

The Perl Journal

Programming Graphical Applications with Gtk2-Perl, Part 2

Gavin Brown • 3

Finding Duplicate Files

Julius C. Duque • 9

Protocol Debugging with POE

Simon Cozens • 11

Blocking Spam with Postfix and Amavis

Randal L. Schwartz • 16

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Andy Lester:

***Mastering Perl/Tk* • 19**

Source Code Appendix • 21

LETTER FROM THE EDITOR

Searching for Search Tools

Over the years, I've produced my fair share of commercial CD-ROMs. Along the way, the one thing I've learned is that a good search tool is hard to find.

Well, I should say it's hard to find if you want one that is lightweight, indexes without crashing, works on a variety of platforms, and doesn't cost an arm and leg to distribute on CD. I've looked at a lot of content-indexing systems, from the cheap-and-dirty right up to the expensive-and-feature-laden.

The problem is complicated by the fact that search tools aren't one application. They're two—an indexer and search client app. For CD-ROM content, you only have to run the indexer once, to produce an index. That index is distributed on your CD-ROM, along with the search client app that reads the index and lets users search the disc. Both of these apps need to be reliable.

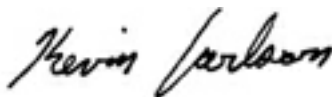
Some search tools have indexers that tie you to one platform, which means that if you want to do all your CD-ROM production work on one machine, the indexer is determining for you which machine you'll be using. Some tools have Java-based search applets that are supposed to work on any end user's VM. We all know how that goes—write once, debug everywhere. Yet another problem is that some tools require you to distribute an app on your CD-ROM that must be installed on the user's machine to allow searching of the disc. Granted, performance may be better than with a cross-platform search tool, but I still don't like CD-ROMs that do this.

So you can imagine my delight in running across jsFind, an open-source tool put together by Shawn Garbett (<http://elucidsoft.net/projects/jsfind/>). It's really just a nice customization of a popular open-source indexer called swish-e, coupled with some Perl to massage the swish-e output into an XML B-tree, and a JavaScript component to search that B-tree. Swish-e is available as source, so you can compile it on nearly any platform you like (I did it on Mac OS X). Shawn provides a patch for swish-e that lets it export its index as XML, but by the time you read this, that patch might well be incorporated into the main swish-e source. The back end of this tool, then, is usable on any machine on which you can manage to successfully compile it. So far, so good.

But what about that JavaScript component? That, too, seems to be a thing of beauty. It works in any Level 2 DOM-compliant browser. That includes many of the relatively recent browsers, and gives users the choice of at least two different browsers on any of the big three platforms—Windows, Linux, and Mac OS X. (Probably Solaris, too, though I don't have a Solaris installation to test this.) The searches are fast and the whole user experience is customizable because you have access to all the code.

From the standpoint of compatibility for end users, a JavaScript client makes so much more sense than a Java client because JavaScript comes built-in to the browser. There's no separate install just to support the search client. Most of the time, the user's browser is adequate.

jsFind isn't perfect: It has a JavaScript timeout bug that shows up on some OS/browser combinations that leaves the search stuck on a "Please wait..." page. But this is usually easily cleared by resubmitting the search. Shawn promises he's looking for a way to squash that one. If you think you know the answer, he'd probably like to hear from you.



Kevin Carlson
Executive Editor
kcarlson@tpj.com

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

THE PERL JOURNAL (ISSN 1545-7567) is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: \$18.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2003 by CMP Media LLC, unless otherwise noted. All rights reserved.



The Perl Journal

EXECUTIVE EDITOR

Kevin Carlson

MANAGING EDITOR

Della Song

ART DIRECTOR

Margaret A. Anderson

NEWS EDITOR

Shannon Cochran

EDITORIAL DIRECTOR

Jonathan Erickson

COLUMNISTS

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

CONTRIBUTING EDITORS

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

INTERNET OPERATIONS

DIRECTOR

Michael Calderon

SENIOR WEB DEVELOPER

Steve Goyette

WEB DEVELOPER

Bryan McCormick

WEBMASTERS

Sean Coady, Joe Lucca, Rusa Vuong

MARKETING / ADVERTISING

PUBLISHER

Timothy Trickett

MARKETING DIRECTOR

Jessica Hamilton

GRAPHIC DESIGNER

Carey Perez

THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403

650-513-4300. <http://www.tpj.com/>

CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND

MARKETING Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

PRESIDENT, TECHNOLOGY SOLUTIONS Robert Faletta

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED

TECHNOLOGIES Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenber

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER ENTERPRISE

ARCHITECTURE GROUP Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE

DEVELOPMENT MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING

SOLUTIONS Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES

Marie Myers

Perl News

Language Updates

Perl 5.8.2 has been finalized, after two release candidates; you can get it from CPAN under `/authors/id/N/NW/NWCLARK`. In addition to restoring binary compatibility with pre-5.8.1 modules, this release solves syntax errors involving unrecognized *filetest* operators, and fixes a problem with initializing and destroying the Perl interpreter more than once in a single process. However, build problems have been reported on some UNIX systems. A maintenance tarball is scheduled for release December 31.

In related news, PerlCE 5.8.2 is out for those who want to run Perl on Windows CE-based devices. The project web site is <http://perlce.sourceforge.net/>.

An update was also issued to the Perl 5.6 branch: According to the documentation, “the 5.6.2 release is aimed at providing only a minimal update to perl 5.6.1, which wasn’t being compatible with some of the newest compilers (most notably gcc 3.x), libraries, and operating systems that have appeared since 5.6.1 was released.” Perl 5.6.3 is in the works with many more bug fixes and module upgrades.

Lastly, Leopold Toetsch announced the 0.0.13 release of Parrot: “Proposed originally as a fun release it has a remarkable list of improvements, additions, and fixes. While some milestones have not really been reached, there have been many steps towards getting these done.” You can download it from CPAN under `/authors/id/L/LT/LTOETSCH/`.

Management Changes at ActiveState

David Ascher has succeeded Dick Hardt as ActiveState’s CTO, and also gained the title of “Director, Tools and Languages” for Sophos, ActiveState’s new parent company. Hardt, the founder and one-time CEO of ActiveState, has left the company in order to found a new start-up in the identity management field.

Ascher’s duties will be broader than those of a typical CTO. While Steve Munford remains President of ActiveState, Munford has also become Global Vice President of Messaging at Sophos. His mandate will be to take the company’s PureMessage anti-spam product worldwide, so the job of overseeing day-to-day operations at ActiveState will fall to Ascher.

Ascher has a strong background in Python. He’s the director of the Python Software Foundation and a coauthor of two O’Reilly books on Python. Additionally, he has helped develop and doc-

ument the Numeric and PyOpenGL Python extensions. He first joined ActiveState nearly four years ago when the company launched its ActivePython development effort. But though ActiveState may be run by a Python enthusiast, executives say the company remains committed to Perl. “Most of our open-source developers are Perl developers,” says ActiveState spokeswoman Lori Pike. “For sure, we have a huge focus on Perl.”

PHP for Parrot

Sterling Hughes, coauthor of *The PHP Developer’s Cookbook*, and PHP Group member Thies C. Arntzen have started a side-project to get PHP running on Parrot. Arntzen writes on his blog, “We have already written a new scanner and parser that understands most of the current PHP-syntax. At `php{con west}` we managed to write the first bits of the actual code-generator for it and we are now able to run some (limited, but real-life) PHP code.” The next step is to choose a name for the project: The rather obvious “Pharrot” has already been rejected. A SourceForge page will be forthcoming as soon as a name is chosen; meanwhile, Hughes and Arntzen have posted a presentation on the project at <http://www.edwardbear.org/pap.pdf>.

LL3 Wrap-Up

The Lightweight Languages 2003 conference was held at MIT in the beginning of November. Webcasts of the presentations are archived at <http://ll3.ai.mit.edu/>; at the end of Session 2 you can catch Dan Sugalski talking about boundaries. Other topics included “Lisp on Lego MindStorms” and “Embedding an Interpreted Language Using Higher-Order Functions and Types.”

Get Well Soon, Larry

The Perl community is surely united in its sympathy and best wishes for Larry Wall, who posted to `perl.perl6.language` to say “Sorry I haven’t be more communicative lately, but I spent most of the last two months in the hospital, where there’s no e-mail access. Had a benign tumor chopped out of my stomach, and there were complications. Full recovery is expected, any month now. But for the time being I’m tied to an IV pole pumping calories into my intestines until I learn to eat again. So please don’t expect me to come to any conferences any time soon...” Here’s to better food, a speedy recovery, and holidays at home.

Programming Graphical Applications with Gtk2-Perl, Part 2

Gtk+ is a toolkit for creating graphical applications on X11 systems. It has also been ported to Windows and BeOS, and runs on Mac OS X using Apple's X11 server. It was originally created for the GIMP, a popular image-manipulation program, and later became the toolkit for GNOME, the GNU Network Object Model Environment.

In Part 1 of this article, published in *TPJ* last month, I gave a brief overview of the event-based programming model and discussed some of the most common widgets available. This month, I'll complete the exploration of the Gtk+ widget set and also explore some of the handy time-saving features of Gtk2-Perl.

Container Widgets Continued: Panes

Panes in Gtk+ are analogous to frames in HTML—they break the window into two separate parts, with a bar between them that can be dragged to resize the two parts.

Pane widgets come in two flavors: horizontal and vertical. They are both created in the same way:

```
my $hpaned = Gtk2::HPaned->new;  
my $vpaned = Gtk2::VPaned->new;
```

Panes can contain precisely two widgets—to reflect this, you add widgets to it using the *add1()* and *add2()* methods:

```
$hpaned->add1($left_widget);  
$hpaned->add2($right_widget);
```

or

```
$vpaned->add1($top_widget);  
$vpaned->add2($bottom_widget);
```

Listing 8 shows how the paned widgets can be used. The resulting widgets are shown in Figure 8.

Gavin works for the domain registry CentralNic, and was a coauthor of Professional Perl Programming (Wrox Press, 2001). He can be contacted at gavin.brown@uk.com.

Scrolled Windows

Scrolled windows are handy when you want to place a widget inside a scrollable area. Almost all applications have something like this because inevitably, the data the program uses will grow so that it can't all be displayed at once.

Scrolled windows are often used in conjunction with another kind of container called a *viewport*. Viewports are rarely used without a scrolled window, so I won't talk about them, but you do need to know about them.

Listing 9 is a simple example of scrolled window usage, which results in the window shown in Figure 9. The scrolled window allows us to view the butterfly, even though it's much larger than the window size.

Some widgets have their own scrollbars and don't need a viewport. These include the *Gtk2::TextView* and *Gtk2::TreeView* widgets. For these, you just need to use the *add()* method.

The *add_with_viewport()* method creates a new viewport, adds the image to it, and then adds it to the scrolled window. The *set_policy()* method tells the widget under what conditions it should display horizontal and vertical scrollbars, in that order. When set to "automatic," it will display a scrollbar when the child widget is larger than the scrolled window. Other values are "always" and "never."

