

The Perl Journal

Programming Graphical Applications with Gtk2-Perl, Part 1

Gavin Brown • 3

Automated Testing with the Perl *Test::* Modules

Andy Lester • 10

Quantifying Popular Programming Languages

Thomas Plum • 14

Perl Poker

Simon Cozens • 15

Free as in Music

Randal Schwartz • 19

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Jack J. Woehr:

Linux and the UNIX Philosophy • 21

Source Code Appendix • 23

LETTER FROM THE EDITOR

Embrace the Interface

Let's face it. Most of the time, as Perl developers, we shun GUIs. For the majority of Perl work, we just don't want to bother with it. One reason is that the old 80/20 rule seems to apply here—the interface represents 20 percent of a program's functionality, yet takes 80 percent of the development effort. And, of course, there are entire realms of Perl programming where a GUI is not just undesirable, it's downright nonsensical.

But it's also true that the lack of an easy way to provide such an interface can get you in a rut. If there's no easy way to do it in Perl, it's tempting to turn to Python or Java (well, tempting for *some* of us). This is a surefire way to marginalize Perl, relegating it to an endless series of CGI scripts and other back-end tasks.

Thankfully, we do have ways of adding some interface-widget goodness to our Perl creations. In the first installment of a two-part article on Gtk2-Perl, Gavin Brown shares a set of bindings for the Gtk+ 2.x libraries. With it, you can use object-oriented Perl to write event-loop-based Gtk and Gnome apps. Maybe it will even inspire you to try your hand at coding a slightly richer interface for your Perl apps.

◆◆◆

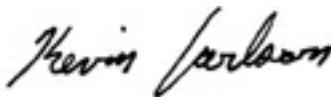
Also in this issue, you'll find an article by Thomas Plum detailing the results of a study he and his colleagues did of software job offers for the 12 months beginning July 2002, analyzing the job offers by programming-language requirements. Roughly half the offers during that time period make mention of C++, while a little over 40 percent make mention of Java. C, of course, makes a strong showing, as does Visual Basic.

Perl's numbers come in around the 12-percent mark and to me, this is where it gets interesting. Is it possible that only 12 percent of software jobs involve significant amounts of Perl coding? I doubt it. I bet the real number is much higher, and that Perl is just exhibiting its accustomed stealthy ubiquity. Here's the number I really want to know: What percentage of job offers make absolutely no mention of Perl, yet result in jobs (such as sys admins) that ultimately require a great deal of Perl coding to get work done in the real world? Maybe someday we'll have an answer to that question.

◆◆◆

At the end of September, UK-based antispam company Sophos inspired a collective sharp intake of breath among Windows Perl developers when it announced it had acquired ActiveState. For Perl coders trying to deliver the goods in a Microsoft-dominated desktop market, ActiveState was a rock, providing solid tools and building a loyal user base. Basically, ActiveState *is* Perl in the Windows world.

We've all seen acquisitions go sour. Sometimes the buyer only wants one piece of the pie, and the rest gets discarded. Or sometimes the buyer fails utterly to understand the acquired company's market or products. But from all accounts, that's not happening with Sophos and ActiveState. If the statements from both companies and the chatter on ActiveState's mailing lists are to be believed, nothing really will change. Sophos has expressed a commitment to all of ActiveState's current product lines, and is retaining all of ActiveState's staff. Here's hoping that the lovefest continues.



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

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, HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED

TECHNOLOGIES Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER NETWORK

COMPUTING MEDIA NETWORK Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE

DEVELOPMENT MEDIA Peter Westerman

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES

Marie Myers

Perl News

Perl Releases

Perl 5.8.1 is still hot from the oyster, but the language developers haven't taken a break—Nicholas Clark has posted the first release candidate for Perl 5.8.2 to CPAN (<http://www.cpan.org/authors/id/N/NW/NWCLARK/perl-5.8.2-RC1.tar.bz2>). Perl 5.8.2 should restore binary compatibility with modules compiled under earlier versions. (Binary compatibility was broken in Perl 5.8.1 due to the new hash randomization security feature). Nicholas also asks testers to check “how scripts that hash large amounts of data behave,” “whether OS X and Solaris build all XS modules correctly now,” “is mod-perl still happy,” “does PAR work smoothly,” and whether it still builds on Windows, VMS, OS/2, and other platforms.

Also, Hugo van der Sanden has posted Perl 5.9.0—the first shoot of a new development branch—to <http://www.cpan.org/authors/id/H/HV/HVDS>.

Sophos Acquires ActiveState

At the end of September (just after the October issue of TPJ was wrapped up), the antivirus company Sophos announced that it had bought ActiveState for \$23 million. ActiveState has been focusing on the antispam arena since the release of its PerlMx mail filtering solution—later rebranded PureMessage—in May 2002; now Sophos PureMessage integrates the spam filters with Sophos' antivirus tools. However, Sophos has also stated that it “is committed to supporting and extending ActiveState's involvement in the open-source community. ActiveState's programming tools, language distributions, and support services will continue to be developed, supported, and marketed under the ActiveState name, in exactly the same way they have until now.” ActiveState now operates as a division of Sophos, and all of the ActiveState employees have been retained.

Perl Foundation Grants Detailed

The Perl Foundation has released details of all its latest grants, except for the one awarded to Larry Wall. Pete Sergeant gets \$4000 to work on a suite of RTF parsing modules, including *RTF::Tokenizer*, *RTF::Parser*, *RTF::Reader*, and *RTF::Tree-Builder*; he'll post updates to <http://rtf.perl.org/>. Daniel Grunblatt's \$6000 grant will go toward porting the Parrot JIT to the IA-64 and HPPA architectures, and continuing its development on ARM, alpha, PPC, i386, and SPARC. Autrijus Tang will use his \$2000 to focus on Slash, the content-management system that powers Slashdot and use.perl.org—he'll work on internationalization and ease of configuration. Lastly, Leopold Toetsch has been awarded \$4000 to get the Parrot packfile code to support multiple code segments.

