable to see whether the encrypt button was clicked. If true, the `PerformEncrypt_OpenPGP` subroutine is called with the passphrase, filename, and text data. The final *else* condition is executed when the script is first requested and calls the `ReturnInputForms` subroutine. Following the *if...elsif* construct, `exit()` is called because the script has completed its execution.

The `ReturnInputForms` subroutine displays the encryption form to encrypt new files and decryption form to display encrypted files. I used the `CGI.pm` object to write out all of the tags, but mostly because I can never remember the attributes for using a

multipart form. I used the `start_multipart_form()` method so that I could easily add the function of uploading a file instead of typing the file in the `textarea` field. However, I decided against enabling this feature, since I didn't want the data I was trying to keep secure written to disk without encryption during the upload process. Also, the form method is POST and not GET to ensure that the passphrase and other data is not sent in the URL of the request. This subroutine finally calls `ReturnDecryptForm` to display the decryption form.

`ReturnDecryptForm` prints the html form. Since it can be called to print a standalone HTML page or as part of a larger page, it has a parameter to control whether the opening and closing `<html>` and other tags should be printed. A True (1) parameter value prints the `<html>` tags.

The `PerformEncrypt_OpenPGP` sub creates a new `Crypt::OpenPGP` object and encrypts the data with the passphrase. The encrypted data or ciphertext is then written to the filename. The parameters to the `encrypt` sub are the data to be encrypted, the passphrase and the encryption algorithm:

```
## encrypt the data
my $ciphertext = $pgp->encrypt(
    Data => $data,
    Passphrase => $passphrase,
    Compress => 'Zlib',
    #Armour => 1,
    Cipher => $enc_algorithm
);
```

I have also specified that the data be compressed before encryption. This is not required, but I wanted to save disk space on the server. The `Armour` parameter can be used to have the output "ASCII-armored," which creates a text file from the binary encrypted file. The default is to not create it ASCII-armored, and since armoring the file will only increase the file size just to make it viewable, I'll skip it to save on disk.

The resulting ciphertext is written to the file as specified by the filename and the directory set in the configuration. Finally, a short page is returned indicating that the file has been encrypted and a URL

printed that can be used to access the file. This URL will display the decryption form, which prompts for the passphrase of the file.

The `PerformDecrypt_OpenPGP` subdirectory decrypts the file using the given passphrase and returns it to the browser. First, the file name is tested to see that it exists; if not, an error is returned stating that this file does not exist. Next, the `Crypt::OpenPGP` object is created and the `decrypt` subdirectory called. Since OpenPGP can read the ciphertext directly from the file, only the filename and passphrase are passed to the `decrypt` sub.

```
## decrypt the ciphertext
my $plaintext = $pgp->decrypt(
    ## specify filename and have it read the file from disk
    Filename => $enc_dir.$filename ,
    Passphrase => $passphrase,
);
```

The plaintext is then printed to the browser because it is assumed that the decrypted file is an HTML web page.

## Next Steps

Currently, this solution assumes that the encrypted page is a single, self-contained web page. It can't have any references to another encrypted page such as an image file, javascript file, or css file. It should be possible to have the first decrypted page decrypt the others, but I haven't thought this whole problem through. Another enhancement that would be nice is to encrypt a database file and possibly the script to execute on it. This should be an easy enough enhancement because instead of decrypting the page and sending that to the browser, it could decrypt a script and database file and then execute the script. But I'll leave both of these as an exercise to the reader or maybe another article.

## Reference

CBC (<http://www.bletchleypark.net/crypt/modes.html>).

TPJ

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

### Listing 1

```
## get the Name of the host for the server and the script name
## will be used when constructing the full url for this script
my ($HOST) = $ENV{SERVER_NAME} =~ /(.*?)$/s; # untaint
my ($SCRIPT) = $ENV{SCRIPT_NAME} =~ /(.*?)$/s; # untaint

## check to see that this script is on a secure link
if ( $ENV{HTTPS} ne 'on' ) {

    print "<strong>ERROR: you should access this over a secure
link.</strong><p>";
    print " Maybe you want <a
href=\"https://\".$HOST.$SCRIPT.\">https://\".$HOST.$SCRIPT.\"</a>";

    exit;
}
```

### Listing 2

```
my ($passphrase) = $q->param('pwd') =~ /(.*?)$/s;
## make sure there are no extra slashes
my ($filename) = $q->param('f') =~ /([^\/*]*)$/s;
my ($data) = $q->param('data') =~ /(.*?)$/s;
```

### Listing 3

```
## determine what button was clicked, and what action should be performed

## get submit button values for state control
my ($decrypt) = $q->param('decrypt') =~ /(.*?)$/s;
my ($encrypt) = $q->param('encrypt') =~ /(.*?)$/s;
my ($pathinfo) = $ENV{'PATH_INFO'} =~ /(.*?)$/s;
```

```
## the decrypt button was clicked and a filename has been specified
if ($decrypt ne '' && $filename) {
    &PerformDecrypt_OpenPGP($passphrase, $filename);
}
## the PATH_INFO is set with the name of the file to decrypt
elsif ($pathinfo ne '') {
    ## populate the file (f) field since that is what is expected
    ## CGI.pm will populate the field in the form
    $q->param('f', substr($pathinfo, 1));

    ## return form to request the password from the user
    ## param = 1, to have a full HTML page returned
    &ReturnDecryptForm( 1 );
}
## encrypt button was clicked
elsif ( $encrypt ne '' ) {
    &PerformEncrypt_OpenPGP($passphrase, $filename, $data);
}
## page requested for the first time
else {
    &ReturnInputForms();
}

exit;
```

TPJ

# Determining List Relationships with *List::Compare*

**T**wo years ago, I was hired by the continuing education department of a local college to teach a six-week introductory course in Perl. The circumstances of my hiring were a bit abrupt: The person originally hired for the course quit the week before the course began. I was hired two nights before my first session, so I needed to develop course materials, and fast.

Perl, of course, came to my rescue. I decided to use Randal L. Schwartz's *Learning Perl* (O'Reilly & Associates, 2001) as the course text and to create an HTML-based slideshow as my curriculum. I was somewhat familiar with such slideshows because, at the first meeting of one of our local user groups, Perl Seminar New York, my colleague Mark Miller had demonstrated such a slideshow. But that show displayed photos; I wanted to display text. I did a quick hack on Mark's script so that I could accomplish the following:

- Write a separate plaintext file, written in Perl's Plain Old Documentation (POD) format, for each slide in the show. Each such plaintext file's name ended in `.slide.txt`:

```
chomp.slide.txt
defined.slide.txt
# false.slide.txt # commented-out for illustrative purposes
hello.slide.txt
intro1.slide.txt
print.slide.txt
```

- Place the name of each slide in a master *slidelist* file.
- Control the order in which slides appear in the show by simply moving their names up or down within *slidelist*. For instance, a more logical order for the slides above might be:

```
intro1.slide.txt
hello.slide.txt
print.slide.txt
chomp.slide.txt
defined.slide.txt
```

## A First Attempt at Using Seen-Hashes

I had to make certain that, for each slide named in *slidelist*, the corresponding POD file actually existed in a particular subdirec-

tory, *texts*, where I was writing such files. Put another way, I had to compare the list of filenames in *slidelist* with the list of actual files in *texts* and make sure that the first list was a mathematical subset of the second list. To code this, I first turned to Tom Christiansen and Nat Torkington's *Perl Cookbook* (O'Reilly & Associates, 2003) and its discussion of what I have since come to call "seen-hashes"; i.e., hashes that record whether certain strings have been "seen" in particular lists. (These can be thought of as a kind of lookup table.)