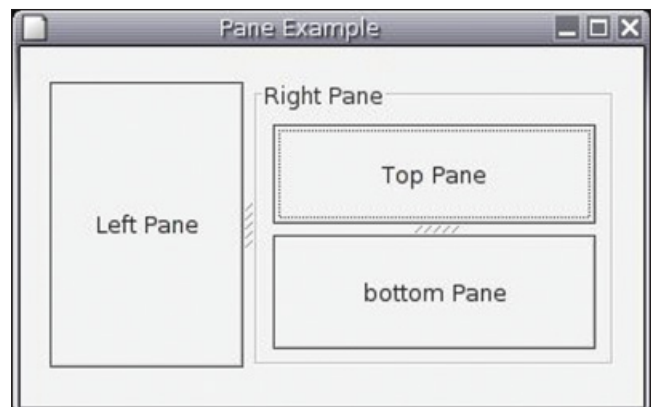


Figure 8: Paned widgets.



Figure 9: Scrolled windows.

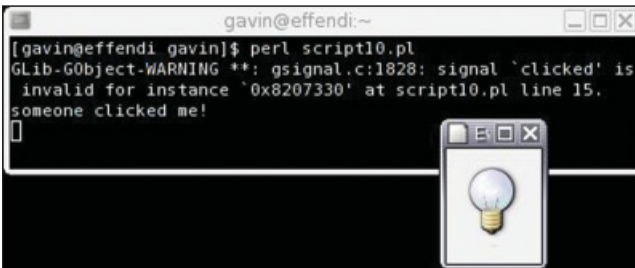


Figure 10: An image in an event box.

Event Boxes

Event boxes allow you to add callbacks to widgets for events and signals that they don't support. For example, a *Gtk2::Image* cannot have any signals connected to it. However, if you place the image inside an event box, you can connect a signal to the event box, as shown in Listing 10 and the corresponding Figure 10.

Manipulating Graphics with *Gtk2::Gdk::Pixbuf*

The *gdk-pixbuf* libraries (namespaced as *Gtk2::Gdk::Pixbuf* in Perl) provide a flexible and convenient way to handle graphics in Gtk+. *gdk-pixbuf* supports a wide range of image formats, including PNG, JPG, GIF, and XPM. It also has support for animations.

The *GdkPixbuf* provides some methods for basic image manipulation. However, you may find that libraries such as *GD* and *Image::Magick* will be better for more complicated tasks.

Resizing an Image

To resize a pixbuf, use the *scale_simple()* method:

```
$pixbuf->scale_simple(
    $destination_pixbuf,
    $new_width,
    $new_height,
    $scaling_type
);
```

This will resize *\$pixbuf* and place the resulting image into *\$destination_pixbuf*. If you don't want to create a new object, just replace *\$destination_pixbuf* with *\$pixbuf*. *\$new_width* and *\$new_height* are the dimensions desired, in pixels. *\$scaling_type* determines the algorithm to be used: "bilinear" produces better quality images, but is slower than "nearest."

If you want to resize the pixbuf into a new pixbuf object, you must make sure that the destination pixbuf has the same properties as the source. To do this, use the various *get_**() methods as arguments to the constructor:

```
my $destination_pixbuf = Gtk2::Gdk::Pixbuf->new(
    $source_pixbuf->get_colorspace,
    $source_pixbuf->get_has_alpha,
    $source_pixbuf->get_bits_per_sample,
    $source_pixbuf->get_width,
    $source_pixbuf->get_height,
);
```

Changing the Brightness of a pixbuf

To change the brightness of an image, use the *saturate_and_pixelate()* method:

```
$pixbuf->saturate_and_pixelate(
    $destination_pixbuf,
    2,
    0
);
```

The second argument is the factor by which to brighten or darken the image: If it's less than 1.0, the image is darkened; if it's higher, then the image is brightened. The last argument determines whether the image is "pixelated." This draws a checkerboard over the image as though it were disabled. In this case, we don't want that, so it's set to 0.

As with the example above, the destination pixbuf must have the same properties as the source.

Copying a Region from a pixbuf

You can copy a region of the image out of a pixbuf using the *copy_area()* method:

```
$source_pixbuf->copy_area(
    $src_x,
    $src_y,
    $width,
    $height,
    $destination_pixbuf,
    $dest_x,
    $dest_y
);
```

\$src_x and *\$src_y* are the horizontal and vertical coordinates of the top left corner of the region to be copied. *\$width* and *\$height* define its dimensions. *\$dest_x* and *\$dest_y* are the coordinates in *\$destination_pixbuf* where the copied area will be positioned.

Writing pixbuf Data to Disk

You can write a pixbuf to disk in any format that *gdk-pixbuf* supports. For example:

```
$pixbuf->save('image.png' 'png');
```

writes the pixbuf in PNG format.

Getting the pixbuf of a Stock Icon

Should you need the pixbuf of a stock icon, you can use the *render_icon()* method of any available widget:

```
my $stock_pixbuf = $window->render_icon($stock_id, $stock_size);
```

See "More Information" for references to the allowed values for *\$stock_id* and *\$stock_size*.

Glade

The Rapid Application Development (RAD) tool Glade is a WYSIWYG graphical interface designer. Using it, you can construct an interface in minutes that would take hours to hand craft.

The Glade program produces UI description files in XML format, and the *Gtk2::GladeXML* module is capable of reading these XML files and constructing your interface at runtime. The advantage in this is immediate—it provides for a separation between logic and presentation that will make your job a lot easier in the future. And if you happen to work with a Human Computer Interface (HCI) expert, they can have complete control over the interface without ever having to touch a line of code, and you can get on with the job of making the program work without having to constantly revise the interface.

Here's a simplified example of how to use a Glade XML file in your program:

```
#!/usr/bin/perl
use Gtk2 -init;
use Gtk2::GladeXML;
use strict;

my $gladexml = Gtk2::GladeXML->new('example.glade');

$gladexml->signal_autoconnect_from_package('main');

Gtk2->main;

exit;

# this is called when the main window is closed:
sub on_main_window_delete_event {
    Gtk2->main_quit;
}

[snip]
```

This program takes a Glade file called “example.glade” and creates an interface. When designing your interface, you can define the names of subroutines to handle widget signals, and the *signal_autoconnect_from_package()* method tells *Gtk2::GladeXML* which package name the handlers can be found in.

Each widget in the Glade file has a unique name, for example, *my_main_window* or *label5*. You can get the widget object for a specific object using the *get_widget()* method:

```
my $label = $gladexml->get_widget('label5');
$label->set_text('Hello World!');
```

You may find that just experimenting with Glade will be a good learning exercise. Glade is also an excellent prototyping utility, so that even if you don't plan to make use of *Gtk2::GladeXML*, you will find it invaluable for planning how your program will be organized.

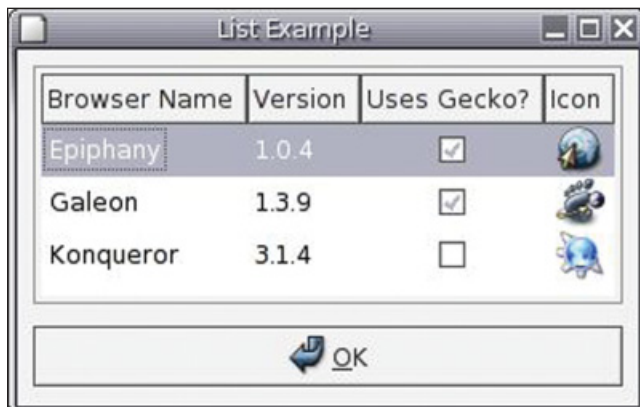


Figure 11: Using *Gtk2::SimpleList*.

Gtk2::SimpleList

As well as providing Perl bindings for the Gtk+ widget set, *Gtk2-Perl* comes with a couple of extra modules that provide a much more Perlish way of working with some of the more complex widgets. An example is the Model, View, Controller (MVC) system for list and tree widgets. While being extremely powerful, they're a pain to work with, especially since the average application will rarely make use of the full features available.

Gtk2::SimpleList is designed to make working with lists easy. Listing 11 is an example of its usage. You can see what this program looks like in Figure 11.

Gtk2::SimpleList acts as a wrapper around the standard *Gtk2::TreeView* and inherits all of the latter's methods, so that once your list is created you can manipulate it in the usual way. Data is represented as a Perl list of lists (an array of array references), and can be modified directly without having to redraw the entire list.

In the constructor, we are required to define the columns in the list we will use. We do this using a pseudohash where the column titles are the keys, and their types are the values. The allowed values are “text” for a text string, “int” for an integer, “double” for a double-precision floating-point value and “pixmap” for a *Gtk::Gdk::Pixmap* object.

We can populate the data in the list either by writing to the *\$list->{data}* value or using a subroutine.

In addition to the methods it inherits from *Gtk2::TreeView*, *Gtk2::SimpleList* also provides:

```
$list = Gtk2::SimpleList->new_from_treeview($treeview, [column_data]);
```

This creates a list using an already created *treeview* widget. This is handy for when you're using Glade and such a widget has already been created for you.

```
@indices = $list->get_selected_indices;
```

This returns an list of numbers representing the currently selected rows in the list.

```
$list->set_data_array($arrayref);
```

This method populates the list data array. *\$arrayref* is a reference to an array of array references.

```
$list->select(@rows);
$list->unselect(@rows);
```

These methods select and unselect the rows identified in the *@rows* array.

```
$list->set_column_editable($column, $editable);
```

Cells in the list can be edited by the user—this method can be used to enable or disable this feature. *\$column* is the index of the column, and *\$editable* should be a true value to enable editing, or false to disable it.

Gtk2::SimpleMenu

This module makes it easy to create hierarchical application menus (the File, Edit, View menu at the top of each window). The constructor takes as an argument a reference to an array containing a tree of menus, submenus, and items. (See Listing 12; you can see the resulting program in Figure 12.)

Each item and submenu is represented by an anonymous hash, containing data needed to set up the menu. The various *item_types* include *Branch* for a submenu, *LastBranch* for a right-justified menu, *Item* for a standard item, *StockItem* for an item with a stock item (which is then specified in the *extra_data* key), and *RadioItem*

and *CheckItem*, which create items with radio buttons and checkboxes, respectively.

You can define a callback for when the item is clicked using the *callback* key, but you can also specify a *callback_action*, which is then passed as an argument to the function defined by the *default_callback* argument in the constructor.

Creating Your Own Widgets

It is very easy to create custom widgets with Gtk2-Perl. All you need to do is create a package that inherits from an existing Gtk+ widget, and extend the functionality. Listing 13 is an example of a simple extension to *Gtk2::Button*. The resulting program is shown in Figure 13.

The first thing we need to do is declare what package we want to inherit from. This is as easy as

```
use base 'Gtk2::SomeWidget';
```

Then we need to override the constructor. But we need to get hold of a *Gtk2::SomeWidget* object so we can play with it. The *\$package->SUPER::new* expression does this for us—then we can *re-bless()* it into our own package and start playing.