Perl Worldwide

Uriel Lizama has launched a new site at <http://perlenespanol.baboonsoftware.com/historia/>, and is busy adding Spanish-language Perl tutorials and forums. Meanwhile, “monsieur_champs” has created a new Perl Monks node (http://www.perlmonks.org/index.pl?node_id=290138), indexing articles that have been translated to Portuguese. Lastly, the dates have been set for the 2004 YAPC North America conference, which will take place in Buffalo, NY, June 16–18 (see <http://www.yapc.org/America/> for more details).

All Right Kid, You're In

German coder Edi Weitz has cooked up a utility for Linux and Windows called “The Regex Coach,” which will graphically walk through and test Perl-compatible regular expressions. It can single-step through the regex's matching process, and shows which parts of a target string correspond to parts of the regular expression. It can also simulate Perl's `split` and `s///` (substitution) operators, and will create a graphic of the regular expression's parse tree.

The Regex Coach is actually written in Lisp, using the Perl-compatible CL-PPCRE regex engine (also written by Weitz). It's free for private or noncommercial use, and Weitz adds: “If you're, say, just using The Regex Coach during your work hours to test regular expressions that's fine for me—you don't have to ask for a commercial license, just use it.” You can download the program from <http://www.weitz.de/regex-coach/>.

eXtreme Perl

The Perl XP Training Project, launched by chromatic, aims to provide free training in “how Extreme Programming works, how to write tests in Perl, or how to contribute to multiuser programming projects”—and, incidentally, to get some real work done. In the project, chromatic takes the role of a customer and provides feature requests (or “story cards,” in the XP world); each one is designed to be simple, requiring only an hour or two of work to implement. The developers get feedback and testing for their work, and a chance to ask questions and receive guidance; meanwhile, as their skills develop, they'll be put to work on practical tasks like the Phalanx project.

The project web site is <http://xptrain.perl-cw.com/>. chromatic writes: “It's okay if you're just getting started with XP, testing, writing modules, or Perl. As long as you're willing to try, to learn, and to accept and to learn from feedback, you're welcome. It's also okay if you can only contribute one story a week—or just one story ever.”

Programming Graphical Applications with Gtk2-Perl, Part 1

Gtk+ is a toolkit for creating graphical applications on X11 systems. It has also been ported to Windows, 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 (GNU Network Object Model Environment).

Perl already has mature, stable, and popular bindings for the 1.x series of Gtk+ and for the GNOME libraries. However, bindings for the 2.x series of Gtk+ reached the 1.0 version in October, so now is a good time to talk about the things that you can do with the new libraries.

This article will be comprised of two parts: This month, I'll discuss some basics of event-based programming and introduce you to widgets and containers in Gtk+. Next month, I'll discuss manipulating graphics, introduce the Rapid Application Development (RAD) tool Glade, and show you how to create your own widgets.

What's New in Gtk+ 2

Gtk+ 2.x builds on the old 1.x series with a number of improvements. These include support for font antialiasing using Xft, better font management using Fontconfig, integrated and themeable stock icons, and an advanced Model-View-Controller (MVC) model for list and tree widgets. All of these new features are available to Gtk2-Perl.

Some Basic Principles

Gtk+ employs some programming paradigms that can be a bit confusing to people who've never done GUI programming before. So before we go too far, I think it's a good idea to cover them, if only briefly.

Event-Based Programming

At the heart of GUI programming is the notion of event-based programming. Unlike writing programs for the console or the Web, your program does not run sequentially—some parts of the code will be run when the program starts, but some may not be executed until an event occurs or a signal is emitted by a widget. This may be the user clicking on a button or pressing a key or

moving the mouse. Events can also be triggered by messages from the window manager telling the application to quit. Or the user could be performing a drag-and-drop action from another application.

As a result, applications using Gtk2-Perl tend to follow this structure:

- Build the basic interface.
- Set up callbacks for various different events and signals.
- Enter a main loop during which the application listens for events and signals and executes the defined callbacks.

The main loop itself is provided by the Gtk+ libraries, so as a rule, there's very little need to create one yourself.

Widgets and Containers

Widgets are graphical objects that the user can interact with. A widget might be a button or an icon, a menu or a scrollbar. Gtk+ includes a large palette of widgets that perform all kinds of tasks.

In order to organize these widgets on screen, and to establish the relationships between them, containers are used to group widgets together. For example, most applications will have a toolbar with icons for different functions. These icons are widgets and are grouped together by a container widget; namely, the toolbar itself. The toolbar is also contained within the top-level window. A large number of widgets are also containers; for example, a button can contain a simple text label or an icon.

In Gtk2-Perl, all widgets are represented as Perl objects. In fact, they're also blessed hash references so that you can do inheritance and extensions and create your own custom widgets (more on that next month).

Hello World

Listing 1 is a simple "Hello World" program, using the Gtk2-Perl modules. The application that this program produces is shown in Figure 1.

The first line after the shebang loads the Gtk2 module. The `-init` argument after the module name is a convenience—it calls the `Gtk2->init` function so that you don't have to later on.

The `$quit` variable contains a reference to an anonymous subroutine. These references are used extensively in Gtk2-Perl as callbacks that are executed when an interface event occurs. Line 5 in Listing 1 is equivalent to

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.


```
sub quit {
    exit;
}

my $quit = \&quit;
```

Similarly, line 8 could be changed to

```
$window->signal_connect('delete_event', \&quit);
```

The rest of the program is involved with the creation of widgets, setting their properties, building up interface by placing widgets inside other widgets, and finally displaying the application on the screen.