In Listing 1, I show the code I wrote before refactoring and optimizing. I first read the filenames in *slidelist* (lines 5–13); next read the POD files in *texts* (lines 15–17); then construct seen-hashes for each (lines 19–21). To determine whether *@selections* is a proper subset of *@sources*, I declare *\$subset\_status* and set its initial value to "1." I loop through the elements in *@sources* and if I encounter one that is not also a key in *%seen\_selections*, I set *\$subset\_status* to a false value of "0" and exit the loop (lines 23–29).

At this point, I wanted to be warned if I had included a filename in *slidelist* that did not actually exist in *texts*. Lines 31–40 accomplish this and shut down the script early if those files are missing.

Next, I decided it would be nice to get a list of those slides for which I had created files but was not currently including in *slidelist*. Lines 50–60 accomplish this. So at this point, I had a validated list of slides I was ready to turn over to the slidebuilder mechanism itself (code not shown here).

Along with my slideshow, I was developing some basic Perl scripts that I distributed to my students in a printed handout. As not all these scripts made it into the final version of the handout, I created another master file, this time called *scriptslist*, with which I could control the order in which the selected scripts appeared in the handout—just as I did for the slides with *slidelist*. Except for some variable names, the code was exactly the same as in Listing 1.

## Refactoring

At that point, I heard Mark Jason Dominus's voice crying out, "Repeated code is a mistake!" Beginning with Yet Another Perl Conference::North America in Pittsburgh in 2000, I had attended several of Mark's "Perl Program Repair Shop" talks and had absorbed two key points:

- Code that is repeated within a single script should be refactored into a subroutine.

---

James founded Perl Seminar New York in 2000, and is Treatment Team Leader for the New York State Office of Mental Health at Kingsboro Psychiatric Center. He can be contacted at [jkeen@cpan.org](mailto:jkeen@cpan.org).

- Code that is used in more than one script should be refactored into a module.

The fact that I was using the same code to manage *scriptslst* as I was for *slidelist* meant that I clearly had the makings of a module. But what functions were to go into that module?

I reread the relevant recipes in the *Perl Cookbook* and noticed that Tom and Nat used seen-hashes to derive several interesting relationships among two lists:

- The union of the two lists.
- The intersection of the two lists.
- Items unique to the first of two lists (what Tom and Nat call “simple difference”).
- The symmetric difference of the two lists: items found in one or the other of two lists, but not both.

In Listing 1, the list described by keys *%seen\_selections* in lines 33–37 was clearly a list of items unique to the first of two lists. In addition, I had used seen-hashes to derive two relationships not explicitly described in the *Cookbook*:

- Items unique to the second of two lists were found in *@unused*, derived between lines 42 and 50.
- The “left-to-right subset status” derived between lines 23 and 29.

Wouldn’t it be nice, I thought, to have a function that, when provided with two lists as inputs, returned all the most interesting relationships between the lists?

## The Birth of *List::Compare*

And thus, I had the inspiration for *List::Compare*. During the next month, when I wasn’t busy writing slides for my students, I was creating an object-oriented interface in which:

1. References to arrays holding two lists are passed to the constructor:

```
use List::Compare;
$lc = List::Compare->new(\@selections, \@sources);
```

2. The constructor’s initializer computes all of the most interesting relationships between the two lists (see Listing 2).
3. The results of those computations are stored in a hash that is blessed into the *List::Compare* object.
4. *List::Compare* methods simply return the various relationships:

```
@union      = $lc->get_union;
@intersection = $lc->get_intersection;
@lonly      = $lc->get_unique;
@ronly      = $lc->get_complement;
@lorronly   = $lc->get_symmetric_difference;
$LR         = $lc->is_lsubsetR;
```

(For additional interlist relationships not described here, see the *List::Compare* documentation.)

Using *List::Compare*, I can rewrite Listing 1, as shown in Listing 3. What once was a 60-line script is now only 41 lines. Moreover, should I need at this point additional relationships between the two lists (e.g., their symmetric difference), I could simply call the appropriate *List::Compare* method and get the result without additional computation.

## A More Mature *List::Compare*

I first showed *List::Compare* to other Perl hackers at the May 2002 meeting of Perl Seminar New York and its first CPAN version was uploaded the next month just before YAPC::NA::2002 in St. Louis.

```
@Al      = qw(abel abel baker camera delta edward fargo golfer);
@Bob     = qw(baker camera delta delta edward fargo golfer hilton);
@Carmen  = qw(fargo golfer hilton icon icon jerky kappa);
@Don     = qw(fargo icon jerky);
@Ed      = qw(fargo icon icon jerky);

$lc = List::Compare->new(\@Al, \@Bob, \@Carmen, \@Don, \@Ed);
@intersection = $lc->get_intersection;
```

### Example 1: Using *List::Compare* in multiple mode.

Since that time, I have worked on both its interface and internals in an attempt to allow potential users maximum flexibility and, where possible, speed improvements. Here are the most significant improvements:

**Accelerated Mode.** Shortly after *List::Compare*’s first presentation, Perl Seminar NY member Josh Rabinowitz argued that if a user wanted to compute only one comparison between two lists (e.g., just their intersection), *List::Compare* should not spend time computing any other relationships between the two lists.

In response, I decided to offer the user the option of an “accelerated” mode in which the user passes *-a* as the first argument to the constructor:

```
$lca = List::Compare->new('-a', \@selections, \@sources);
@intersection = $lca->get_intersection;
```

Internally, the constructor calls an initializer, which populates the object with references to the lists submitted as arguments rather than with the results of computations of set relationships. In the accelerated mode, it is the individual method called that does the computation, not the initializer.

Preliminary benchmarking showed that if a user indeed wanted to extract only one comparison between two lists, the accelerated mode was faster. However, it was no faster than the regular mode if the user wanted two comparisons, and the regular mode pulled ahead if the user wanted three or more comparisons.

**Multiple Mode.** Not long after *List::Compare*’s CPAN debut, I wondered: Why should a user be restricted to comparisons between just two lists? Why shouldn’t a user be able to compute, say, the intersection of three or more lists at a time? So I set out to develop a “multiple” mode in which a user would simply pass additional lists as arguments to the constructor; see Example 1.

Comparing three or more lists at a time, however, requires a more careful specification of certain comparisons. When calling the *get\_unique()* method, for example, I wanted the user to be able to get those items unique to, say, *@Carmen* and not just to the first list passed by reference to the constructor. Therefore, I designed that method’s interface so the user could specify the index position of the targeted list as an argument to the method. Since Perl starts counting at zero, *@Carmen*’s index position is “2” and a list of items unique to that array is generated like this:

```
@unique_Carmen = $lcm->get_unique(2);
```

Similarly, the items *not* found in *@Don* would be generated like so:

```
@complement_Carmen = $lcm->get_complement(3);
```

Enabling *List::Compare* to handle more than two lists also enables us to define two new relationships among the various lists: the complement of their intersection and the complement of their symmetric difference. But since that’s very verbose, I have simplified their names for the purpose of method calls and documentation into *nonintersection* and *shared*.