The *Gtk2::SimpleList* and *Gtk2::SimpleMenu* widgets are examples of this kind of custom widget, which is derived from an existing Gtk+ widget. There's also the *Gtk2::PodViewer* widget, available from CPAN, which provides a Perl POD rendering widget based on *Gtk2::TextView*.

More Information

This article cannot really cover every aspect of Gtk2-Perl programming, but I hope it has given you enough of an introduction that you will want to find out more. There is certainly a steep learning curve, but once you've reached the top I'm sure you'll find that it was worth it!

To help you on your way, I have compiled a reasonably complete list of references that should make your Gtk2-Perl programming much less painful.

Gtk+ Documentation

There is no direct POD documentation for Gtk2-Perl. This is because the Perl bindings adhere as closely as possible to the original C implementation. The C API is fully documented and is available in two forms:

Online. You can read the full Gtk+ documentation at <http://www.gtk.org/api/>. This includes documentation for Gtk+, the Gtk+ stock icon set, the *Glib* general-purpose utility library, *GdkPixbuf*, and *Pango*.

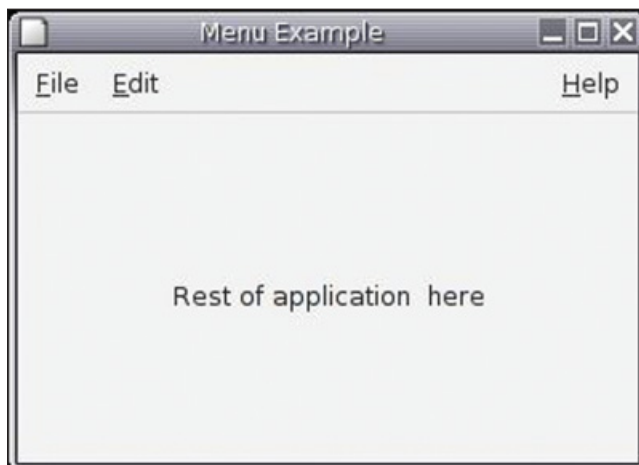


Figure 12: Creating menus with *Gtk2::SimpleMenu*.

Devhelp. *Devhelp* is an API documentation browser that works natively with the Gtk+ documentation system. As well as having a full index of the documentation, you can also search the function reference. It is available from <http://www.imendio.com/projects/devhelp/>, and I thoroughly recommend it.

If you're not familiar with C you may have trouble at first translating the examples from C to Perl. Fear not! The Gtk2-Perl team has documented the C to Perl mapping in the *Gtk2::api* POD document.

Tutorials

There are a number of tutorials that will help the learning process. Stephen Wilhelm's tutorial on the old Gtk-Perl 1.x series still works with Gtk2-Perl in 90 percent of the examples, and is a pretty complete guide to the available widgets. The URL for his tutorial is <http://jodrell.net/files/gtk-perl-tutorial/>.

Ross McFarland, one of the Gtk2-Perl developers, has also written a tutorial that is available at <http://gtk2-perl.sourceforge.net/doc/intro/>.

General Information

The main Gtk+ web site is at <http://www.gtk.org/> and the Gtk2-Perl site is <http://gtk2-perl.sf.net/>. The Gtk2-Perl site has a great deal of extra information including documentation for *Glib*, as well as build utilities like *ExtUtils::PkgConfig* (the perl bindings *pkg-config*), and information on developing XS bindings and the internals of Gtk2-Perl.

Asking Questions

Gtk2-Perl has a mailing list at <http://mail.gnome.org/mailman/listinfo/gtk-perl-list>. There is also an active IRC channel—join #gtk-perl on irc.gnome.org and ask your question.

Installing Gtk2-Perl

If you like what you see here and want to get started with Gtk2-Perl, the first thing you'll need to do is install the Gtk2-Perl modules. You can do this by downloading and compiling the source libraries from the Gtk2-Perl web site, but you can also use the CPAN bundle. With this, installation is as simple as issuing this command:

```
perl -MCPAN -e 'install Bundle::Gnome2'
```

This will install the latest versions of all the available Gtk2-Perl libraries.

TPJ

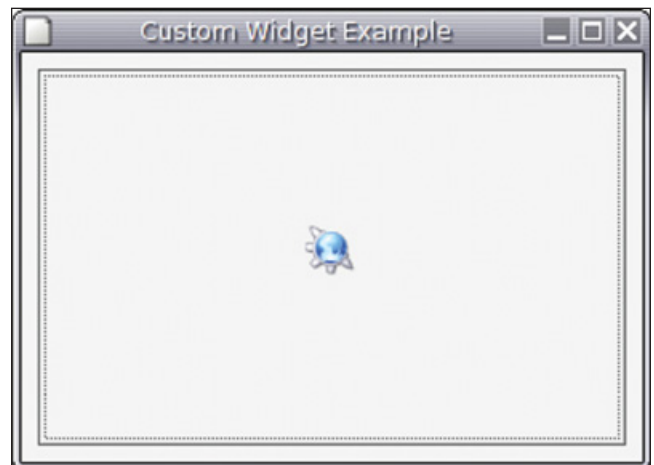


Figure 13: A simple extension to *Gtk2::Button*.

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

Listing 8

```
#!/usr/bin/perl
use Gtk2 -init;
use strict;

my $window = Gtk2::Window->new;
$window->set_title('Paned Example');
$window->signal_connect('delete_event', sub { Gtk2->main_quit });
$window->set_border_width(8);

my $vpaned = Gtk2::VPaned->new;
$vpaned->set_border_width(8);

$vpaned->add1(Gtk2::Button->new('Top Pane'));
$vpaned->add2(Gtk2::Button->new('bottom Pane'));

my $frame = Gtk2::Frame->new('Right Pane');
$frame->add($vpaned);

my $hpaned = Gtk2::HPaned->new;
$hpaned->set_border_width(8);

$hpaned->add1(Gtk2::Button->new('Left Pane'));
$hpaned->add2($frame);

$window->add($hpaned);

$window->show_all;

Gtk2->main;
```

Listing 9

```
#!/usr/bin/perl
use Gtk2 -init;
use strict;

my $window = Gtk2::Window->new;
$window->set_title('Scrolled Window Example');
$window->signal_connect('delete_event', sub { Gtk2->main_quit });
$window->set_border_width(8);

my $image = Gtk2::Image->new_from_file('butterfly.jpg');

my $scrwin = Gtk2::ScrolledWindow->new;
$scrwin->set_policy('automatic', 'automatic');

$scrwin->add_with_viewport($image);

$window->add($scrwin);

$window->show_all;

Gtk2->main;
```

Listing 10

```
#!/usr/bin/perl
use Gtk2 -init;
use strict;

my $window = Gtk2::Window->new;
$window->set_title('Event Box Example');
$window->signal_connect('delete_event', sub { Gtk2->main_quit });
$window->set_border_width(8);

my $image = Gtk2::Image->new_from_stock('gtk-dialog-info', 'dialog');

# a Gtk2::Image widget can't have the 'clicked' signal, so this causes
# an error:
$image->signal_connect('clicked', \&clicked);

my $box = Gtk2::EventBox->new;

# but this works!
$box->signal_connect('button_release_event', \&clicked);

$box->add($image);

$window->add($box);

$window->show_all;

Gtk2->main;

sub clicked {
    print "someone clicked me!\n";
}
```

Listing 11

```
#!/usr/bin/perl
use Gtk2 -init;
use Gtk2::SimpleList;
use strict;

my $window = Gtk2::Window->new;
$window->set_title('List Example');
$window->set_default_size(300, 200);
$window->signal_connect('delete_event', sub { Gtk2->main_quit });
$window->set_border_width(8);

my $list = Gtk2::SimpleList->new(
    'Browser Name' => 'text',
    'Version'      => 'text',
    'Uses Gecko?'  => 'bool',
    'Icon'         => 'pixbuf'
);

$list->set_column_editable(0, 1);

@{$list->{data}} = (
    [ 'Epiphany', '1.0.4', 1, Gtk2::Gdk::Pixbuf-
    >new_from_file('epiphany.png') ],
    [ 'Galeon', '1.3.9', 1, Gtk2::Gdk::Pixbuf-
    >new_from_file('galeon.png') ],
    [ 'Konqueror', '3.1.4', 0, Gtk2::Gdk::Pixbuf-
    >new_from_file('konqueror.png') ],
);

my $scrwin = Gtk2::ScrolledWindow->new;
$scrwin->set_policy('automatic', 'automatic');

$scrwin->add_with_viewport($list);

my $button = Gtk2::Button->new_from_stock('gtk-ok');
$button->signal_connect('clicked', \&clicked);

my $vbox = Gtk2::VBox->new;
$vbox->set_spacing(8);

$vbox->pack_start($scrwin, 1, 1, 0);
$vbox->pack_start($button, 0, 1, 0);

$window->add($vbox);

$window->show_all;

Gtk2->main;

sub clicked {
    my $selection = ($list->get_selected_indices)[0];
    my @row = @{$list->{data}}[$selection];
    printf(
        "You selected %s, which is at version %s and %s Gecko.\n",
        $row[0],
        $row[1],
        ($row[2] == 1 ? "uses" : "doesn't use")
    );
    Gtk2->main_quit;
}
```

Listing 12

```
#!/usr/bin/perl
use Gtk2 -init;
use Gtk2::SimpleMenu;
use strict;

my $window = Gtk2::Window->new;
$window->set_title('Menu Example');
$window->set_default_size(300, 200);
$window->signal_connect('delete_event', sub { Gtk2->main_quit });

my $vbox = Gtk2::VBox->new;

my $menu_tree = [
    _File => {
        item_type => '<Branch>',
        children => [
            _New => {
                item_type => '<StockItem>',
                callback => \&new_document,
                accelerator => '<ctrl>N',
                extra_data => 'gtk-new',
            },
            _Save => {
                item_type => '<StockItem>',
                callback => \&save_document,
                accelerator => '<ctrl>S',
            },
        ],
    },
];
```



```

        extra_data    => 'gtk-save',
    },
    _Quit => {
        item_type      => '<StockItem>',
        callback        => sub { Gtk2->main_quit },

        accelerator    => '<ctrl>Q',
        extra_data      => 'gtk-quit',
    },
    ],
    ],
    _Edit => {
        item_type => '<Branch>',
        children => [
            _Cut => {
                item_type      => '<StockItem>',
                callback_action => 0,
                accelerator    => '<ctrl>X',
                extra_data      => 'gtk-cut',
            },
            _Copy => {
                item_type      => '<StockItem>',
                callback_action => 1,
                accelerator    => '<ctrl>C',
                extra_data      => 'gtk-copy',
            },
            _Paste => {
                item_type      => '<StockItem>',
                callback_action => 2,
                accelerator    => '<ctrl>V',
                extra_data      => 'gtk-paste',
            },
        ],
    },
    ],
    _Help => {
        item_type => '<LastBranch>',
        children => [
            _Help => {
                item_type      => '<StockItem>',
                callback_action => 3,
                accelerator    => '<ctrl>H',
                extra_data      => 'gtk-help',
            },
        ],
    },
    ],
];

my $menu = Gtk2::SimpleMenu->new (
    menu_tree      => $menu_tree,
    default_callback => \&default_callback,
);

$vbox->pack_start($menu->{widget}, 0, 0, 0);
$vbox->pack_start(Gtk2::Label->new('Rest of application here'), 1, 1, 0);

$window->add($vbox);

$window->show_all;

Gtk2->main;

sub new_document {
    print "user wants a new document.\n";
}

sub save_document {
    print "user wants to save.\n";
}

sub default_callback {
    my (undef, $callback_action, $menu_item) = @_;
    print "callback action number $callback_action\n";
}

Listing 13

#!/usr/bin/perl
use Gtk2 -init;
use strict;

my $window = Gtk2::Window->new;
$window->set_title('Custom Widget Example');
$window->set_default_size(150, 100);
$window->signal_connect('delete_event', sub { Gtk2->main_quit });
$window->set_border_width(8);

my $button = Gtk2::FunnyButton->new(
    normal    => 'Click me!',
    over      => 'Go on, click me!',
    clicked   => 'Click me harder!',
);