Gtk+ now directly supports a wide range of image formats and features

Lines 7–12 create a top-level window widget that will contain all the other widgets, and then sets various properties of that widget. Line 14 creates a simple text label. Lines 16 and 17 create a button, and set a callback to use when the button is clicked. Lines 19–23 create a special container called a “packing box” that is used to group widgets. In this case, it is a vertical packing box, so the widgets are stacked one below the other (a horizontal packing box is also available).

Finally, on lines 27 and 29, the widgets are rendered on screen and the main Gtk+ loop is called. Without this first line, nothing would appear on screen. Without the second, the UI would be unresponsive, as it would not respond to any events that occur.

Controlling Program Flow

A vital component of event-based programming is the concept of a program loop. All GUI systems have such a loop—during each iteration, the program checks for any events and responds to them accordingly.

You can control the Gtk+ main loop using the following functions:

```
Gtk2->main;
```

This starts the main loop. Sequentially, your program pauses at this point in the program until the main loop ends.

```
Gtk2->main_quit;
```

This tells the main loop to end. Your program will then continue as before.

There are some situations where you might want to perform an iteration outside of the main loop—for example, while your program is reading or writing a filehandle or socket, or some other process that would otherwise freeze your user interface. In such situations, you can use this code to “catch up” on any unhandled events:

```
Gtk2->main_iteration while (Gtk2->events_pending);
```

The `Gtk2->main_iteration` function does the job of checking for and responding to any events that might have cropped up since the last check. Unsurprisingly, `Gtk2->events_pending` returns a true value if there are any events that haven’t been handled.

Basic Widgets and Their Properties

Widgets all have a common set of methods that they inherit from their base class, `Gtk2::Widget`. There are far too many widgets and properties to discuss here, but the most important ones are worth taking a look at. Full information about the available widgets can be found in the Gtk+ API documentation and the Gtk-Perl tutorial; see the the “More Information” section at the end of the article.

Common Widget Methods

```
my $widget = Gtk2::Widget->new;
```

Since all Gtk+ widgets are Perl objects, they need to be constructed and, as expected, the `new()` method is the constructor. Most widgets can be created without any arguments, but a few will require various parameters to be set when they’re constructed. If you aren’t sure, use the `new()` method without arguments and run your program—if an argument is missing, you will be told.

```
$widget->show;
$widget->show_all;
$widget->hide;
$widget->hide_all;
```

A widget has to be rendered onto the display before it can be used—the `show()` method does this rendering. Every widget has to be shown, so the `show_all()` method is handy since it calls `show()` on the widget and all the widget’s children. The `hide()` and `hide_all()` methods do the reverse—they remove a widget from the screen.

```
my $handler_id = $widget->signal_connect($signal, $callback, \@callback_data);
$widget->signal_disconnect($handler_id);
```

This method attaches the supplied callback reference to the given signal for the widget. There are a wide range of signals and events available—again, too many to discuss in this article. The `signal_disconnect()` method detaches the callback identified by the handler ID.

```
my $container = $widget->get_parent;
```

This method returns the parent widget of the widget—for example, in Listing 1, using `get_parent()` on the `$vbox` container would return `$window`.



Figure 1: “Hello World” program using the Gtk2-Perl modules.

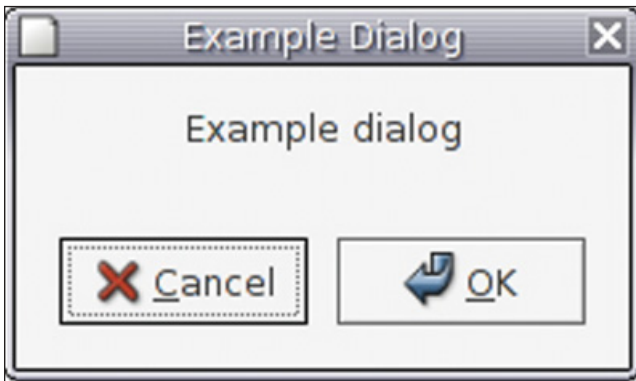


Figure 2: Gtk2::Dialog in action.

Windows

Window widgets come in two types: top-level windows (*Gtk2::Window*), which contain the application's main interface; and dialogs (*Gtk2::Dialog*) that appear occasionally; for example, to ask the user a question or display configuration settings.

To create a top-level window, use the following:

```
my $window = Gtk2::Window->new;
```

A top-level window is a container, so you can use the *add()* method to place another widget inside it. This method is common to all container widgets (more on this in the “Containers” section).

Dialogs are similar to normal windows, but come with some extra bits and pieces to save time. Listing 2 is a simple example of *Gtk2::Dialog* usage. The resulting window is shown in Figure 2.

Dialog windows come prepackaged with an *action area* for buttons, and a vertical packing box for widgets (the *vbox()* method of the dialog returns the packing box). You can add buttons to the action area using the *add_buttons()* method, using a hash containing the label or stock ID for each button, and the response code. The *response* signal can be used to determine what to do when the dialog is closed or when one of the buttons is clicked. For example, if the user presses the *close* button on the window frame, the response code is *delete-event*, and we've told the dialog to use *cancel* for the “cancel” button and *ok* for “OK.”

Labels

```
my $label = Gtk2::Label->new('This is a label.');
```

```
$label->set_text('Ain't it neat?!');
```

```
my $text = $label->get_text;
```

Labels are simple, rectangular widgets containing text. This text can be formatted in a number of ways. Gtk+ uses a system called “Pango” to format text, and you can use a simple HTML-like markup language to add formatting to your text labels (see Listing 3). The resulting window is shown in Figure 3.