*List::Compare* defines the nonintersection of several lists as those found in any of the lists passed to the constructor that do not appear in all of the lists (i.e., all items except those found in the intersection of the lists):

```
@nonintersection = $lcm->get_nonintersection;
```

*List::Compare* defines items shared among several lists as those which appear in more than one of the lists passed to the constructor (i.e., all items except those found in their symmetric difference):

```
@shared = $lcm->get_shared;
```

---

*In the accelerated mode,  
it is the individual method  
called that does the computation,  
not the initializer*

---

At first, *List::Compare*'s *multiple* mode did not have an "accelerated" variant for faster calculation of just one relationship among three or more lists, but this limitation has been eliminated in the last year. To choose accelerated calculation, pass the *-a* option as the first argument to the constructor:

```
$lcm = List::Compare->new(
    '-a', \@Al, \@Bob, \@Carmen, \@Don, \@Ed
);
@intersection = $lcm->get_intersection;
```

## Returning Array References

Later in 2002, I had occasion to use *List::Compare* in some work on my day job for the New York State Office of Mental Health. I was using the *get\_union* method to return a list, but then had to immediately pass a reference to the array holding that list to another function.

```
@union = $lc->get_union;
some_other_function(@union);
```

Since having subroutines receive arguments by—and return results by—references is faster than by receiving/returning whole lists, why not allow *List::Compare* to return just a reference to a list? This led to a new set of *List::Compare\_ref* methods:

```
$unionref = $lc->get_union_ref;
some_other_function($unionref);
```

After this revision had been uploaded to CPAN, Glenn Maciag of Perl Seminar NY noted that if I had simply rewritten the *get\_union()* method to use Perl's *wantarray* function to examine the method's calling context, I could have dispensed with a separate *get\_union\_ref()* method:

```
@union = $lc->get_union;
$unionref = $lc->get_union; # NOT IMPLEMENTED
```

However, since this approach was slightly less self-documenting and since the *\_ref* methods were already out there on CPAN, I didn't implement his suggestion—though I may in the future.

## Four Membership Methods

In a subsequent revision, I gave *List::Compare* the ability to answer these questions:

**Given a string, determine to which of the lists passed to the constructor the string belongs.**

```
@memb_arr = $lcm->is_member_which('golfer');
```

The list returned by *is\_member\_which()* is a list of the indexes in the constructor's argument list in which *golfer* is found; i.e.:

```
( 0, 1, 2 )
```

**Given several strings, determine to which of the lists passed to the constructor the various strings belong.** Do this by passing to the *are\_members\_which()* method a reference to an array holding references to the various lists under examination. Get a hash of array references as the return value:

```
$memb_hash_ref = $lcm->are_members_which(
    [ qw| abel baker fargo hilton zebra | ]
);
```

"\$memb\_hash\_ref" will be:

```
{
    abel    => [ 0          ],
    baker   => [ 0, 1       ],
    fargo   => [ 0, 1, 2, 3, 4 ],
    hilton  => [ 1, 2       ],
    zebra   => [           ],
};
```

**Given a string, determine whether it can be found in any of the lists passed as arguments to the constructor.** Return 1 if a specified string can be found in any of the lists and 0 if it cannot:

```
$found = $lcm->is_member_any('abel');
```

In the example above, *\$found* will be *1* because *abel* is found in one or more of the lists passed as arguments to *new()*.

**Given several strings, determine if each such string can be found in any of the lists passed as arguments to the constructor.** Do this by passing to the *are\_members\_any()* method a reference to an array holding references to the various lists under examination. Get a hash reference as the return value where the value of each element is either *1* if the particular string can be found in any of the lists under examination and *0* if it cannot.

```
$memb_hash_ref = $lcm->are_members_any(
    [ qw| abel baker fargo hilton zebra | ]
);
```

"\$memb\_hash\_ref" will be:

```
{
    abel    => 1,
    baker   => 1,
```

```

use List::Compare::SeenHash;

my %seenAl = (
    abel    => 2,
    baker   => 1,
    camera  => 1,
    delta   => 1,
    edward  => 1,
    fargo    => 1,
    golfer   => 1,
);

my %seenBob = (
    baker   => 1,
    camera  => 1,
    delta   => 2,
    edward  => 1,
    fargo    => 1,
    golfer   => 1,
    hilton  => 1,
);

$lcsH = List::Compare::SeenHash->new(\%seenAl, \%seenBob);
@intersection = $lcsH->get_intersection;

```

**Example 2: Using List::Compare::SeenHash.**

```

    fargo    => 1,
    hilton   => 1,
    zebra    => 0,
};

```

## Passing Seen-Hashes to the Constructor

In the course of a Perl script, if a user has already created two or more lookup tables in the form of seen-hashes and needs to determine set relationships among the lists implied by those seen-hashes, it seemed to me that he or she should be able to pass references to those seen-hashes directly to the constructor. This is now possible; see Example 2.

`C<@intersection>` will contain the following:

```

qw( baker camera delta edward fargo golfer)

```

The constructor figures out for itself whether it has been passed arrays or seen-hashes.

## Returning Unsorted Lists

When I first wrote *List::Compare*, all comparison lists were sorted in ASCII-betical order by default. Since sorting imposes a cost in speed, a later revision enables a user who does not need a pre-sorted list to pass an *unsorted* option to the constructor:

```

$lC = List::Compare->new('-u', \@Llist, \@Rlist);

```

or

```

$lC = List::Compare->new('unsorted', \@Llist, \@Rlist);

```

I haven't benchmarked this approach, yet, but my impression is that the speed boost is largely marginal. It's there for you to use; TMTOWTDL.

## A Functional Interface

In the autumn of 2003, I began to wonder if an object-oriented interface to *List::Compare* was always necessary. What would I have to do to enable a user to pass references to several lists directly to a function such as *get\_union()* rather than first passing them to a constructor?

This is now possible with *List::Compare::Functional*:

```

use List::Compare::Functional qw( get_union get_complement );

# same 5 lists as above: @Al, @Bob, @Carmen, @Don, @Ed

@union = get_union( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );

```

Note that, as with many Perl modules that employ a functional interface rather than an object oriented one, the user must specifically import the function(s) he or she later wishes to call. (Import tag groups are available; see the documentation.)

Note also that with *List::Compare::Functional*, the first argument passed to the function is a reference to an array (anonymous or named), which, in turn holds a list of references to the lists under examination. This proved necessary to distinguish the arguments representing the lists under examination from particular lists that are passed to certain functions. For example, if we wish to find those items not found in *@Don*, we need to pass a second argument (also via reference) like this:

```

@complement_Don = get_complement(
    [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
    [ 3 ]
);

```

Since, on the inside, much of *List::Compare::Functional*'s code works like that of *List::Compare*'s accelerated mode, and since it incurs no overhead for object creation, it should provide faster results than *List::Compare*. But the interface to each *List::Compare::Functional* function is necessarily less elegant than that to *List::Compare*'s methods.

## Conclusion

*List::Compare* offers a variety of ways to compare two or more lists. The Perl code around which it is built is old, well-tested, and certainly not mine. It is, at its best, a sound implementation of a good interface. It is not rocket science.

It should be noted that, with the exception of *List::Compare*'s *get\_bag()* method (not discussed in this article), all comparisons conducted by *List::Compare* only ask whether a given string was seen in a given list at all. In general, *List::Compare* ignores how many times a string was seen in a given list. If you need to make decisions based on how many times a string was seen in a given list, don't use *List::Compare*; look elsewhere. You will find, however, many situations in which *List::Compare* or its *Functional* variants can save you from typing that *Perl Cookbook* code over again.