```

```

$window->add($button);

$window->show_all;

Gtk2->main;

package Gtk2::FunnyButton;
use base 'Gtk2::Button';
use strict;

sub new {
    my ($package, %args) = @_;
    my $self = $package->SUPER::new;
    bless($self, $package);
    $self->{labels} = \%args;
    $self->add(Gtk2::Label->new($self->{labels}{'normal'}));
    $self->signal_connect('enter_notify_event', sub { $self->child->set_text($self->{labels}{'over'}) });
    $self->signal_connect('leave_notify_event', sub { $self->child->set_text($self->{labels}{'normal'}) });
    $self->signal_connect('clicked', sub { $self->child->set_text($self->{labels}{'clicked'}) });
    return $self;
}

```

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.

Finding Duplicate Files

Numerous programs in the UNIX system (and its clones) are small and yet excel at what they do. Individually, they can't do much, but when used together they can make a large task easier to implement, debug, and maintain. In keeping with this UNIX philosophy, I'll show you how to use a little UNIX utility, *find*, and two small Perl scripts to accomplish the job of finding duplicate files in a given directory.

Using *find*

find is a small but powerful utility that is available on all UNIX/Linux systems. The following command, for example, tells *find* to descend into /tmp (and recursively descend into all subdirectories it encounters), and print to the standard output the names of all files and subdirectories it finds:

```
find /tmp -name ***
```

find's output is similar to this:

```
/tmp
/tmp/textpdf-root
/tmp/Gladman
/tmp/Gladman/sha2.c
/tmp/Gladman/uitypes.h
/tmp/Gladman/test.c
/tmp/Gladman/sha2.h
/tmp/Gladman/a.out
/tmp/guile-1.6.4
```

The best feature of *find*, just like any good UNIX tool, is that its output can be redirected as input to another program. So, instead of displaying its output to the screen, you can use a pipe to "hand over" its output to the next program for processing, as in

```
find /tmp -name *** | ./md5
```

Julius is a freelance network consultant in the Philippines. He can be contacted at jcdunque@lycos.com.

The symbol "|" is the pipe, and the command above tells us that md5 is the next program that takes in *find*'s output.

The md5 Script

md5 is a short Perl script that uses the MD5 one-way hash function to generate a unique message digest for a given input file. The md5 script is shown in Listing 1. The MD5 algorithm guarantees that every file has its own unique digest. (Well, not quite. But mathematically speaking, the probability of two different files having identical digests is 1 in 2^{64} , which is practically nil.) So, if MD5 says that two files have the same digest, you can be sure, with a very high degree of confidence, that they are indeed identical.

The output of the command:

```
find /tmp -name *** | ./md5
```

gives us something like:

```
294da4cc09dd0024a1f21a7f810dfe60 /tmp/Gladman/sha2.c
7218b118cc8de06b13bd4d997cb00250 /tmp/Gladman/uitypes.h
72f0f6f900800fbdb7b14af552b40d8a6 /tmp/Gladman/test.c
b0b8baelaaf4169ec6dffbla78b13372 /tmp/Gladman/sha2.h
009b12ab19c4e5640f2bd263f4970d70 /tmp/Gladman/a.out
```

Each line of md5's output consists of a 32-character MD5 digest, followed by two single spaces, and lastly, a filename.

Line 7 of md5 uses the "diamond operator" (\diamond) to receive whatever that is output by *find*. Although invisible to the naked eye, each line of *find*'s output ends in a newline, something we don't need. md5 strips off these newlines via the *chomp* command (line 8). Line 9 checks whether or not the input line is a regular file (the *-f* switch).

Most novice Perl programmers would be puzzled by lines 7–9, since \diamond , *chomp*, and *-f* seem to operate on a global variable that was not explicitly defined anywhere in the script. Let me explain lines 7–9 in more detail.

By default, every input line read by \diamond is loaded into the special variable, $$_$. When *chomp* is called without any argument, it

is understood that we mean to discard the newline character of `$_`. The `-f` switch (guess what?) also acts on `$_` implicitly. `$_` is updated every time a new input line is read.

Lines 11–13 generate an MD5 digest of the filename, and line 15 prints the digest and the filename.

find is a small but powerful utility that is available on all UNIX/Linux systems

The finddup Script

finddup takes the output of md5, again, through the use of a pipe. finddup then produces an associative array, `%dups`, using the digests computed by md5 as keys. See Listing 2 for the whole finddup script.

On lines 11 and 12 of finddup, we again employ the familiar `<>` and `chomp`, respectively. On line 13, the first 32 characters of `$_` are assigned to `$digest`, while the rest of `$_` gets assigned to `$filename`.

On line 15, `$filename` is saved in `%dups`, using `$digest` as key. Normally, when the contents of `%dups` are finally printed out, duplicate files for every key are displayed on one line only, with just a single space separating them. This is ugly to look at. To make the output pleasing to the eye (I hope), I saved a newline every time `$filename` is push-ed into `%dups` (line 14). This way, the filenames are displayed in one nice-looking column. In doing this, however, notice that every key of `%dups` now contains at least two elements (the filenames and the newlines).

On lines 18–24, finddup loops through `%dups`, and checks for keys with multiple elements. On line 19, if a certain key (the MD5

digest) contains more than two elements (take the newlines into account), we know that duplicate files exist for this digest, and they get printed to the standard output (line 21).

If finddup is called with the `--verbose` or `-v` options switched on, the digests for duplicate files are printed as well (line 20). Line 22 is there for cosmetic purposes only.

Sample Output

On my Linux machine, I made a directory and called it `testdir`. I then created six small test files, some with duplicates.

The command:

```
find ./testdir -name *** | ./md5 | ./finddup --verbose
```

or

```
find ./testdir -name *** | ./md5 | ./finddup -v
```

produces these results:

```
1 900150983cd24fb0d6963f7d28e17f72
2 ./testdir/ba3e2571
3 ./testdir/9cd0d89d
4 ./testdir/a9993e36
5
6 d4b7c284882ca9e208bb65e8abd5f4c8
7 ./testdir/1a3472da
8 ./testdir/f2c8d22e
9 ./testdir/hello world
```

The three files listed on lines 2–4 are all identical; their MD5 digest is printed on line 1. Likewise, line 6 shows the digest of identical files printed on lines 7–9.

Unfortunately, you'll find out that, using either of the commands above, it's not possible to get the digest of a file that starts with a dot ("`.`"). There's a way, though, but I'll leave this as an exercise for the interested reader. (Hint: check the *find* man page.)

TPJ

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

Listing 1

```
1 #!/usr/local/bin/perl
2 use diagnostics;
3 use strict;
4 use warnings;
5 use Digest::MD5;
6
7 while (<>) {
8     chomp;
9     if (-f) {
10         open INFILE, $_;
11         my $context = new Digest::MD5;
12         $context->addfile(*INFILE);
13         my $digest = $context->hexdigest;
14         close INFILE;
15         print "$digest $_\n";
16     }
17 }
```

Listing 2

```
1 #!/usr/local/bin/perl
2 use diagnostics;
3 use strict;
4 use warnings;
5 use Getopt::Long;
6
7 my %dups = ();
8 my ($digest, $filename, $verbose) = ();
9 GetOptions("verbose" => \$verbose);
10
```

```
11 while (<>) {
12     chomp;
13     ($digest, $filename) = /(.{32})(.*)/;
14     push(@{$dups{$digest}}, "\n");
15     push(@{$dups{$digest}}, $filename);
16 }
17
18 foreach $digest (sort keys %dups) {
19     if (scalar(@{$dups{$digest}}) > 2) {
20         print "$digest" if ($verbose);
21         print "@{$dups{$digest}}\n";
22         print "\n" if ($verbose);
23     }
24 }
```

TPJ



Protocol Debugging with POE

Simon Cozens

Last time we met, I talked about the very important work I'd been doing in allowing Perl hackers to waste a lot of time on Internet Poker servers, with the Online Poker Protocol and *Games::Poker::OPP*.

I also mentioned that when I wrote that module, I needed a protocol debugger to help me understand how the binary protocol works, which turned out to be an interesting little project in its own right.

The Problem

In the good old days, when people set out to create a communications protocol, they did so following the UNIX design philosophies, keeping the protocols simple, easy for humans to debug and understand, and, primarily, text-based. (Eric Raymond has a lot to say about this in "The Importance of Being Textual," Chapter 5 of *The Art of Unix Programming*; see <http://www.catb.org/~esr/writings/taoup/html/ch05s01.html>.)

For instance, if I have a problem with a mail server, I know that I can happily telnet to port 25 on the offending machine and type commands in SMTP to help understand the problem and the server's responses. (In fact, I've been putting together a guide to help people do just this at <http://simon-cozens.org/programmer/phrasbook.html>) This is a major bonus if I'm writing clients for the protocol since I can telnet in and chat away with the server myself, discovering what responses to expect in edge cases.

Then the discipline of computer science progressed, and people began designing and implementing binary-based protocols, generally with fixed-width commands making them difficult to extend, and removing the ability to effectively debug them via telnet.

Unfortunately, the poker protocol I began implementing last month is one such protocol. I wanted to check that I was packing the data correctly and getting the responses I expected, but I didn't yet have a working client that I could use to confirm that! In short, the problem is that my keyboard doesn't have a NULL key, nor would I be able to read the output I get back even if it did.

So I decided that the nicest way around the problem was to create a version of telnet that could easily send null characters and

the like, and could translate any such characters it got back into something visible.

The Overengineering Stage

Of course, there were many ways I could implement such a beast. One idea would be to write a wrapper around netcat, which would turn binary data into something readable and pack user input into binary data. But I started thinking that I'd like to have line-editing capabilities and other bits and pieces, and I could foresee this nice little program turning into a massive monster.