Images

A real headache with the old Gtk+ 1.x series was the complexity involved in displaying images. In version 2.x, image support has been vastly improved. Gtk+ now directly supports a wide range of image formats and features, including GIF animation and PNG's alpha channels.

```
my $image = Gtk2::Image->new_from_file('icon2.png');
```

```
$image->set_from_file('icon2.png');
```

```
$image->set_from_animation(Gtk2::Gdk::PixbufAnimation
```

```
->new_from_file('animation.gif', my $error));
```

Additionally, you can access Gtk+'s wide range of stock icons:

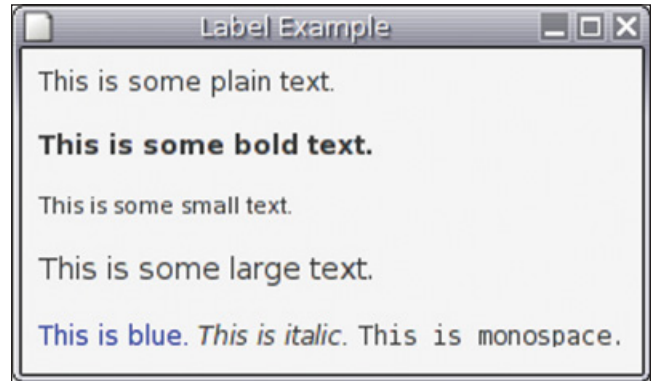


Figure 3: Formatting label text using the Pango system.

```
my $image = Gtk2::Image->new_from_stock('gtk-quit');
```

```
$image->set_from_stock('gtk-ok');
```

A complete list of all the stock icons can be found in the Gtk+ reference (see “More Information” for links).

Buttons

Buttons can be created in a number of ways. First, you can call the constructor with a string argument as a text label:

```
my $button = Gtk2::Button->new('Click Me!');
```

or you can request a stock button:

```
my $button = Gtk2::Button->new_from_stock('gtk-ok');
```

The button will use the appropriate stock icon and use the standard label for that icon (e.g., “OK” or “Quit”). This label is internationalized automatically based on the user's language settings.

To use your own child widget, simply call the constructor without arguments:

```
my $button = Gtk2::Button->new;
```

```
$button->add($widget);
```

Most of the time, you'll want to use the *clicked* signal to connect a callback to a button, like so:

```
$button->signal_connect('clicked', \&clicked);
```

```
sub clicked {
```

```
    my $button = shift;
```

```
    print "button was clicked\n";
```

```
}
```

Text Inputs

This creates a simple, single-line text input:

```
my $entry = Gtk2::Entry->new;
```

```
$entry->set_text('Enter your name here.');
```

```
my $text = $entry->get_text;
```

You can connect the *activate* signal to a *Gtk2::Entry* widget to catch when the user presses the Enter key.

Containers

Containers are widgets that can hold other widgets. They are vital for creating an organized interface because they define the relationships between elements on screen.

Common Container Methods

```
$container->add($widget);  
$container->remove($widget);
```

Most containers support the *add()* and *remove()* methods. They simply add a widget to a container or remove it. In general, there are two kinds of containers: Those that can hold just a single widget (for example, a button) and are derived from the *Gtk2Bin* base widget; and those that can hold more than one. The *add()* and *remove()* methods are common to this first kind. Multiple containers usually have their own methods for adding widgets.

```
my @child_widgets = $container->get_children;
```

This returns an array of all the widgets that are held by a widget. Single-widget containers also have the *child()* method.

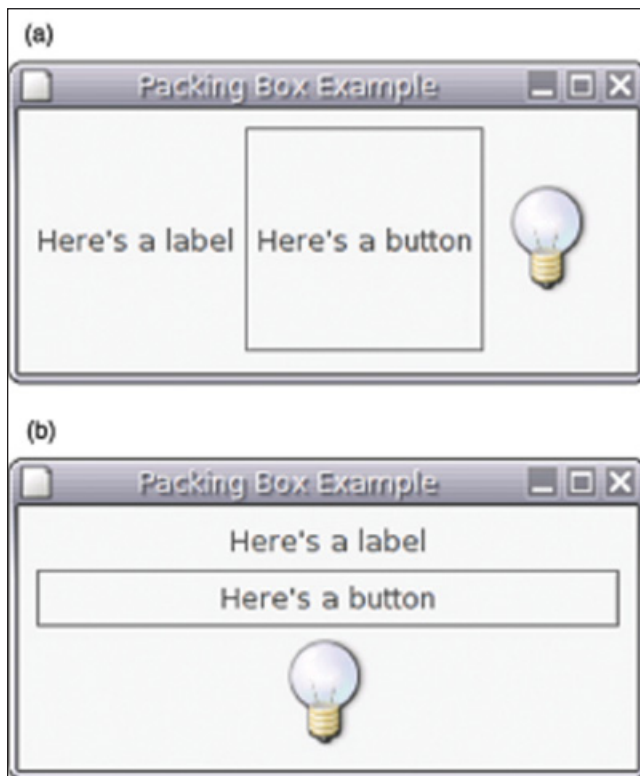


Figure 4: (a)Horizontal packing box; (b)vertical packing box.

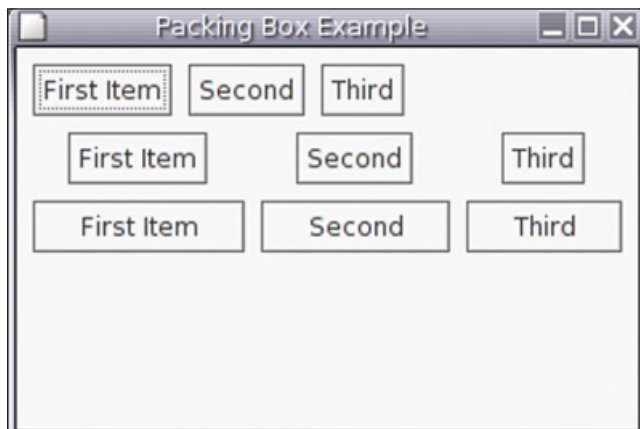


Figure 5: Three different packing modes.