TPJ

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

### Listing 1

```

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

my ($raw, @selections, @sources);
open LIST, 'slidelist' or die "Can't open slidelist for reading: $!";
while ($raw = <LIST>) {
    next if ($raw =~ /\s$/ or $raw =~ /^#/); # no comments or blanks

```

```

    next unless ($raw =~ /\slide\.txt$/);
    chomp $raw;
    push(@selections, "texts/$raw");
}
close LIST or die "Cannot close slidelist: $!";

opendir DIR, 'texts' or die "Couldn't open texts directory: $!";
@sources = map {"texts/$_" } grep /\slide\.txt$/ readdir DIR;
closedir DIR;

my (%seen_selections, %seen_sources);
$seen_selections{$_}++ foreach @selections;
$seen_sources{$_}++ foreach @sources;

```

```

my $subset_status = 1;
foreach (@selections) {
    if (! exists $seen_sources{$_}) {
        $subset_status = 0;
        last;
    }
}

unless ($subset_status) {
    my (%selections_only);
    foreach (keys %seen_selections) {
        $selections_only{$_}++ if (! exists
$seen_sources{$_});
    }
    print "These files, though listed in 'slidelist', are not found in
'texts' directory.\n\n";
    print "   $_\n" foreach (keys %selections_only);
    print "\nEdit 'slidelist' as needed and re-run script.\n";
    exit (0);
}

my (%intersection, %difference);
foreach (keys %seen_selections) {
    $intersection{$_}++ if (exists $seen_sources{$_});
}
foreach (keys %seen_sources) {
    $difference{$_}++ unless (exists $intersection{$_});
}

my @unused = keys %difference;
open UNUSED, ">unused" or die "Could not open unused for writing: $!";
print UNUSED "Files currently unused:\n";
if (@unused) {
    print UNUSED "   $_\n" foreach (sort @unused);
    print "There are unused files; see 'unused'.\n";
} else {
    print UNUSED "   [None.]\n";
    print "There are no unused files.\n";
}
close UNUSED or die "Could not close unused: $!";

Listing 2

# Computation of Relationships between Two Lists in List::Compare's Regular
Mode
# Regular Mode => Compares only two lists; computes all relationships at
once;
# stores the results in a hash which is blessed into the List::Compare
object.
# Below: _init(), which is called by List::Compare's constructor and which
returns
# a reference to the hash which the constructor will bless.

sub _init {
    my $self = shift;
    my ($unsortflag, $refL, $refR) = @_;
    my (%data, %seenL, %seenR);
    my @bag = $unsortflag ? (@$refL, @$refR) : sort(@$refL, @$refR);

    my (%intersection, %union, %Lonly, %Ronly, %LorRonly);
    my $LsubsetR_status = my $RsubsetL_status = 1;
    my $LequivalentR_status = 0;

    foreach (@$refL) { $seenL{$_}++ }
    foreach (@$refR) { $seenR{$_}++ }

    foreach (keys %seenL) {
        $union{$_}++;
        if (exists $seenR{$_}) {
            $intersection{$_}++;
        } else {
            $Lonly{$_}++;
        }
    }

    foreach (keys %seenR) {
        $union{$_}++;
        $Ronly{$_}++ unless (exists $intersection{$_});
    }

    $LorRonly{$_}++ foreach ( (keys %Lonly), (keys %Ronly) );

    $LequivalentR_status = 1 if ( (keys %LorRonly) == 0);

    foreach (@$refL) {
        if (! exists $seenR{$_}) {
            $LsubsetR_status = 0;
            last;
        }
    }
}

```

```

}
foreach (@$refR) {
    if (! exists $seenL{$_}) {
        $RsubsetL_status = 0;
        last;
    }
}

$data{'seenL'} = \%seenL;
$data{'seenR'} = \%seenR;
$data{'intersection'} = $unsortflag ? [ keys %intersection ]

: [ sort keys %intersection ];
$data{'union'} = $unsortflag ? [ keys %union ]

: [ sort keys %union ];
$data{'unique'} = $unsortflag ? [ keys %Lonly ]

: [ sort keys %Lonly ];
$data{'complement'} = $unsortflag ? [ keys %Ronly ]

: [ sort keys %Ronly ];
$data{'symmetric_difference'} = $unsortflag ? [ keys %LorRonly ]

: [ sort keys %LorRonly ];
$data{'LsubsetR_status'} = $LsubsetR_status;
$data{'RsubsetL_status'} = $RsubsetL_status;
$data{'LequivalentR_status'} = $LequivalentR_status;
$data{'bag'} = \@bag;
return \%data;
}

```

## Listing 3

```

#!/usr/bin/perl
use strict;
use warnings;
use List::Compare;

my ($raw, @selections, @sources);
open LIST, 'slidelist' or die "Can't open slidelist for reading: $!";
while ($raw = <LIST>) {
    next if ($raw =~ /\s+$/ or $raw =~ /^#/); # no comments or blanks
    next unless ($raw =~ /\.slide\.txt$/);
    chomp $raw;
    push(@selections, "texts/$raw");
}
close LIST or die "Cannot close slidelist: $!";

opendir DIR, 'texts' or die "Couldn't open texts directory: $!";
@sources = map {"texts/$_"} grep { /\.slide\.txt$/ } readdir DIR;
closedir DIR;

my $lc = List::Compare->new(\@selections, \@sources);
my $LR = $lc->is_LsubsetR;
unless ($LR) {
    my @slidelist_only = $lc->get_unique();
    print "These files, though listed in 'slidelist', are not found in
'texts' directory.\n\n";
    print "   $_\n" foreach (@slidelist_only);
    print "\n";
    print "Edit 'slidelist' as needed and re-run script.\n";
    exit (0);
}

my @unused = $lc->get_complement;
open(UNUSED, ">unused") or die "Could not open unused for writing: $!";
print UNUSED "Files currently unused:\n";
if (scalar(@unused)) {
    print UNUSED "   $_\n" foreach (sort @unused);
    print "There are unused files; see 'texts/unused'.\n";
} else {
    print UNUSED "   [None.]\n";
    print "There are no unused files.\n";
}
close(UNUSED) or die "Could not close unused: $!";

```

TPJ



# Automating Distributions with *scriptdist*

*brian d foy*

Last year (*TPJ*, March 2003), I showed how to create a Perl distribution for a script, but I did not give any way to actually create those distributions other than doing a lot of typing by hand. Since then, while tidying up my own scripts, I created a program I call “*scriptdist*” to automate all of the things I was doing manually. I also want to make it as easy as possible for other people to create distributions so they can pass around scripts that are easy to test and easy to install. Once I automate the repetitive, boring parts, I have more time to concentrate on the rest of the script lifecycle.

My program is only one of a few scripts that try to do this. The *h2xs* program, designed to turn C header files into the appropriate Perl glue, can make module and script distributions. It has several command-line options to turn various things on or off, but other than that, I am stuck with the files it creates and what it puts in those files. On the other hand, the *mmm* (My Module Maker) program by Mark Fowler can create distribution files from external templates. The *ExtUtils::ModuleMaker* module by R. Geofrey Avery acts as a replacement for *h2xs* and comes with a *mod-ulemaker* script to interactively create a distribution.

I decided to create something new after several years of dealing with *h2xs*, which creates a structure that I mostly discard and stub files that I do not use. I have to interact with the *module-maker* script and that is too much work. The *mmm* script is nice, but it does not go as far as I want it to go, and I have to tell it to include things, which is too much work, too. I want things to happen by default, and the less I have to type on the command line, the better it works out. Everyone seems to have a difference in preference for this process, so no option is going to please everyone, but with enough options, most people should find one that works for them.

I have been working on and adding to this script every time I use it, and Soren Anderson has been a faithful tester and patcher

---

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

who got a lot of the features to actually work on more than just my computer. By the time you read this, we may have done even more. The *scriptdist* distribution is on CPAN (<http://search.cpan.org/>) and SourceForge (<http://sourceforge.net/projects/brian-d-foy/>). I even used this script to build its own distribution.

My program automates what I have usually done to create a distribution, all of which is repetitive work—the realm of computers and scripts, not humans. Given a script file by itself, I build a distribution around it. So far, *scriptdist* expects the script file to actually exist, but I want to fix that so it will create the script from a template as well.