But then someone pointed me at the lovely *Term::Visual* module, which provides a Curses-based terminal front end to an application. Anyone who's used a text-mode IRC client will be familiar with the sort of interface provided by *Term::Visual*; see Figure 1.

There's a title line at the top, a couple of status lines near the bottom, a line for user input underneath that, and the main "conversation" goes on in the middle. *Term::Visual* does its magic by hooking into the POE module. POE is both a Perl module and a way of programming; it provides many of the features of a time-scheduling operating system, so before we get into looking at *Term::Visual* in any depth, let's take a look at how POE works.

POEtry in Motion

When Linus Torvalds announced the source of his new operating system to Usenet, apparently he was particularly excited to be able

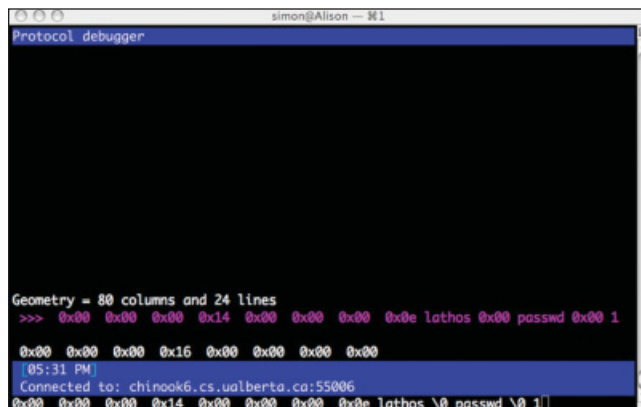


Figure 1: *Term::Visual*'s interface.

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

to start a process that wrote “aaaaa...” to standard output and another one that wrote “bbbbbb...” and have the two processes run simultaneously:

```
abababababab...
```

Let’s emulate that great moment with POE. The POE equivalent to a process is called a session.

```
use POE;
POE::Session->create();
POE::Session->create();
```

At the moment, these sessions don’t do very much. This is for two reasons. The first reason is that we haven’t given them much in the way of code. Unlike operating-system processes, POE sessions don’t do anything by themselves, but only respond to *interrupts*, which POE calls *events*. There are certain system-defined events, such as `_start`, which are posted to sessions by POE itself, but other events need to be defined and posted explicitly.

The second reason this won’t do very much is that, just as with an operating system, having two processes around isn’t all that useful unless the kernel is running. POE has a kernel as well, and we need to tell it to run when we’re ready.

Let’s address these two issues and come up with a slightly expanded version of the code:

```
use POE;
POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->alias_set("a_process") }
    });
POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->alias_set("b_process") }
    });
POE::Kernel->run;
```

Now we’re getting further, but not much further. The kernel wakes up and runs the `_start` events of each session. The `_start` events are associated with subroutine references, and these subroutines tell the kernel the name it should give to each session. If you’re wondering what `$_[KERNEL]` is, that’s simply looking at the `KERNEL`th element of the parameters passed to the subroutine, where `KERNEL` is a constant exported by POE, which specifies where in the argument list the `POE::Kernel` object will come.

However, once we’ve named each session, there’s little else to do, so the kernel quits. Let’s rectify that by creating and calling some user-defined events:

```
use POE;
POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->alias_set("a_process") },
        say_something => \say_a
    });
POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->alias_set("b_process") },
        say_something => \say_b
    });
$|++;
POE::Kernel->post("a_process", "say_something");
POE::Kernel->run;

sub say_a { print "a"; $_[KERNEL]->post("b_process", "say_something"); }
sub say_b { print "b"; $_[KERNEL]->post("a_process", "say_something"); }
```

Unlike Linux, POE’s event loop isn’t preemptively multitasking; instead, sessions need to cede time to each other. In this case, our *a* session responds to the `say_something` event by printing an “a” and then posting another `say_something` to the *b* session. Similarly, the *b* session prints its “b” and sends a `say_something` event to the *a* session. All we need is an initial `say_something` event to be posted to one of the sessions to kick the whole thing off, and:

```
abababababab....
```

*Anyone who’s used a text-mode
IRC client will be familiar with the
sort of interface provided by
Term::Visual*

Hoorah. Our two sessions can talk to each other and process events. Now we can reveal the secret: *Term::Visual* works by providing a ready-made session that turns events it receives from POE into Curses graphics on the screen. But what can we use to generate these events?

Hymn of the Big Wheel

POE communicates with the outside world primarily through a mechanism called a wheel. A wheel is a Perl object that not only provides an interface to some kind of filehandle, but is also capable of posting events into POE. For instance, here’s a POE session that emulates *tail -f*.

```
POE::Session->create(
    inline_states => {
        _start => sub {
            my $log_watcher = POE::Wheel::FollowTail->new(
                Filename => "my_log_file.txt",
                InputEvent => "got_record",
            );

            $_[HEAP]->{watcher} = $log_watcher;
        },
        got_record => sub { print $_[ARG0], "\n"; }
    });
```

When this session starts up, it creates a new *POE::Wheel::FollowTail*. This wheel opens a file and watches for new lines being added to it. When it sees a new line, it posts whatever event we’ve defined as being its *InputEvent*—in our case, that’s *got_record*.

Now, we don’t want this object to be immediately destroyed at the end of the `_start` block, so we need to stash it somewhere for safe keeping. POE provides another named parameter, the *HEAP*, which is a hash reference that sessions can use to keep stuff in. Accordingly, we file away our log watcher in the heap.

This time when our kernel runs, as well as spewing out *a*’s and *b*’s, it will detect when lines are put on the end of the log file, and

call our session's *got_record* event. The event gets passed an argument—the line that has been added, which is passed as the named parameter *ARG0*, and so we print it out.

That's a simple wheel; there are more complex ones, as we'll see when we dissect the protodebug program. But for now, that's all the POE we need to know. Let's take a quick detour into how we're going to present the data flowing between the client and the server before diving back into *Term::Visual*.

Come On, Feel the Noise

There are two data presentation problems with our protocol debugger; the first is that we want to display binary data in a way that's not going to freak out the user. This happens in protodebug's *to_visual* subroutine:

```
sub to_visual {
    my $out;
    for (split //, shift) {
        if (/\\r/) { $out .= "\\r\\n"; }
        elsif (/\\n/) { $out .= "\\n\\n"; }
        elsif (/[:print:]/) { $out .= $_; }
        else { $out .= sprintf(" 0x%02x ", ord $_) }
    }
    return $out;
}
```

This caters to three types of data coming from the wire. The first type is end-of-line characters, in one form or another; these are made visible as “\n” or “\r”, but additionally a “real” newline is added to the output. This means that an SMTP conversation would appear something like this:

```
220 alibi.simon-cozens.org ESMTP Exim 3.35 blah blah blah\r
>>> HELO sailor\n
250 alibi.simon-cozens.org Hello sailor [195.188.xx.yy]\r
```

which is more or less what you'd expect; the newline stops the display from looking absolutely hideous, but the representation of the newline (“\r” or “\n”) tells you what sort of newline it actually was.

The second class of data is data that is already visible, and so we don't need to do anything specific to it. The POSIX character class *[:print:]* describes this case neatly for us:

```
elsif (/[:print:]/) { $out .= $_; }
```

And the third class is obviously just generic binary spew, which we escape as a hex character:

```
else { $out .= sprintf(" 0x%02x ", ord $_) }
```

This means that a “logged in” reply from a Poker server would look like this:

```
0x00 0x00 0x00 0x15
```

Now we need to turn to the other way around, where we have to give the user a friendly way of specifying a binary stream. We try to make this as flexible as possible by providing multiple ways to describe the same binary byte. We use a “nibbling” tokenizer to shift tokens off the user's input and add them to the binary output stream; first, we handle the “0x..” case for specifying a hex byte:

```
sub to_binary {
    my $output;
    my $input = shift;
    while ($input) {
```

```
$input =~ s/^0x([\da-fA-F]+)/
and do { $output .= chr(hex($1)); next};
```

We also want to handle backslash notation for characters like “\0”, “\n”, “\cJ”, and so on:

```
$input =~ s/^(\\(w+|s))//
and do { $output .= eval "\"$1\""; next; };
```

Then, of course, there are ordinary printable characters that can be reached from the keyboard:

```
$input =~ s/^([\x21-\x5b\x5d-\x7e]+)/
and do { $output .= $1; next };
```

But not space, it may surprise you to note! The reason for this is to allow the user to break up bytes and control sequences in a readable manner; you don't want to type, for instance:

```
0x3\0\0\0test\0\04\n
```

if you have the opportunity to type:

```
0x3 \0\0\0 test\0 \04 \n
```

To make this possible, a literal space has to be input using “\”, but other whitespace is ignored:

```
$input =~ s/^\s// and next;
```

The nibbler technique, categorized by this *s/\$^regex// and do { ...; next;}* idiom, is a great way to make simple tokenizers; for instance, it's allowed us to give backslash-space priority over a space on its own.

Finally, if we now come across a character we haven't catered for, we should give an error:

```
$vt->print($window_id, "Couldn't understand ".to_visual($input));
return;
}
```

Those are the two subroutines that marshal data from and to the user.

Sound and Vision

Now let's slip back into interfacing these routines, the TCP connection and *Term::Visual*. First, we'll set up a new *Term::Visual* object and specify some color preferences for it:

```
my $vt = Term::Visual->new( Alias => "interface" );
$vt->set_palette( ncolor      => "white on black",
                 st_frames   => "bright cyan on blue",
                 st_values   => "bright white on blue",
                 stderr_bullet => "bright white on red",
                 stderr_text  => "bright yellow on black",
                 err_input    => "bright white on red",
                 );
```

The *Alias* parameter to the constructor is just like the *alias_set* method we used in our POE sessions—we give the terminal an alias so that we can post events to it during the course of the program.

The next stage is to set up our window, which will contain the two-line status bar. We need to tell *Term::Visual* what the status bar is going to look like. We do this with a set of *printf*-like format strings:

```
my $window_id = $vt->create_window(
    Window_Name => $0,
```

```

Status => { 0 =>
    { format => " [%8.8s] ",
      fields => [qw( time )],
    },
    1 => {
        format => " %s to: %s:%s",
        fields => [qw( status host port )]
    },
},
Buffer_Size => 1000,
History_Size => 50,
Title => "Protocol debugger" );

```

Unlike Linux, POE's event loop isn't preemptively multitasking; instead, sessions need to cede time to each other

This will set up one line to contain a little clock, of the form “[08:30 AM]”, and another line to tell us about the connection status—whether “connecting to somehost:1234” or “connected to somehost:1234.” It will also set the title line at the top of the screen, and set up buffers and history appropriately.