Packing Boxes

Packing boxes allow you to align widgets in horizontal rows or vertical columns. They are among the most useful containers—you will probably find that your applications will make heavy use of them. Listing 4 demonstrates the use of packing boxes and produces the output in Figure 4.

When the user clicks the button, the horizontal packing box is removed from the window and is replaced with a vertical packing box.

```
$box->pack_start($widget, $expand, $fill, $padding);  
$box->pack_end($widget, $expand, $fill, $padding);
```

To add a widget to a packing box, use either *pack_start()* or *pack_end()*. The former places the widget in the next slot to the left (or top, in the case of a vertical packing box); the latter places the widget into the next slot to the right (or bottom).

The remaining three arguments define the widget's appearance. Setting *\$expand* to 1 (or any true value) will cause the packing box to grow to fill all the space available. If *\$fill* is true, then the widgets inside the packing box are expanded to fill all the space available. The *\$padding* variable determines the padding (in pixels) between widgets.

Listing 5 demonstrates the different packing modes and generates the layout seen in Figure 5. In the first row, the packing box is shrunk to fit the size of the buttons. In the second, the packing box is expanded but the buttons are spread out. In the third, the buttons are expanded to fill the available size.

Tables

Tables are similar to packing boxes, but they add a second dimension. Widgets can be arranged in rows and in columns. The table size, however, is fixed—while you can add as many widgets to a packing box as you like, with a table, you must define the number of rows and columns in the table when you create it.

Listing 6 shows you how tables work. Since they are two-dimensional, you can attach widgets in many more ways. The corresponding Figure 6 shows widgets that span multiple rows and columns.

When you create a table, you must specify its size:

```
my $table = Gtk2::Table->new(6, 5);
```

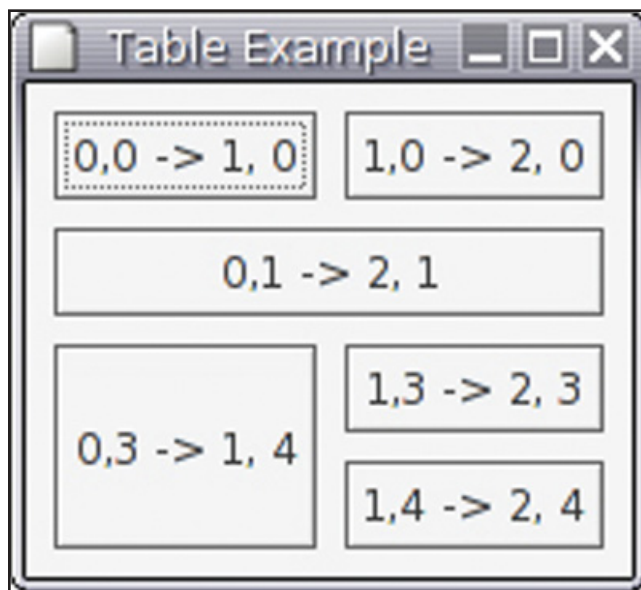


Figure 6: Widgets spanning multiple table rows and columns.

The above code creates a table six rows high and five columns wide. If you try to insert a widget outside this grid, you will get an error.

The `attach_defaults()` method is used to add widgets to a table. There is also `attach()`, which gives you more fine control over the way widgets appear within the table—see the “More Information” section for details.

```
$table->attach_defaults($widget, $left_coord, $right_coord, $top_coord,  
                        $bottom_coord);
```

The four arguments to `attach_defaults()` after the widget are the coordinates of the left-, right-, top-, and bottom-most edges of the widget. The table grid starts in the top left at “0, 0” and extends to `$x-1`, `$y-1`, where `$x` and `$y` are the dimensions specified when you create the table.

Frames

Frames create a border around a widget with a text label as a title. Frames can only contain a single widget. An example of frame usage is shown in Listing 7, and the resulting window is shown in Figure 7.

The frame style can be controlled with the `set_shadow_type()` method. Allowed values are “in,” “etched_in,” “out,” and “etched_out.”

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. Also, check out Part 2 of this article next month for more on creating your own widgets.

Gtk+ Documentation

There is currently no 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 the Gtk+ widget set, the *Glib* general purpose utility library, *GdkPixbuf*, and *Pango*.

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 at <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.

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.

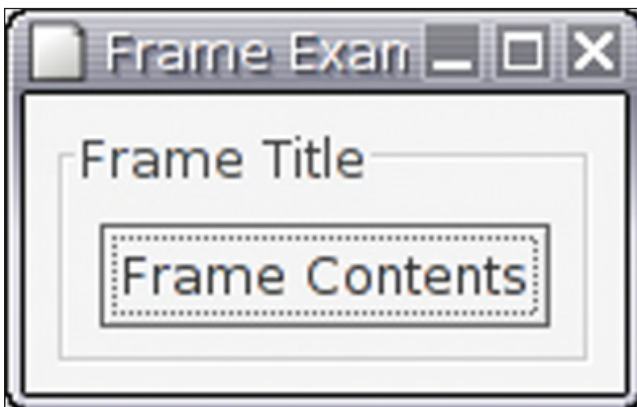


Figure 7: Creating a border around a widget using a frame.