Suppose I want to build a distribution for my fictitious Perl script named “mimi.” I simply give the script name to *scriptdist* as its sole argument. The script figures the rest out on its own (but then, I programmed it to think like me).

```
% scriptdist mimi
```

As I write this, that is all there is to it. *scriptdist* then does its thing:

```
brian$ scriptdist mimi
Processing mimi...
Making directory mimi.d...
Making directory mimi.d/t...
Checking for file [.cvsignore]... Adding file [.cvsignore]..
Checking for file [.releaserc]... Adding file [.releaserc]..
Checking for file [Changes]... Adding file [Changes]..
Checking for file [MANIFEST.SKIP]... Adding file [MANIFEST.SKIP]..
Checking for file [Makefile.PL]... Adding file [Makefile.PL]..
Checking for file [t/compile.t]... Adding file [t/compile.t]..
Checking for file [t/pod.t]... Adding file [t/pod.t]..
Checking for file [t/prereq.t]... Adding file [t/prereq.t]..
Checking for file [t/test_manifest]... Adding file [t/test_manifest]..
Adding [mimi]...
Opening input [mimi] for output [mimi.d/mimi]
Copied [mimi] with 0 replacements
Creating MANIFEST...
```

---

Remember to commit this directory to your source control system.



In fact, why not do that right now? Remember, 'cvs import' works from within a directory, not above it.

The script first creates a directory to hold all of the files. It uses the script name with an appended “.d,” but stops if that directory already exists so it will not overwrite what I may have already

---

*My program automates what I  
have usually done to create a  
distribution, all of which is  
repetitive work*

---

done. The particular directory name turns out to be only temporary, though, as I will show later. It also creates the *t* directory for test scripts.

Once *scriptdist* creates the directory structure, it checks for template files in the `~/.scriptdistrc` directory in my home directory. If it finds a file, it copies it to the new distribution directory. It does not matter what the file name is—the entire directory structure of `~/.scriptdistrc`, other than CVS and subversion files, ends up in the distribution directory, preserving the structure. This way, I can set up the distribution in my favorite way, regardless of what the script wants to do. For each template file, *scriptdist* replaces the strings `SCRIPTDIST_SCRIPT` with the name of the script and `SCRIPTDIST_VERSION` with 0.10. Eventually, I want to add more such replacements and to make their values configurable, but these are good enough for me so far.

After *scriptdist* copies the template files, it goes through an internal list of files that I want to put in the distribution, although it skips files that already exist from the previous step. This way, I only have to create new templates if I do not like the default versions. Inside *scriptdist*, I hard coded templates for each of these files just the way that I like them (but probably not the way anyone else does, hence the templating mechanism). Alternately, if someone likes *scriptdist* but not the internal templates, he or she is free to change them so that it is still a single file.

I automatically add a bunch of test files, including: `compile.t`, which ensures the script compiles so I can bail out of the rest of the tests if the script needs fixing; a test for the pod that forces me to write the documentation alongside the code; and a test to ensure that the prerequisite modules for the script show up in `Makefile.PL`. These last two tests only run if I have installed `Test::Pod` (created by me originally, but greatly improved and now developed by Andy Lester) or `Test::Prereq` (also by me).

Next, *scriptdist* copies the script into the directory. Originally, I created *scriptdist* to create distributions for scripts that I already had. I would like *scriptdist* to automatically create the script (perhaps with a template). The technological fix for this is easy, but I have to work on the script template a bit, and only that and laziness has kept this feature out of *scriptdist*.

Finally, *scriptdist* creates the MANIFEST file by calling `ExtUtils::Manifest::mkmanifest`. The `ExtUtils::*` modules have sev-

eral functions I can use to work with distributions, and browsing through a Makefile shows a lot of them in action.

At the end of the run, *scriptdist* reminds me to import the new directory into my source control system, and since I often forget which directory I am in when I do this, *scriptdist* reminds me to change into the script directory before I import the distribution into CVS. When I import the directory into CVS, I choose the name for the (CVS, not Perl) module. The current directory name makes no difference because I need to check out the (again, CVS, not Perl) module to continue to work on my script distribution. Later, I would like to automate these steps as well.

I still have a short list of things I need to automate, although I am waiting until I get out of the Army (and by the time you read this, that should only be a couple of weeks) to make any more changes. I would like *scriptdist* to automatically create the `PRE-REQ_PM` value for `WriteMakefile()` in `Makefile.PL`, and I am planning to do that along with some changes to `Test::Prereq`. I would like to add command-line switches to control the behavior of *scriptdist*, so I can choose between features such as using the usual `Makefile.PL` or `Module::Build's Build.PL` method. I would also like to add support for `Module::Release`, the tool created out of my “release” program to automate the other side of the distribution lifecycle.

This program is a work in progress, but it does most of my script distribution set-up work, and it works for a few other people as well. I have a lot of things I would like to add to it. Most of the features do things the way that I like, but I want to make that more flexible. I appreciate any comments and suggestions *TPJ* readers may have.

*TPJ*

**Fame & Fortune  
Await You!**

**Become a  
*TPJ*  
author!**

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

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



# Eight Million Ways to *die*

Randal L. Schwartz

What is that old saying? “The best-laid plans of mice and men...” And like that old saying, sometimes your programs don’t go the way you expect.

For example, a user might not enter a number between one and five, even though your prompt carefully suggests that they do. Or maybe the file you expected to create in that directory can’t be created. Or the database connection fails to connect (and was it because the system was down or because you were given a bad password?). Or the module you needed for a particular part of the application failed to load (maybe because it was never installed).

Usually, when things go wrong, you want to know about it and do something in your program in response. For example, consider a program that updates a data file, incrementing a value:

```
open OLD, "counter";
open NEW, ">counter.tmp";
print NEW <OLD> + 1;
close OLD;
close NEW;
rename "counter.tmp", "counter";
```

Note that we have six lines of code, any of which could fail. Let’s take the simplest ones first. If the first *open* fails, we’ll be using a closed filehandle in the third line, which will look like a 0 to the *add 1* operation, and we’ll get a “1” value in the final file.

Now, this might actually make sense for this application: The first invocation of the program yields a 1 value. But, if we have warnings enabled, we’ll get a warning when we attempt to read from a closed filehandle on the third line because that’s generally considered bad style, if not a more serious error. We could notice the return value from that *open* and rewrite the code like this:

```
my $old_value = 0;
if (open OLD, "counter") {
    $old_value = <OLD>;
close OLD;
}
open NEW, ">counter.tmp";
print NEW $old + 1;
close NEW;
rename "counter.tmp", "counter";
```

---

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

and this does solve the extraneous warning. We now take an alternate execution path if the file is not initially present, thus, nicely sidestepping the warning.

But what if the *open* failure is from something more serious than “file not found”? My UNIX *open(2)* manpage lists about a dozen different reasons for a failure, including esoteric things such as “a symbolic link loops back onto itself.” How do we distinguish those?

The error variable *\$!* starts to look pretty interesting. For example, we can distinguish between *good*, *file not found*, and *everything else* with a three-way branch:

```
if (open OLD, "counter") {
    # good
} elsif ($! =~ /file.*not found/) {
    # not found, default to 0
} else {
    # everything else
}
```

Because I’m using *\$!* in a string context, I get to see the stringish error message. This is fairly operating-system specific, but if you’re not trying to be portable across a wide variety of systems, you can get away with such matches.

Note that I’m testing *\$!* only when I’ve had a failure and immediately afterward. This is the only time I can be sure that there’s really an error in there because, although an operating-system request failure sets *\$!*, nothing normally resets it. Thus, this code is broken:

```
## BAD CODE DO NOT USE
open OLD, "counter";
if ($! =~ /file.*not found/) { # file not found
    ..
} elsif ($!) { # other error
    ..
} else { # everything OK
    ..
}
```

We’re not necessarily testing the failed *open* call here—any prior failed call might give us a false positive.

But now, we need to decide what to do if we get that unexpected error. There’s an old joke among programmers: “Don’t test for anything you aren’t willing to handle,” but we can no longer plead ignorance here. The most common solution is to abort the entire program and let the sysadmin on duty watch take care of it, and that’s easy with *die*. Let’s redesign our program so that a

missing counter file is considered a bad, bad thing, and abort the program if that first *open* fails:

```
unless (open OLD, "counter") {  
    die "Cannot open counter";  
}
```

In this case, a false return value (for any reason) from *open* triggers the *die*, which aborts our program immediately. The error message is sent to STDERR (rather than STDOUT) to ensure that the message is not lost in a typical redirection to a file or pipe. In addition, the filename and the line number are automatically appended to the message, unless the message string ends in a new-line. This helps us find the source of the *die* among many modules and files.