Now we’re ready to go. We need to create the main POE session that is going to handle all the action, and then we can run the POE kernel. Here’s what the main session looks like:

```

POE::Session->create
(inline_states =>
    { _start      => \&start_guts,
      update_time => \&update_time,

      connect_success => \&switch_wheels,
      connect_failure => sub {
          $vt->print($window_id, "An error occurred!: $_[ARG2]");
      }

      got_term_input => \&handle_term_input,
      got_packet    => sub { $vt->print($window_id, to_visual($_[ARG0])) },
    }
);

```

There aren’t that many events we need to cater for: First, when the kernel starts up, we need to start connecting to the remote server and kick off something to update the clock on the status bar. *start_guts* handles all this. Then there’s the event that gets fired every minute to actually do the clock update.

After we’ve established the connection, we need to turn our initial TCP socket into a filehandle (or rather, a wheel) that we can send data to and get data back from; the *connect_success* and *connect_failure* states deal with what happens here.

And when we’re up and running, if the user says something, we need to tell the server about it; similarly, if the server says something, we need to tell the user about it. This last state is so easy to handle, we do so on the spot: we run the data we’ve received from

the server through the *to_visual* routine discussed above, then tell the terminal object to display it on our main window.

Let’s start our analysis with the *_start* state. First, we need to tell *Term::Visual* how it should inform us about new input. We have to register a callback state so that it knows what to do with any input it receives. We do this by posting the name of a state to its *send_me_input* state. Our input handler state, as we’ve defined above, is called *got_term_input*, so that’s what we’ll send it:

```

sub start_guts {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    $kernel->post( interface => send_me_input => "got_term_input" );
}

```

We also call the *update_time* state; this will not only update the clock once, it will schedule itself to be called again every minute, keeping the clock updated:

```
$kernel->yield( "update_time" );
```

And finally, we connect to the remote server. We’ll first update the status bar to say something like “Connecting to myserver:1234”:

```

$vt->set_status_field( $window_id, host => $host,
    port => $port, status => "Connecting" );

```

And we create a new wheel to connect to the host; as before, we store this on the heap. As well as telling it what host and port to go to, we need to tell it what to do on success and failure; as before, these are the states we’ve named in the previous session constructor:

```

$heap->{connector} = POE::Wheel::SocketFactory->new
( RemoteAddress => $host,
  RemotePort    => $port,
  SuccessEvent  => 'connect_success',
  FailureEvent  => 'connect_failure',
);

```

Updating the time is pretty easy, although we’ll borrow some help from *POSIX*’s *strftime* function to handle the formatting for us:

```

sub update_time {
    use POSIX qw(strftime);
    $vt->set_status_field( $window_id,
        time => strftime("%I:%M %p", localtime) );
    $_[KERNEL]->alarm( update_time => int(time() / 60) * 60 + 60 );
}

```

POE’s *alarm* method will arrange for us to be called back at the beginning of the next minute.

When the connection happens, we need to create a new wheel that can be used to talk to the remote host. The *SocketFactory* wheel’s success state (*connect_success*) will be called with the TCP socket, and so we set up another wheel that communicates with that socket:

```

sub switch_wheels {
    my ($heap, $kernel, $connected_socket) = @_[HEAP, KERNEL, ARG0];
    delete $heap->{connector};
    $vt->set_status_field( $window_id, host => $host, port => $port,
        status => "Connected" );
    $heap->{socket_wheel} = POE::Wheel::ReadWrite->new
    ( Handle => $connected_socket,
      Driver => POE::Driver::SysRW->new,
      Filter => POE::Filter::Stream->new,
      InputEvent => 'got_packet',
    );
}

```

Again, we need to tell the new wheel what to do when it gets some data; specifying callback events is a big part of the POE I/O model. This is the *got_packet* event that was so simple to implement above.

The final event we need to handle is the input by the user. *Term::Visual* will call the callback event we registered, *got_term_input*, with any data received from the terminal. This is mildly complicated because the input can be in one of three forms. First, it can be an exception: the user hits “^C” or “^” and expects to be dumped out of the application. Thankfully, *Term::Visual* tells us about such an exception directly:

```
sub handle_term_input {
    my ($heap, $input, $exception) = @_ [HEAP, ARG0, ARG1];

    if (defined $exception) {
        warn "got exception: $exception";
        $vt->delete_window($window_id);
        exit;
    }
}
```

The second form is a command; we handle this just like an IRC client, with a command being defined as a forward slash at the start of the line, followed by a command word. At the moment, the only command we care about is *QUIT*, but it would be easy enough to extend the program to support a *CONNECT* command to connect to a different server:

```
if ($input =~ s{^/s*(\S+)\s*}{} ) {
    my $cmd = uc($1);
    if ($cmd eq 'QUIT') { $vt->delete_window($window_id); exit; }
    warn "Unknown command: $cmd";
    return;
}
```

Anything else we encode using the *to_binary* routine we previously detailed, and then send the result out to the server through the connection wheel. We also echo the output to the terminal, so the user can see what was said:

```
my $output = to_binary($input);
if ($output) {
    $heap->{socket_wheel}->put($output);
    $vt->print($window_id, ">>> ".to_visual($output)."\n");
}
```

And that's it! All our states are defined, all our bases are covered, and all we need to do is kick off the POE kernel and let the user talk to the server:

```
POE::Kernel->run;
```

The protocol debugger comes in at just over 150 lines of code, and swiftly became an invaluable tool in implementing the poker modules, as well as debugging several other network issues I came across. And, thanks to POE and *Term::Visual*, it has a friendly interface that makes it not too much of a burden to use, and was crafted up in little under an hour.

POE is a wonderfully flexible set of tools for developing event-based applications in Perl. There are a whole range of components like *Term::Visual*, which can take the pain out of programming servers and clients for all kinds of protocols. Take a look at the POE home page (<http://poe.perl.org>) to learn more.

TPJ

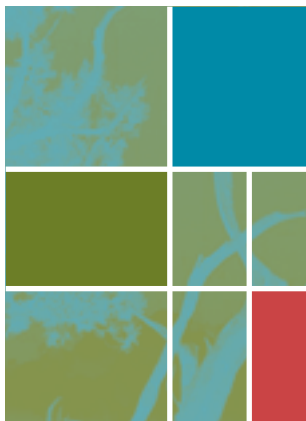
Subscribe now to

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/>



Blocking Spam with Postfix and Amavis

Randal L. Schwartz

Spam: What a mess. The wonderful *Mail::SpamAssassin* can catch most of it, looking at various spammy things in headers and bodies of messages, and even check external real-time block lists (RBLs) to help determine if a particular piece of e-mail is indeed a reasonable e-mail or simply someone getting a free ride to promote their own commercial activity. This month, I'd like to look at a recent change I made at the stonehenge.com mail server to help deal with the rapidly increasing amount of spam on the Internet.

Mail for the stonehenge.com domain is handled by blue.stonehenge.com, a server at a rack location provided by Sprocket Data (<http://sprocketdata.com/>). We're running OpenBSD and Postfix because I like to sleep at night and not worry about the "hack of the week."

Prior to the recent change, I had most of the mail for the stonehenge.com domain (with a few notable exceptions) be delivered to my personal merlyn account's mail, rewriting the destination to use the *plus* extended address format provided by Postfix. I accomplished this with an `/etc/postfix/virtual_regexp` entry that looked something like:

```
/^stonehenge\.com$/
  whatever
# other stonehenge.com rewrites are here
# final catch-all:
/^(.*)@stonehenge\.com$/
  merlyn+for-stonehenge+$1
```

I also included this line into `/etc/postfix/main.cf`:

```
virtual_maps = regexp:/etc/postfix/virtual_regexp
```

so that the virtual map was properly specified. As each e-mail came in, procmail would launch and consult my `.procmailrc` file, which looked something like:

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

```
LOGFILE=$HOME/.procmail.log
LOGABSTRACT=yes
## :0c
## $HOME/JustInCase/
:0w:
| Sortmail >>SORTMAIL.LOG 2>&1
LOG="... Sortmail failed, bouncing ..."
EXITCODE=75
:0
/dev/null
```

The key here is that each mail would be piped to my Sortmail program, but if anything went wrong, Postfix would retain the message in its own queue for a subsequent delivery. This saved my bacon more than once when I was editing Sortmail and forgot to check syntax before writing it out.

Within my Sortmail program, I extracted the very first "Delivered-To" header, and then undid the transformation applied by the virtual rewrite. This got me back to the original stonehenge.com address that had been requested.

I then constructed a *Mail::Internet* and *Mail::Audit* object from the incoming message:

```
my $mi = Mail::Internet->new(\*STDIN);
my $ma = Mail::Audit->new(data => [$mi->as_string =~ /(.*\n?)/g], noexit=> 1,
log => '-', loglevel => 2);
```

I did this because, although I started with *Mail::Audit*, I later found out it lacked some of the header-access functions that I needed, so I had to punt and use *Mail::Internet* instead. Eventually, I hope to eliminate *Mail::Audit* entirely, as I've found it to be too funky and ugly for my needs.

After sorting through the delivery address to determine how the message would be delivered or autoresponded, I started adding checks using *Mail::SpamAssassin* to try not to autorespond to spam or deliver spam into my significant inboxes. Anything addressed to me personally that was spammish got dropped into my "ube" folder (unsolicited bulk e-mail). Anything that was not addressed to me got dropped into my "ubetrap" file (as in a "spamtrap" address, but I always pronounced this rhyming with "boobytrap").

For any message that required an autoresponse, I eventually hooked in a Template Toolkit-based response template, passing the *Mail::Internet* object, the *Mail::Audit* object, and the constructed reply headers. A typical template looks like:

```
[%
    head.Subject = "Your recent message to $to\n";
    INCLUDE normal_header;
%]

Why did you send a message to [% to %]?

(It's very possible given the current sorry state of Microsoft
so-called operating systems security that your address has been
forged. If so, please ignore me. Sorry.)

Randal L. Schwartz
postmaster@stonehenge.com
[% INCLUDE signature %]
```

This template handles “bounces” (addresses that aren’t otherwise assigned within stonehenge.com). I send out a human message instead of a normal sendmail-like message because I need to know if they really intended the message for some other domain that was similar to stonehenge.com: It’s amazing how many of those are out there. Most people ignore sendmail-like messages, but they’ll respond in plain English to this letter.

There are other little parts to the mail system, but I hope you get the sense that it’s a bunch of bailing wire and duct tape, because it is. And it evolved slowly over time, starting out initially as procmailrc targets, then evolving to use the Perl-based Mail-Agent, and then to this bizarre hodgepodge.

Initially, this system worked rather well. I was dealing with about 300 to 500 pieces of e-mail a day, sorting them into mailing list folders, personal mail, autoresponse mail for our Perl Training services and my legal case, and handling comp.lang.perl.announce postings, and a few lightweight mailing list rebroadcasters. Each incoming message triggered just two forks (procmail, then my Perl sortmail), and then got delivered.

But around the summer of 2003, the various Microsoft virus mailers started hitting. They started sending mail from randomly selected addresses to many targets, carrying their DNA along to infect the next system on the list.