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

Listing 1

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

my $quit = sub { exit };

my $window = Gtk2::Window->new;
$window->signal_connect('delete_event', $quit);
$window->set_position('center');
$window->set_border_width(8);
$window->set_title('Hello World!');
$window->set_default_size(200, 100);

my $label = Gtk2::Label->new('Hello World!');

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

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

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

$window->add($vbox);

$window->show_all;

Gtk2->main;
```

Listing 2

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

my $dialog = Gtk2::Dialog->new;

$dialog->set_title('Example Dialog');
$dialog->set_border_width(8);
$dialog->vbox->set_spacing(8);
$dialog->set_position('center');
$dialog->set_default_size(200, 100);

$dialog->add_buttons(
    'gtk-cancel' => 'cancel',
    'gtk-ok' => 'ok'
);

$dialog->signal_connect('response', \&response);

my $label = Gtk2::Label->new('Example dialog');

$dialog->vbox->pack_start($label, 1, 1, 0);

$dialog->show_all;

$dialog->run;

sub response {
    my ($button, $response) = @_;
    print "response code is $response\n";
    exit;
}
```

Listing 3

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

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

my $label = Gtk2::Label->new;

my $markup = <<"END";
This is some plain text.

<b>This is some bold text.</b>

<span size="small">This is some small text.</span>

<span size="large">This is some large text.</span>
```

```
<span foreground="blue">This is blue.</span> <span style="italic">This is
italic.</span> <tt>This is monospace.</tt>
END
```

```
$label->set_markup($markup);

$window->add($label);

$window->show_all;

Gtk2->main;
```

exit;

Listing 4

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

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

my $label = Gtk2::Label->new("Here's a label");
my $button = Gtk2::Button->new("Here's a button");
my $image = Gtk2::Image->new_from_stock('gtk-dialog-info', 'dialog');

$button->signal_connect('clicked', \&clicked);

my $box = Gtk2::HBox->new;
$box->set_spacing(8);

$box->pack_start($label, 0, 0, 0);
$box->pack_start($button, 0, 0, 0);
$box->pack_start($image, 0, 0, 0);

$window->add($box);

$window->show_all;

Gtk2->main;

exit;

sub clicked {

    $window->remove($box);          # remove from the window
    foreach my $child ($box->get_children) {
        $box->remove($child);      # remove all the children
    }
    $box->destroy;                  # destroy

    # create a new box:
    my $new_box = (ref($box) eq 'Gtk2::VBox' ? Gtk2::HBox->new :
        Gtk2::VBox->new);

    # re-pack the widgets:
    $new_box->pack_start($label, 0, 0, 0);
    $new_box->pack_start($button, 0, 0, 0);
    $new_box->pack_start($image, 0, 0, 0);
    $new_box->show_all;

    $box = $new_box;
    $window->add($box);             # add the new box
    return 1;
}
```

Listing 5

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

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

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

my @values = (
    [ 0, 0 ],
    [ 1, 0 ],
    [ 1, 1 ],
);

foreach my $args (@values) {
```

```

my $hbox = Gtk2::HBox->new;
$hbox->set_spacing(8);
$hbox->pack_start(Gtk2::Button->new('First Item'), @{$sargs}, 0);
$hbox->pack_start(Gtk2::Button->new('Second'), @{$sargs}, 0);
$hbox->pack_start(Gtk2::Button->new('Third'), @{$sargs}, 0);
$vbox->pack_start($hbox, 0, 0, 0);
}

$window->add($vbox);

$window->show_all;

Gtk2->main;

exit;

```

Listing 6

```

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

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

my $table = Gtk2::Table->new(4, 2);
$table->set_col_spacings(8);
$table->set_row_spacings(8);

# two buttons side-by-side:
$table->attach_defaults(Gtk2::Button->new('0,0 -> 1, 0'), 0, 1, 0, 1);
$table->attach_defaults(Gtk2::Button->new('1,0 -> 2, 0'), 1, 2, 0, 1);

# one button spanning two columns:
$table->attach_defaults(Gtk2::Button->new('0,1 -> 2, 1'), 0, 2, 1, 2);

# one button spanning two rows:
$table->attach_defaults(Gtk2::Button->new('0,3 -> 1, 4'), 0, 1, 2, 4);

# two more buttons to fill the space:
$table->attach_defaults(Gtk2::Button->new('1,3 -> 2, 3'), 1, 2, 2, 3);
$table->attach_defaults(Gtk2::Button->new('1,4 -> 2, 4'), 1, 2, 3, 4);

$window->add($table);
$window->show_all;

Gtk2->main;

exit;

```

Listing 7

```

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

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

my $frame = Gtk2::Frame->new('Frame Title');

$frame->set_shadow_type('etched_in');

my $button = Gtk2::Button->new('Frame Contents');
$button->signal_connect('clicked', sub { Gtk2->main_quit });
$button->set_border_width(8);

$frame->add($button);

$window->add($frame);

$window->show_all;