Note that the error message contains the attempted operation as well. Again, this helps the debugging a bit, other than the cryptic “died at line 14” from the default message. This is especially handy when the filename for the operation might have come from another source:

```
chomp(my $filename = <SOMEBOTHERFILE>);  
unless (open OLD, $filename) {  
    die "Cannot open $filename";  
}
```

Before making it a habit to include such information in my *die* messages, I was occasionally confused about why my program was failing because I had presumed that a variable contained something other than what it did. Always echo the input parameters in the error message!

Another thing to include is the *\$!* I mentioned earlier. That can help us figure out the kind of failure:

```
unless (open OLD, "counter") {  
    die "Cannot open counter: $!";  
}
```

And finally, this is too much typing. The *or* operator executes its right operand only when the left operand is False, so we can shorten this to the traditional:

```
open OLD, "counter"  
or die "Cannot open counter: $!";
```

So, to fully instrument my original program, I could add *or die* to each of the steps that might fail:

```
open OLD, "counter" or die;  
open NEW, ">counter.tmp" or die;  
print NEW <OLD> + 1 or die;  
close OLD or die;  
close NEW or die;  
rename "counter.tmp", "counter" or die;
```

Wait a second? Why am I checking the return value from *print*? And from *close*? Those can't fail, can they? Certainly they can, although this is probably one of the few times you'll see any program that tests for them. The *print* can fail if the filehandle is closed or if there's an I/O error, such as a disk being full. And the *close* can fail if the filehandle is closed or if the final buffer being flushed at the time of the close couldn't be written (again, typically from a full disk).

This seems like a lot of typing. Can we reduce this? Sure, with the *Fatal* module, part of the Perl core for recent versions of Perl. We simply list the subroutines that should have an automatic *or die* added, and away we go:

```
use Fatal qw(open close rename);  
open OLD, "counter";  
open NEW, ">counter.tmp";  
print NEW <OLD> + 1;  
close OLD;  
close NEW;  
rename "counter.tmp", "counter";
```

Now we have (nearly) the same program with a lot less typing. The downside to this approach is that we don't really get to say

---

*My UNIX open(2) manpage lists about a dozen different reasons for a failure, including esoteric things like “a symbolic link loops back onto itself”*

---

what the error message is, other than the default *Died*. To get a bit more control, I could add *:void* to that argument list, and then any of those calls that have explicit testing for the return value will no longer be fatal:

```
use Fatal qw(:void open close rename);  
open OLD, "counter" or warn "old value unavailable, presuming 0\n";  
open NEW, ">counter.tmp";  
print NEW <OLD> + 1;  
close OLD or "ignore";  
close NEW;  
rename "counter.tmp", "counter";
```

Why didn't I list *print* here? Well, *Fatal* uses some magic behind the scenes, and *print* resists this magic. Oops. We'll have to do that one by hand.

The *die* operator is fatal to the program unless it is enclosed within an *eval* block (or by a *\_\_DIE\_\_* handler, but I digress). Once safely within the *eval* block, any *die* aborts the block, not the program. Immediately following the block, we check the *\$@* variable, which is guaranteed to be empty if the block executed to completion (or the text message that would have been sent to STDERR if we would have otherwise aborted). Time for an example:

```
use Fatal qw(:void open close rename);  
for my $file (qw(counter1 counter2 counter3)) {  
    eval {  
        open OLD, "$file" or warn "old value unavailable, presuming 0\n";  
        open NEW, ">$file.tmp";  
        print NEW <OLD> + 1;  
        close OLD or "ignore";  
        close NEW;  
        rename "$file.tmp", "$file"  
    };  
    if ($@) {  
        print "ignored error on $file (continuin): $@";  
    }  
}
```

Here, I've put the previous code inside the *eval* block, using *\$file* in place of the literal filenames. If any of the steps within the *eval* block fail, we skip immediately to the end of the block. The message ends up in *\$@*. If the message is present, we note it on *STDOUT*. Whether there was an error or not, we're continuing the loop.

Now, suppose we conclude that any permission-denied message inside the *eval* block is likely to mean we're not going to get much further on the rest of the program. We can take different actions based on the value within *\$@*. For example:

```
use Fatal qw(:void open close rename);
for my $file (qw(counter1 counter2 counter3)) {
    eval {
        open OLD, "$file" or warn "old value unavailable, presuming 0\n";
        open NEW, ">$file.tmp";
        print NEW <OLD> + 1;
        close OLD or "ignore";
        close NEW;
        rename "$file.tmp", "$file"
    };
    if ($@ =~ /permission denied/i) {
        die $@; # rethrow $@
    } elsif ($@) {
        print "ignored error on $file (continuing): $@";
    }
}
```

If the message in *\$@* after the loop matches permission denied, we *rethrow* the error. In this case, there's no outer *eval* block, so the program aborts. However, had there been an outer *eval* block, we'd simply pop out one more level. In turn, that outer block could handle the error or *rethrow* it again to the next level (if any), and so on.

Matching the specific text of error messages can be a bit problematic, especially when you have to change the text for internationalization of your program. Fortunately, modern versions of Perl permit the *die* parameter to be an object, not just a text message. When an object value is thrown with *die*, the *\$@* value contains that object as well. Not only does this let us pass structured data up the exception-handling logic, we can also create hierarchies of error classifications to quickly sort entire groups of errors apart.

The best framework I've seen for creating such error categories is *Exception::Class* found in CPAN. Let's restructure our program to use exception objects rather than text testing (see Listing 1).

The first lines (invoking *Exception::Class* with parameters) create a hierarchy of classes, starting with my *E* class (selected be-

cause the name is short). From *E*, I break errors into two categories: user-related errors and file-related errors. File-related errors are further categorized into various file operations. The *isa* parameter defines the base class for each derived class, permitting the use of normal *isa* tests for quick categorization.

Now, inside the *eval*, instead of a simple *die*, I use the *throw* method of an appropriate error class, with a specific error mes-

## Always echo the input parameters in the error message

sage. I won't need to include *\$!* here because I'll know that every error in the *E::File* category was system-call related, and I can put that just once in the error handler.

Finally, the error-handling logic just past the end of the *eval* block is also changed. If *\$@* is an object derived from my *E*, then I sort out what kind of error it might be. Note that I've chosen to handle all *E::Create* errors as relatively "fatal" to my loop (although they might in turn be caught by some outer *eval* block not shown here). User errors are distinguished from *E::File* errors, with the latter displaying the *\$!* value automatically. Also note that any legacy errors (from an ordinary *die* or maybe a reference or object not within my hierarchy of classes) simply get *rethrown* as well.

This framework is actually quite flexible, permitting additional structured attributes to be carried along in the error object, as well as having objects inherit from multiple class hierarchies to distinguish multiple traits (file versus database, fatal versus recoverable, and so on). If you're building a complex application, you should definitely look into using *Exception::Class* or something similar. Until next time, enjoy!