Right away, I noticed that I was getting a lot of “your mail contains a virus” mail, from well-intentioned antivirus programs. Let me make this perfectly clear. If you write an antivirus program, and your antivirus program can recognize that the virus fakes the “From” line, do not send a response to that clearly faked “From.” These “your mail contains a virus” mails are worse than the virus mails themselves, at least for me.

But what I also noticed was a steady increase in the MIRVs. I’m using MIRV here in the “multiple independently targetable reentry vehicles” sense. An incoming spam letter would be addressed to a number of “stonehenge.com” addresses, and delivered in one SMTP connection.

The trouble with MIRVs is that they get burst by the local Postfix and delivered as separate messages (although sharing one Message ID) to separate invocations of procmail, and then to separate invocations of my Sortmail. Each Sortmail would eventually get around to wondering if this was spam, and would make all the regex matches against the mail and all the RBL checks out on the Internet, and come to identical conclusions (usually “yes, it’s spam”).

So, each MIRV caused 20 new processes to fire up on my box in the space of about two seconds, beating up on a lot of memory and CPU as those complex regexen were dragged through the mail, and generating a lot of DNS Internet traffic to see about RBLs. Ugh. It was a nice design before MIRV spam, but clearly failing now.

But how to fix it? It wouldn’t be enough to move to *spamc/spamd*, which at least would remove the need to fork and reload all of that SpamAssassin code on each mail, because we still are asking the same question 10 times because of the MIRVs.

But luckily, I recently stumbled across a Slashdot posting that mentioned Amavis (A Mail Virus Scanner), found at <http://www.amavis.org/>. Unlike mail-user-agent (MUA) tools like *spamc* or simple *Mail::SpamAssassin*-based custom tools, I could hook Amavis in at the mail-transfer-agent (MTA) level. Ahh! Before the MIRV has burst! This looked very promising.

*Let me make this perfectly clear.
If you write an antivirus
program, and your antivirus
program can recognize that the
virus fakes the “From” line, do
not send a response to that
clearly faked “From”*

Even more promising is that Amavis is written in Perl, and uses *Spam::Assassin* and *Net::Server*, both technologies with which I was familiar. I figured that if I had any trouble with Amavis, my Perl skills were probably sufficient enough to either reverse-engineer for understanding or customize the needed features.

Although Amavis can be used as the normal port-25 listener on a server, I didn’t want to remove the known reliability and flexibility of having Postfix be my port-25 listener. Luckily, in the installation instructions, I saw how to make Amavis work alongside Postfix, and followed the instructions rather directly.

First, I unpacked Amavis into */opt/amavisd/*, and created an *etc* and *sbin* directory alongside the now unpacked source directory. I also created an *amavis* user, allowing the home directory to default to */home/amavis*.

Next, I copied *amavisd* to the *sbin* directory, and *amavisd.conf* to the *etc* directory. I edited the *amavisd.conf* file (putting it under RCS first) to reflect local preferences. Many of the settings were as recommended by the *README.postfix* file included with the distribution.

First, I fixed *\$MYHOME* to */home/amavis* and *\$mydomain* to stonehenge.com. (I like that the config file is in Perl and not some obscure config language.) Then I set *\$daemon_user* and *\$daemon_group* both to *amavis*, as I had chosen, and pushed *\$TEMP_BASE* into the *tmp* subdirectory.

Skimming down, I found the “POSTFIX” section, uncommenting the lines for *\$forward_method* and *\$notify_method* there. Finally, I uncommented the line that set *@bypass_virus_checks_acl* to a single period. Since I didn’t care about virus checks, and only about spam, I wanted to keep Amavis single minded.

I changed the *\$QUARANTINEDIR* to be below */home/amavis* for simplicity. And finally, I commented the *\$sa_local_tests_only* line, causing SpamAssassin to also consider the RBL tests.

After making all these changes, I then proceeded with the testing as indicated in the *README.postfix* file, double checking every step because my machine was handling live e-mail.

Ultimately, my `/etc/postfix/master.cf` was altered to comment out three lines:

```
#AMAVIS# smtp      inet n      -       -       -       -       smtpd
#AMAVIS# pickup    fifo n      -       -       60      1       pickup
#AMAVIS# cleanup    unix n      -       -       -       0       cleanup
```

replacing those with:

```
smtp      inet n      -       -       -       -       smtpd
-o cleanup_service_name=pre-cleanup
pickup    fifo n      -       -       60      1       pickup
-o cleanup_service_name=pre-cleanup
cleanup    unix  n      -       -       -       0       cleanup
-o mime_header_checks=
-o nested_header_checks=
-o body_checks=
-o header_checks=
```

and adding these new services as well:

```
pre-cleanup unix n      -       -       -       0       cleanup
-o virtual_alias_maps=
-o canonical_maps=
-o sender_canonical_maps=
-o recipient_canonical_maps=
-o masquerade_domains=

smtp-amavis unix -       -       y       -       2       smtp
-o smtp_data_done_timeout=1200
-o disable_dns_lookups=yes

127.0.0.1:10025 inet n      -       y       -       -       smtpd
-o content_filter=
-o local_recipient_maps=
-o relay_recipient_maps=
-o smtpd_restriction_classes=
-o smtpd_client_restrictions=
-o smtpd_helo_restrictions=
-o smtpd_sender_restrictions=
-o smtpd_recipient_restrictions=permit_mynetworks,reject
-o mynetworks=127.0.0.0/8
-o strict_rfc821_envelopes=yes
```

I also added the startup for `amavisd` to `/etc/rc.local`:

```
if [ -x /opt/amavisd/sbin/amavisd ]; then
    echo -n ' amavis ';
    sudo -u amavis /opt/amavisd/sbin/amavisd -c /opt/amavisd/etc/amavisd.conf
fi
fi;
```

But that's it. It's been working quite well for me, and my load average has been significantly less. Now the MIRVs get processed once instead of 10 times, and all is well. Almost immediately after making this change, Internet connections on the box were more stable, and my system didn't suddenly spike and freeze up my Emacs session nearly as much as it formerly did. And I'm not dealing with anywhere near the volume of identical spams, so my personal mail volume is also lower.

So, consider adding Amavis to your MTA, and help fight your neverending spam battle in an efficient way. Until next time, enjoy!

TPJ



Don't waste time
searching past *DDJ* issues
for your programming
solution...

Use *DDJ's* NEW CD-ROM, Release 14!

Save **TIME** and **MONEY** sorting
through your magazine
library of
Dr. Dobbs'.



ORDER TODAY!

www.ddj.com/cdrom/
800-444-4881

U.S. and Canada

785-841-1631 International
Fax: 785-841-2624
Dr. Dobbs' CD-ROM Library
1601 West 23rd Street, Suite 200
Lawrence, KS 66046-2703


CMP
United Business Media

Dr. Dobbs'

CD-ROM LIBRARY



Mastering Perl/Tk

Andy Lester

For a while, it seemed that we didn't really need a cross-platform GUI. The Macintosh was here, and Windows was there, and UNIX was its own weird self. Now, with Mac OS X gaining ground, especially among the Perl world, and the Panther release including X11, it may be time for Tk to fill the need. *Mastering Perl/Tk*, by Steve Lidie and Nancy Walsh, is a fine place to start. It's a replacement, update, and expansion of Walsh's earlier effort, *Learning Perl/Tk*.

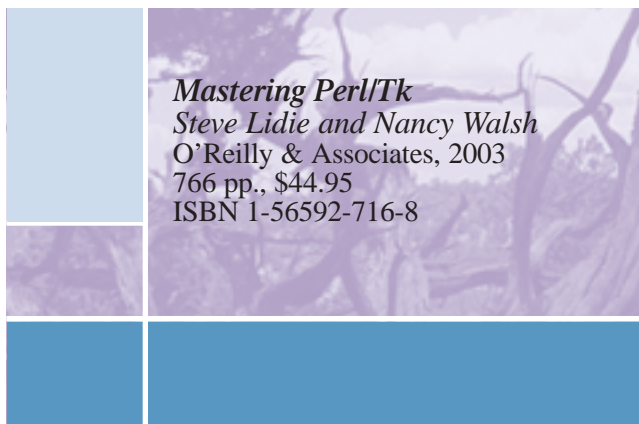
Chapter 1 takes an approach that I'd like to see more common in computer books that introduce a new subculture to the reader. It's sort of a crash course on what may be surprisingly different when writing Perl/Tk programs, such as: Indentation styles, how to debug a GUI program, naming conventions, and the proper way to end a program. It's good to know the methods to the madness before starting the journey.

After the introductions, half the book is devoted to the panoply of Perl/Tk widgets: text entry fields, scrollbars, frames, listboxes, menus, and so on. The progression of widgets is logically ordered and builds on previous chapters. Each widget is presented with a list of options available, including the default in bold. There's also a later chapter on the Tk Interface Extension widgets, a set of widgets that comes with the Tk module itself including TList, Tree, DirTree, HList, and TList—indispensable for any sort of hierarchy or filesystem navigation applications.

The chapter on geometry management explains each of the four different geometry managers (*pack*, *place*, *grid*, and *form*) in depth, discussing the differences between each, and when each is appropriate to use. The geometry manager is responsible for placing the widgets in a window, and can be a daunting learning experience to the Perl/Tk programmer, but the illustrations and examples are clear and instructive.

It's not surprising that Perl/Tk has a variety of different graphic capabilities, from custom cursors, to defining bitmaps in code, to tiling images on canvases, and more. I was disappointed, then,

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



that the entire graphics chapter was only 33 pages long. The 3D progress bar was cool, but I would have liked to have seen more examples of Tk being more than text.

Things start to get really interesting in the second half of the book, making the book much more than a rehash of the online docs. It starts off with a chapter on creating your own widgets in Perl. Every step of the process is included, from the idea to creating the framework of required subroutines, to filling in the code, to packaging and releasing your widget for CPAN distribution.

Unfortunately, I was disappointed to see the section on releasing and packaging your widget use the outdated *test.pl* single test program instead of a *t* directory, and the use of the *Test* module instead of the newer, more flexible *Test::More*.

Later in the book, there's a chapter on creating widgets in C. The sample widget is very simple, only drawing a square, but it's just as well. The intricacies of C widget creation are apparently far more torturous than in Perl, and the discussion of interfacing Perl and C is long enough as it is. As with the chapter on Perl widget creation, each step is explained, including having a completed distribution. For most readers, custom Perl widgets will be the way to go, but for those interfacing with existing third-party libraries, C extensions may be necessary.

The rest of the chapters are a grab bag, showing ways of using Perl/Tk with parts of the system that don't specifically relate to Tk. They're great for showing Tk as an actual application framework. The chapter on IPC shows parallel processing of the value of pi, the LWP chapter shows a simple program to fetch comics from web sites, and so on.

The appendices are a disappointment—almost an afterthought. They make up 18 percent of the book's page count and don't add anything very usable. Appendix A is a two-page treatment of in-

Half the book is devoted to the panoply of Perl/Tk widgets: text entry fields, scrollbars, frames, listboxes, menus, and so on

stalling both Perl and Perl/Tk on UNIX and Windows. This isn't worth the mention for one of the most notoriously hard-to-compile modules.