Gtk2->main;

```

TPJ

Renew Now & Save!

Plus A Special Offer for Current *TPJ* Subscribers!

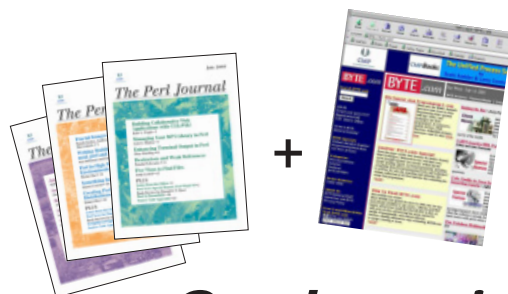
The time for subscription renewal is **NOW**
—and we have a special offer for **all** current subscribers!

- You can lock in next year's subscription (and the year beyond that, too!) at the low rate of \$16/year by renewing **now**!
- Have access to the complete *TPJ* archives—Spring 1996 to the present! (Effective September 1, 2003.)
- And by renewing between now and November 1, 2003, you also get one year of access to **BYTE.com** **at no extra charge**!

That's a savings of nearly \$20—and you get **BYTE** columns by Moshe Bar, Jerry Pournelle, Martin Heller, David Em, Andy Patrizio, and Lincoln Spector, plus features on a wide range of technology topics.

Currently, all new subscriptions to *The Perl Journal* are \$18/year. But to show our appreciation for your support in our first year of publication, we're making this special limited-time offer available to current *TPJ* subscribers who renew between now and November 1.

**Don't miss out on a single issue or
this special offer! Renew now!**



**= One low price!
One great deal!**

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

The Perl Journal

Automated Testing with the Perl *Test::* Modules

Automated testing is a big part of Extreme Programming, but how do you do it in Perl? Actually, you've been using automated testing all along—when you install a new version of Perl or install a module. All the tools are there, in use, but you may have never noticed them before when you ran *make test*.

If you've installed a module from CPAN, and run *make test*, you've seen the testing modules at work, as in this snippet from the installation of *WWW::Mechanize*:

```
$ make test
perl "-MExtUtils::Command::MM" "-e" "test_harness...."
t/00.load.....ok
t/add_header.....ok
... 32 omitted lines ...
t/warn.....ok
All tests successful.
Files=35, Tests=413, 46 wallclock secs
(36.10 cusr + 2.55 csys = 38.65 CPU)
```

After a few moments of flashing counters, the battery of tests is completed and you know that everything has run correctly on your specific machine. You can install the module with confidence.

Automated tests aren't just for module distributions. You can use Perl's tools to automate testing of your code even if it's not going to be uploaded to CPAN or even made into module. Think of each test that runs as both guardian angel and tireless assistant, doing the drudgery of checking your code every time you run the test suite, safeguarding against future breakage. My department has a battery of 11,000 tests that run once an hour, making sure all parts of our web site, from component to web interface, work and haven't been inadvertently broken. Testing can even help you write better code by forcing you to think early in the process about your API.

The Perl Testing Tools

There are two parts to the Perl testing tools: *Test::Harness* parses output created by *Test::Simple* in a simple format that's easy for humans to read. Each test program is a small Perl program with, by convention, a ".t" suffix for a filename. The test program prints out a plan that tells how many tests it plans to run, and then a series of test lines. Each test line starts with "ok" or "not ok," followed by a test number and an optional comment. For exam-

ple, a program, *prime.t*, that tests a function that checks for prime numbers might create the following output:

```
$ perl prime.t
1..4
ok 1 - 2 is prime
ok 2 - 3 is prime
not ok 3 - 4 is not prime
# Failed test (prime.t at line 9)
ok 4 - 7 is prime
# Looks like you failed 1 tests of 4.
```

We can see that the third test failed, and the comment tells us what was being tested, so we can easily find it in the source. However, this output is usually handled by the *Test::Harness::runtests()*, which runs our ".t" files, analyzes the output, and reports a summary.

```
$ perl -MTest::Harness -e'runtests(@ARGV)' prime.t
test....FAILED test 3
      Failed 1/4 tests, 75.00% okay
Failed Test Stat Wstat Total Fail   Failed List of Failed
-----
prime.t              4      1 25.00% 3
Failed 1/1 test scripts, 0.00% okay. 1/4 subtests failed, 75.00% okay.
```

When we fix our tests in *prime.t*, and it runs clean as:

```
$ perl prime.t
1..4
ok 1 - 2 is prime
ok 2 - 3 is prime
ok 3 - 4 is not prime
ok 4 - 7 is prime
```

then *Test::Harness* gives us the thumbs-up and gives timing statistics on what was run:

```
$ perl -MTest::Harness -e'runtests(@ARGV)' prime.t
prime....ok
All tests successful.
Files=1, Tests=4, 0 wallclock secs ( 0.01 cusr + 0.00 csys = 0.01 CPU)
```

Writing Your First .t File

We've seen the test results, but what got tested and how? Our *test.t* is a regular Perl program, but with a ".t" extension. It

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.

exercises the code that we want to test, and uses *Test::Simple*'s *ok()* function to do the test tracking, leaving the programmer free to think about what needs to be tested. Every test comes down to a pass/fail response: Did my code do what I expected it to do? The result of that question is a Boolean, and is passed to *ok()*.

Let's take my *is_prime()* function, which should return True if the value passed is prime. Three is a prime number, so *is_prime(3)* should return True. If it does, our test passes. Here's a simple "prime.t":

```
#!/usr/bin/perl
use Test::Simple tests=>1;
use Primes;      # The module we're testing
ok( is_prime(3) );
```

The *Test::Simple* line sets up the testing framework, based on the number of tests we specified, and makes the *ok()* function available for your use. Then, the *ok()* call means we're saying, "The test passes if *is_prime(3)* returns True." Assuming that my code is working correctly and the *is_prime()* call does return a True value, *ok()* takes the value and outputs

```
ok 1
```

which is what tells *Test::Harness* that the test actually passed. This is assuming that our call actually returns True. Conversely, *is_prime(10)* should return a False value, so we negate the return from it and pass that to *ok()*. We'll add a comment as well.

```
ok( !is_prime(10), "Ten is not prime" );
```

which prints

```
ok 2 - Ten is not prime
```

Test::Harness ignores the comment after "ok 2", but it helps the human to find failing tests. Get in the habit of giving each test a comment. It's not much fun to have to count tests in the course code because you know only that test 57 of 72 failed.

Note that *ok()* is keeping track of our test numbering for us. You may see old tests in the Perl core where the test program keeps track of its own test count, but *Test::Simple* frees you from that drudgery.

Any expression can be passed to *ok()* and will be validated in a Boolean context.