TPJ

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

### Listing 1

```
use Exception::Class (
    E => { description => "my base error class" },
    E::User => { description => "user-related errors", isa => qw(E) },
    E::File => { description => "file-related errors", isa => qw(E) },
    E::Open => { description => "cannot open", isa => qw(E::File) },
    E::Create => { description => "cannot create", isa => qw(E::File) },
    E::Rename => { description => "cannot rename", isa => qw(E::File) },
    E::IO => { description => "other IO", isa => qw(E::File) },
);
for my $name (@ARGV) {
    eval {
        $name =~ /^[\w+$/ or E::User->throw("bad file name for $name");
        open IN, $name or E::Open->throw("reading $name");
        open OUT, ">$name.tmp" or E::Create->throw("creating $name.tmp");
        print OUT <IN> + 1 or E::IO->throw("writing $name.tmp");
        close IN or E::IO->throw("closing $name");
        close OUT or E::IO->throw("closing $name.tmp");
        rename "$name.tmp", $name or E::Rename->throw("renaming $name.tmp to
```

```
$name");
    };
    if (UNIVERSAL::isa($@, "E")) { # an object error from my tree
        if ($@->isa("E::User")) {
            warn "Pilot error: $@"; # warn and continue
        } elsif ($@->isa("E::Create")) {
            $@->rethrow; # same as die $@
        } elsif ($@->isa("E::File")) { # other IO errors
            warn "File error: $@: $!";
        } else {
            warn "Uncategorized error: $@"; # warn and continue
        }
    } elsif ($@) { # a legacy die error
        die $@; # abort (possibly caught by outer eval)
    } # else everything went ok
}
```

TPJ





# Perl Medic: Optimizing Legacy Code

Jack J. Woehr

The subtitle of *Perl Medic*, “Optimizing Legacy Code,” sounds like bizspeak. A more nerdlily description of this book might be, “Maintaining Perl code whether it’s written by you or others and whether or not it was designed to be maintained.” Apropos the latter, author Peter J. Scott is cheerfully free to suggest the classic solution—a total rewrite—where appropriate. That, by itself, would have made a very short book and an even shorter review. Luckily for us, there are many more solutions and development patterns to discuss regarding Perl maintenance. *Perl Medic*’s target audience begins at the level of intermediate Perl programmers who have been “working with Perl long enough to have heard terms like scalar, array, and hash,” but reaches well beyond that entry level.

In many cases, the patient, a sick or dying Perl corpus, can be saved. One half of the value in *Perl Medic* is in the diagnostic and remedial techniques Scott provides. The residual value of the book are found in the practices that make your Perl code more maintainable. Maintainability strategies for Perl code have sometimes been obscure, even to experienced Perl programmers. It’s been a hair-raising roller-coaster ride watching the language develop. The degree of cult participation traditionally necessary to stay on top of trends that could unexpectedly invalidate a seemingly reasonable application lifecycle taxes the patience of working developers. Scott offers the reader a compendium of what one might call “simplest-best” practices that are easily followed.

In the first four chapters, after introducing his subject and offering a few practical tips for achieving handover in some other fashion than having the code dumped onto your disk, Scott starts off reasonably enough by examining the use of Perl warnings. He continues with forensic techniques for evaluating the code and your tasks in taking up the code. Then testing is introduced, right where it should be, at the point before you start modifying code. Next comes the actual labor of rewriting code, beginning from style and variable names—in no other language than FortH do variable names seem to matter so much as in Perl—and concluding with antipatterns and the ongoing evolution of the code.

In the fifth and sixth chapters, the book moves on to issues of discipline, cognition, and semantics (or so I would characterize it) with an overarching theme of knowing why you are coding the way you are coding, why you are following certain patterns, and how to recognize code that follows blindly misunderstood or only partially grasped patterns. Then there’s a complete chapter on upgrading, covering Perl 4 and 12 earlier releases of Perl 5 to Perl 5.8.3, the latter being the stable version used in the book, followed by chapters on CPAN and the use of modules—other people’s and your own.

Jack J. Woehr is an independent consultant and team mentor practicing in Colorado. He can be contacted at <http://www.softwoehr.com>.

## *Perl Medic: Optimizing Legacy Code*

Peter J. Scott

Addison-Wesley, 2004

336 pp, \$34.99

ISBN 0-201-79526-4

In Chapter 9, the book returns to forensics: static analysis, superfluous code, inefficient code, and debugging. Chapter 10 deals with robustness and Chapter 11 is a case study. There follows a brief wrap-up, which concludes with the sage (albeit utterly self-evident) warning that, “One day your code, too, will be inherited by someone else.”

This is really a working programmer’s book. Section 4.10 casts an interesting light on this point. In it, the author introduces a pithy and sometimes witty several-pages-long list of steps and considerations in program evolution, after begging off from expanding on each of these points on grounds of time and space. As they say in the East, “The good horse runs at the shadow of the whip.” While Scott wriggles himself out from under the onus of spelling everything out, he concomitantly liberates the reader from having to slog through a bog of minutiae. It’s the sort of bullet-list help a busy senior developer in the next cubicle might e-mail you in response to a very broad and general question. The subtext in both situations is, “If you’re good enough to do this sort of work, this is all the help you need.” Serendipity!

*Perl Medic* is a cute little book with the flavor of a 1980’s programming language tome. Each chapter is introduced by a relevant (or occasionally irrelevant) quote and a cartoon à la Leo Brodie. Some of the quotations are rather apt: My favorite is Chapter 2’s quote from Henrik Ibsen about how one’s outlook may be haunted by the ghosts of “dead ideas and lifeless old beliefs,” a concept many team programmers will greet with a sigh of recognition. Scott has a sharp eye for fluff and ambiguity. He communicates his insights in a “teach a man to fish” fashion that is equally rewarding to the reader, whether that reader is immersed in a chapter or browsing a paragraph for tips. The writing is colloquial and intimate and exhibits very little dependency between chapters, making this an open-anywhere book. The publication production quality is high, and the book is gratifyingly well indexed.

The book’s web site is <http://www.perlmedic.com/>, where you can download the files that accompany the book after you answer a question about the book. Or, if you’re tired of reading, you can click on links for the author’s consulting services.

TPJ

---

# Source Code Appendix

---

## Craig Riter “Encrypting Web Pages on a Server”

### Listing 1

```
## get the Name of the host for the server and the script name
## will be used when constructing the full url for this script
my ($HOST) = $ENV{SERVER_NAME} =~ /(.*)/s; # untaint
my ($SCRIPT) = $ENV{SCRIPT_NAME} =~ /(.*)/s; # untaint

## check to see that this script is on a secure link
if ( $ENV{HTTPS} ne 'on' ) {

    print "<strong>ERROR: you should access this over a secure link.</strong><p>";
    print " Maybe you want <a href=\"https://\".$HOST.$SCRIPT.\">https://\".$HOST.$SCRIPT."</a>";

    exit;
}
```

### Listing 2

```
my ($passphrase) = $q->param('pwd') =~ /(.*)/s;
## make sure there are no extra slashes
my ($filename) = $q->param('f') =~ /[^\|\\]*/s;
my ($data) = $q->param('data') =~ /(.*)/s;
```

### Listing 3

```
## determine what button was clicked, and what action should be performed

## get submit button values for state control
my ($decrypt) = $q->param('decrypt') =~ /(.*)/s;
my ($encrypt) = $q->param('encrypt') =~ /(.*)/s;
my ($pathinfo) = $ENV{'PATH_INFO'} =~ /(.*)/s;

## the decrypt button was clicked and a filename has been specified
if ($decrypt ne '' && $filename) {
    &PerformDecrypt_OpenPGP($passphrase, $filename );
}
## the PATH_INFO is set with the name of the file to decrypt
elsif ($pathinfo ne '') {
    ## populate the file (f) field since that is what is expected
    ## CGI.pm will populate the field in the form
    $q->param('f',substr($pathinfo,1)) ;

    ## return form to request the password from the user
    ## param = 1, to have a full HTML page returned
    &ReturnDecryptForm( 1 );
}
## encrypt button was clicked
elsif ( $encrypt ne '' ) {
    &PerformEncrypt_OpenPGP($passphrase, $filename, $data);
}
## page requested for the first time
else {
    &ReturnInputForms();
}

exit;
```