Appendix B is a listing of options and default values for each of the widgets. Unfortunately, it's entirely redundant to the package docs, printed in landscape, and there's a meaningless column called "Current value" populated with values like ARRAY(0x87d8b14). I would have preferred an aggregation of the clear, useful option listings from the chapters. Just because it's an appendix doesn't mean that it has to change the format of the information from earlier in the book.

Finally, Appendix C is a printout of all the code mentioned in the book proper. I hope that this doesn't signal a return to the bad old days of the computer book publishing boom when book thickness was everything, superfluous code listings padded the page count, and CD-ROMs contained archives of what was freely and easily available online.

Far more useful appendix content might have included a graphical map of the Perl/Tk widget class hierarchy, since every Perl/Tk widget is derived from the Tk base class and shares commonalities with other widgets. There's even a program in the chapter on building your own widgets that shows an example of a single widget's inheritance tree, so I'm baffled that there's not one in the appendix.

The book is put together well, and has the excellent O'Reilly layout and style. There are copious screenshots and figures throughout, making it clear how the widget will look onscreen. The figures are appropriately sized, unlike many books that, for example, print entire screens to illustrate one listbox. It was nice to see screens from Windows mixed with X11, each with a slightly different widget look.

Despite the criticisms, *Mastering Perl/Tk* is a fine book that I recommend for anyone interested in graphics programming with Perl. I only wonder how much of its \$44.95 cover price could have been shaved off if not for the superfluous appendices.

TPJ

Sign Up For BYTE.com!

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

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

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



**ONLY \$19.95
FOR ANNUAL
ACCESS!**



As a special thank you for signing up, receive the BYTE CD-ROM and a year of access for the low price of \$29.95!

Registering is easy...go to www.byte.com and sign up today! Don't delay!



BYTE

Source Code Appendix

Gavin Brown “Programming Graphical Applications with Gtk2-Perl, Part 2”

Listing 8

```
#!/usr/bin/perl
use Gtk2 -init;
use strict;

my $window = Gtk2::Window->new;
$window->set_title('Paned Example');
$window->signal_connect('delete_event', sub { Gtk2->main_quit });
$window->set_border_width(8);

my $vpaned = Gtk2::VPaned->new;
$vpaned->set_border_width(8);

$vpaned->add1(Gtk2::Button->new('Top Pane'));
$vpaned->add2(Gtk2::Button->new('bottom Pane'));

my $frame = Gtk2::Frame->new('Right Pane');
$frame->add($vpaned);

my $hpaned = Gtk2::HPaned->new;
$hpaned->set_border_width(8);

$hpaned->add1(Gtk2::Button->new('Left Pane'));
$hpaned->add2($frame);

$window->add($hpaned);

$window->show_all;

Gtk2->main;
```

Listing 9

```
#!/usr/bin/perl
use Gtk2 -init;
use strict;

my $window = Gtk2::Window->new;
$window->set_title('Scrolled Window Example');
$window->signal_connect('delete_event', sub { Gtk2->main_quit });
$window->set_border_width(8);

my $image = Gtk2::Image->new_from_file('butterfly.jpg');

my $scrwin = Gtk2::ScrolledWindow->new;
$scrwin->set_policy('automatic', 'automatic');

$scrwin->add_with_viewport($image);

$window->add($scrwin);

$window->show_all;

Gtk2->main;
```

Listing 10

```
#!/usr/bin/perl
use Gtk2 -init;
use strict;

my $window = Gtk2::Window->new;
$window->set_title('Event Box Example');
$window->signal_connect('delete_event', sub { Gtk2->main_quit });
$window->set_border_width(8);

my $image = Gtk2::Image->new_from_stock('gtk-dialog-info', 'dialog');

# a Gtk2::Image widget can't have the 'clicked' signal, so this causes
# an error:
$image->signal_connect('clicked', \&clicked);

my $box = Gtk2::EventBox->new;

# but this works!
$box->signal_connect('button_release_event', \&clicked);

$box->add($image);

$window->add($box);

$window->show_all;
```

```

Gtk2->main;

sub clicked {
    print "someone clicked me!\n";
}

```

Listing 11

```

#!/usr/bin/perl
use Gtk2 -init;
use Gtk2::SimpleList;
use strict;

my $window = Gtk2::Window->new;
$window->set_title('List Example');
$window->set_default_size(300, 200);
$window->signal_connect('delete_event', sub { Gtk2->main_quit });
$window->set_border_width(8);

my $list = Gtk2::SimpleList->new(
    'Browser Name' => 'text',
    'Version'      => 'text',
    'Uses Gecko?'  => 'bool',
    'Icon'         => 'pixbuf'
);

$list->set_column_editable(0, 1);

@{$list->{data}} = (
    [ 'Epiphany', '1.0.4', 1, Gtk2::Gdk::Pixbuf->new_from_file('epiphany.png') ],
    [ 'Galeon',   '1.3.9', 1, Gtk2::Gdk::Pixbuf->new_from_file('galeon.png') ],
    [ 'Konqueror', '3.1.4', 0, Gtk2::Gdk::Pixbuf->new_from_file('konqueror.png') ],
);

my $scrwin = Gtk2::ScrolledWindow->new;
$scrwin->set_policy('automatic', 'automatic');

$scrwin->add_with_viewport($list);

my $button = Gtk2::Button->new_from_stock('gtk-ok');
$button->signal_connect('clicked', \&clicked);

my $vbox = Gtk2::VBox->new;
$vbox->set_spacing(8);

$vbox->pack_start($scrwin, 1, 1, 0);
$vbox->pack_start($button, 0, 1, 0);

$window->add($vbox);

$window->show_all;

Gtk2->main;

sub clicked {
    my $selection = ($list->get_selected_indices)[0];
    my @row = @{$list->{data}}[$selection];
    printf(
        "You selected %s, which is at version %s and %s Gecko.\n",
        $row[0],
        $row[1],
        ($row[2] == 1 ? "uses" : "doesn't use")
    );
    Gtk2->main_quit;
}

```

Listing 12

```

#!/usr/bin/perl
use Gtk2 -init;
use Gtk2::SimpleMenu;
use strict;

my $window = Gtk2::Window->new;
$window->set_title('Menu Example');
$window->set_default_size(300, 200);
$window->signal_connect('delete_event', sub { Gtk2->main_quit });

my $vbox = Gtk2::VBox->new;

my $menu_tree = [
    _File => {
        item_type => '<Branch>',
        children => [
            _New => {
                item_type => '<StockItem>',
                callback   => \&new_document,
                accelerator => '<ctrl>N',
                extra_data  => 'gtk-new',
            },

```

```

        _Save => {
            item_type      => '<StockItem>',
            callback        => \&save_document,
            accelerator     => '<ctrl>S',
            extra_data      => 'gtk-save',
        },
        _Quit => {
            item_type      => '<StockItem>',
            callback        => sub { Gtk2->main_quit },
            accelerator     => '<ctrl>Q',
            extra_data      => 'gtk-quit',
        },
    ],
},
_Edit => {
    item_type => '<Branch>',
    children => [
        _Cut => {
            item_type      => '<StockItem>',
            callback_action => 0,
            accelerator     => '<ctrl>X',
            extra_data      => 'gtk-cut',
        },
        _Copy => {
            item_type      => '<StockItem>',
            callback_action => 1,
            accelerator     => '<ctrl>C',
            extra_data      => 'gtk-copy',
        },
        _Paste => {
            item_type      => '<StockItem>',
            callback_action => 2,
            accelerator     => '<ctrl>V',
            extra_data      => 'gtk-paste',
        },
    ],
},
_Help => {
    item_type => '<LastBranch>',
    children => [
        _Help => {
            item_type      => '<StockItem>',
            callback_action => 3,
            accelerator     => '<ctrl>H',
            extra_data      => 'gtk-help',
        }
    ],
},
];

my $menu = Gtk2::SimpleMenu->new (
    menu_tree      => $menu_tree,
    default_callback => \&default_callback,
);

$ vbox->pack_start($menu->{widget}, 0, 0, 0);
$ vbox->pack_start(Gtk2::Label->new('Rest of application here'), 1, 1, 0);

$window->add($vbox);

$window->show_all;

Gtk2->main;

sub new_document {
    print "user wants a new document.\n";
}

sub save_document {
    print "user wants to save.\n";
}

sub default_callback {
    my (undef, $callback_action, $menu_item) = @_;
    print "callback action number $callback_action\n";
}

```

Listing 13

```

#!/usr/bin/perl
use Gtk2 -init;
use strict;

my $window = Gtk2::Window->new;
$window->set_title('Custom Widget Example');
$window->set_default_size(150, 100);
$window->signal_connect('delete_event', sub { Gtk2->main_quit });
$window->set_border_width(8);

my $button = Gtk2::FunnyButton->new(

```



```

        normal => 'Click me!',
        over   => 'Go on, click me!',
        clicked => 'Click me harder!',
    );

    $window->add($button);

    $window->show_all;

    Gtk2->main;

package Gtk2::FunnyButton;
use base 'Gtk2::Button';
use strict;

sub new {
    my ($package, %args) = @_;
    my $self = $package->SUPER::new;
    bless($self, $package);
    $self->{labels} = \%args;
    $self->add(Gtk2::Label->new($self->{labels}{'normal'}));
    $self->signal_connect('enter_notify_event', sub { $self->child->set_text($self->{labels}{'over'}) });
    $self->signal_connect('leave_notify_event', sub { $self->child->set_text($self->{labels}{'normal'}) });
    $self->signal_connect('clicked', sub { $self->child->set_text($self->{labels}{'clicked'}) });
    return $self;
}

```

Julius C. Duque “Finding Duplicate Files”

Listing 1

```

1  #!/usr/local/bin/perl
2  use diagnostics;
3  use strict;
4  use warnings;
5  use Digest::MD5;
6
7  while (<>) {
8      chomp;
9      if (-f) {
10         open INFILE, $_;
11         my $context = new Digest::MD5;
12         $context->addfile(*INFILE);
13         my $digest = $context->hexdigest;
14         close INFILE;
15         print "$digest $_\n";
16     }
17 }

```

Listing 2

```

1  #!/usr/local/bin/perl
2  use diagnostics;
3  use strict;
4  use warnings;
5  use Getopt::Long;
6
7  my %dups = ();
8  my ($digest, $filename, $verbose) = ();
9  GetOptions("verbose" => \$verbose);
10
11 while (<>) {
12     chomp;
13     ($digest, $filename) = /(.{32})(.*)/;
14     push(@{$dups{$digest}}, "\n");
15     push(@{$dups{$digest}}, $filename);
16 }
17
18 foreach $digest (sort keys %dups) {
19     if (scalar(@{$dups{$digest}}) > 2) {
20         print "$digest" if ($verbose);
21         print "@{$dups{$digest}}\n";
22         print "\n" if ($verbose);
23     }
24 }

```