## James E. Keenan “Determining List Relationships with *List::Compare*”

### Listing 1

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

my ($raw, @selections, @sources);
open LIST, 'slidelist' or die "Can't open slidelist for reading: $!";
while ($raw = <LIST>) {
    next if ($raw =~ /^s$/ or $raw =~ /^#/); # no comments or blanks
    next unless ($raw =~ /\.slide\.txt$/);
    chomp $raw;
    push(@selections, "texts/$raw");
}
close LIST or die "Cannot close slidelist: $!";

opendir DIR, 'texts' or die "Couldn't open texts directory: $!";
@sources = map {"texts/$_" } grep {/\.slide\.txt$/} readdir DIR;
closedir DIR;

my (%seen_selections, %seen_sources);
$seen_selections{$_}++ foreach @selections;
```

```

$seen_sources{$_}++    foreach @sources;

my $subset_status = 1;
foreach (@selections) {
    if (! exists $seen_sources{$_}) {
        $subset_status = 0;
        last;
    }
}

unless ($subset_status) {
    my (%selections_only);
    foreach (keys %seen_selections) {
        $selections_only{$_}++ if (! exists $seen_sources{$_});
    }
    print "These files, though listed in 'slidelist', are not found in 'texts' directory.\n\n";
    print "    $_\n" foreach (keys %selections_only);
    print "\nEdit 'slidelist' as needed and re-run script.\n";
    exit (0);
}

my (%intersection, %difference);
foreach (keys %seen_selections) {
    $intersection{$_}++ if (exists $seen_sources{$_});
}
foreach (keys %seen_sources) {
    $difference{$_}++ unless (exists $intersection{$_});
}

my @unused = keys %difference;
open UNUSED, ">unused" or die "Could not open unused for writing: $!";
print UNUSED "Files currently unused:\n";
if (@unused) {
    print UNUSED "    $_\n" foreach (sort @unused);
    print "There are unused files; see 'unused'.\n";
} else {
    print UNUSED "    [None.]\n";
    print "There are no unused files.\n";
}
close UNUSED or die "Could not close unused: $!";

```

## Listing 2

# Computation of Relationships between Two Lists in List::Compare's Regular Mode  
 # Regular Mode => Compares only two lists; computes all relationships at once;  
 # stores the results in a hash which is blessed into the List::Compare object.  
 # Below: \_init(), which is called by List::Compare's constructor and which returns  
 # a reference to the hash which the constructor will bless.

```

sub _init {
    my $self = shift;
    my ($unsortflag, $refL, $refR) = @_;
    my (%data, %seenL, %seenR);
    my @bag = $unsortflag ? (@$refL, @$refR) : sort(@$refL, @$refR);

    my (%intersection, %union, %Lonly, %Ronly, %LorRonly);
    my $LsubsetR_status = my $RsubsetL_status = 1;
    my $LequivalentR_status = 0;

    foreach (@$refL) { $seenL{$_}++ }
    foreach (@$refR) { $seenR{$_}++ }

    foreach (keys %seenL) {
        $union{$_}++;
        if (exists $seenR{$_}) {
            $intersection{$_}++;
        } else {
            $Lonly{$_}++;
        }
    }

    foreach (keys %seenR) {
        $union{$_}++;
        $Ronly{$_}++ unless (exists $intersection{$_});
    }

    $LorRonly{$_}++ foreach ( (keys %Lonly), (keys %Ronly) );

    $LequivalentR_status = 1 if ( (keys %LorRonly) == 0);

    foreach (@$refL) {
        if (! exists $seenR{$_}) {
            $LsubsetR_status = 0;
            last;
        }
    }
    foreach (@$refR) {
        if (! exists $seenL{$_}) {
            $RsubsetL_status = 0;
            last;
        }
    }
}

```

```

    }

    $data{'seenL'}      = \%seenL;
    $data{'seenR'}      = \%seenR;
    $data{'intersection'} = $unsortflag ? [ keys %intersection ]

    : [ sort keys %intersection ];
    $data{'union'}      = $unsortflag ? [ keys %union ]

    : [ sort keys %union ];
    $data{'unique'}     = $unsortflag ? [ keys %Lonly ]

    : [ sort keys %Lonly ];
    $data{'complement'} = $unsortflag ? [ keys %Ronly ]

    : [ sort keys %Ronly ];
    $data{'symmetric_difference'} = $unsortflag ? [ keys %LorRonly ]

    : [ sort keys %LorRonly ];
    $data{'LsubsetR_status'} = $LsubsetR_status;
    $data{'RsubsetL_status'} = $RsubsetL_status;
    $data{'LequivalentR_status'} = $LequivalentR_status;
    $data{'bag'}            = \@bag;
    return \%data;
}

```

### Listing 3

```

#!/usr/bin/perl
use strict;
use warnings;
use List::Compare;

my ($raw, @selections, @sources);
open LIST, 'slidelist' or die "Can't open slidelist for reading: $!";
while ($raw = <LIST>) {
    next if ($raw =~ /^$/ or $raw =~ /^#/); # no comments or blanks
    next unless ($raw =~ /\.slide\.txt$/);
    chomp $raw;
    push(@selections, "texts/$raw");
}
close LIST or die "Cannot close slidelist: $!";

opendir DIR, 'texts' or die "Couldn't open texts directory: $!";
@sources = map {"texts/$_" } grep { /\.slide\.txt$/ } readdir DIR;
closedir DIR;

my $lc = List::Compare->new(\@selections, \@sources);
my $LR = $lc->is_LsubsetR;
unless ($LR) {
    my @slidelist_only = $lc->get_unique();
    print "These files, though listed in 'slidelist', are not found in 'texts' directory.\n\n";
    print "  $_\n" foreach (@slidelist_only);
    print "\n";
    print "Edit 'slidelist' as needed and re-run script.\n";
    exit (0);
}

my @unused = $lc->get_complement;
open(UNUSED, ">unused") or die "Could not open unused for writing: $!";
print UNUSED "Files currently unused:\n";
if (scalar(@unused)) {
    print UNUSED "  $_\n" foreach (sort @unused);
    print "There are unused files; see 'texts/unused'.\n";
} else {
    print UNUSED "  [None.]\n";
    print "There are no unused files.\n";
}
close(UNUSED) or die "Could not close unused: $!";

```

## Randal L. Schwartz “Eight Million Ways to die”

### Listing 1

```

use Exception::Class (
    E => { description => "my base error class" },
    E::User => { description => "user-related errors", isa => qw(E) },
    E::File => { description => "file-related errors", isa => qw(E) },
    E::Open => { description => "cannot open", isa => qw(E::File) },
    E::Create => { description => "cannot create", isa => qw(E::File) },
    E::Rename => { description => "cannot rename", isa => qw(E::File) },
    E::IO => { description => "other IO", isa => qw(E::File) },
);
for my $name (@ARGV) {
    eval {
        $name =~ /^w+$/ or E::User->throw("bad file name for $name");
        open IN, $name or E::Open->throw("reading $name");
        open OUT, ">$name.tmp" or E::Create->throw("creating $name.tmp");
    };
}

```



---

```

print OUT <IN> + 1 or E::IO->throw("writing $name.tmp");
close IN or E::IO->throw("closing $name");
close OUT or E::IO->throw("closing $name.tmp");
rename "$name.tmp", $name or E::Rename->throw("renaming $name.tmp to $name");
};
if (UNIVERSAL::isa($@, "E")) { # an object error from my tree
  if ($@->isa("E::User")) {
    warn "Pilot error: $@"; # warn and continue
  } elsif ($@->isa("E::Create")) {
    $@->rethrow; # same as die $@
  } elsif ($@->isa("E::File")) { # other IO errors
    warn "File error: $@: $!";
  } else {
    warn "Uncategorized error: $@"; # warn and continue
  }
} elsif ($@) { # a legacy die error
  die $@; # abort (possibly caught by outer eval
} # else everything went ok
}

```

***TPJ***