```
ok( $hostname eq "chimpy",      "Valid hostname" );
ok( $parent =~ /^A(m|nd)y$/,    "Correct parent name" );
ok( $statecount == 50,          "Got all states, but not DC" );
```

The Importance of the Plan

Note how at the top of our test programs, we tell *Test::Simple* the number of tests to expect, which we call the plan. This lets *Test::Harness* validate that all tests have actually been run. If our test program exited after only running 10 tests, but we'd told *Test::Simple* to expect 15, then *Test::Harness* would know something was amiss and fail the test. In a way, validating the plan is itself a test, although the harness doesn't include it in the test count.

Sometimes you don't know how many tests you'll be running at compile time, when the *use* is executed. You can calculate the number of tests and then call the *plan* function directly, as long as it's before any *ok()* calls have been made. Say we want to check that all the HTML files in our directory only have lowercase letters with no digits or punctuation. The number of files might change over time, but not the rule.

```
use Test::Simple;

my @files = glob( "**.html" ); plan( tests => scalar @files ); for
my $filename ( @files ) {
    ok( $filename =~ /^[a-z]+\..html$/, "$filename is lowercase" );
}
```

Finally, there may be times when it's just not possible to determine the number of tests before calling *ok()*. In this case, tell *Test::Simple* that you don't have a plan. Pass the string '*no_plan*' in your *use*:

```
use Test::Simple 'no_plan';
```

*Think of each test that
runs as both guardian angel
and tireless assistant*

Using prove

The *prove* utility comes with *Test::Harness* as of version 2.32, and lets you easily run tests on a set of files or one single file. It's basically a wrapper around the *run_tests()* function in *Test::Harness*, but it is designed to make it easy to get into a rhythm of test-code-test-code (more on this in the section entitled "Test-First Programming"). *prove* is run simply as

```
$ prove *.t
digits...ok
prime....ok
All tests successful.
Files=2, Tests=17,  0 wallclock secs ( 0.05 cusr +  0.03 csys =  0.08 CPU)
```

The *-v* option turns on the verbosity, so you can see each sub-test, not just the summary, for a specific test:

```
$ prove -v prime.t
prime....1..4
ok 1 - 2 is prime
ok 2 - 3 is prime
ok 3 - 4 is not prime
ok 4 - 7 is prime
ok
All tests successful.
Files=1, Tests=4,  0 wallclock secs ( 0.06 cusr +  0.01 csys =  0.07 CPU)
```

Any directories that *prove* sees will get expanded to mean all files in that directory, and the *-r* flag recurses into all the directories underneath that.

```
$ prove -r users/
users/admin/new....ok
users/admin/privs..ok
users/delete.....ok
users/insert.....ok
users/new.....ok
All tests successful.
Files=5, Tests=528,  8 wallclock secs ( 2.72 cusr +  1.63 csys =  4.35 CPU)
```


On Beyond ok()

With the knowledge of *ok()* to test your functions and *prove* to run the tests, you have all you need to create effective test suites. In fact, much of the Perl core uses functionality such as *ok()* as part of the tests that get run when you build Perl.

However, most of your tests will be doing the same sorts of things over and over, so you'll want to turn to *Test::Simple*'s big brother, *Test::More*. These are wrapper functions around *ok()* that encapsulate common testing tasks and make debugging easier. *Test::More* is 100 percent compatible with *Test::Simple*, so you can start writing tests with *Test::Simple* and move to *Test::More* when you're comfortable.

Many times when you're testing, you'll want to see if a given value is what you expected. For instance, we may have a function *allcaps()* that should turn a string into all capital letters.

```
ok( allcaps( "Perl 5.8.1" ) eq "PERL 5.8.1" );
```

If the returned value is what we're expecting, the equality matches and the test passes. But what if it doesn't match? All we know is that the test failed, and on what line:

```
not ok 1
#   Failed test (caps.t at line 3)
```

The *is()* function compares two values passed to it, and if they're not equal, shows the expected and actual results, making debugging easy.

```
not ok 1
#   Failed test (caps.t at line 3)
#       got: 'PERL'
#   expected: 'PERL 5.8.1'
```

Now we can see that *allcaps()* seems to have dropped all non-alpha characters, and we know where to get started looking.

is() compares its values as strings, using the *eq* operator. If you want to use some other relational operator, especially if you want the values compared as numbers, you'll need to use the *cmp_ok()* function. The two following examples work the same, but just as with the *is()* function, *cmp_ok()* will show exactly what values caused the problem:

```
ok( $files >= 3, "We have at least three files" );
cmp_ok( $files, ">=", 3, "We have at least three files" );
```

is() has a counterpart, *isnt()*, which checks that one value is anything except for the second value, and *like()* has *unlike()* that works the same way with regexes. *cmp_ok()* doesn't need an opposite since each relational operator has its own opposite.

Throughout your test programs, you may find that the comments in your test calls aren't enough. The *diag()* function lets you embed diagnostics in your test output. Any strings passed to *diag()* are printed with a pound sign prepended, so that *Test::Harness* ignores them.

More Complex Checking

Test::More also has functions to help testing complex data structures. The *isa_ok()* checks that a given variable is defined and blessed into a specific package:

```
my $user = new My::User();
isa_ok( $user, 'My::User' );
```

If you do anything with objects, always remember to use *isa_ok()* throughout your tests—don't just use test constructors. Test any-

thing that can return an object, like a factory method or a file-walking method that returns an object instance.

Hashes and arrays are also supported. Instead of writing code to walk arrays or hashes, use the *is_deeply()* function to make an element-by-element comparison of your structures. If the structures differ, *is_deeply()* reports on the first element that fails:

```
my @expected_stooges = qw( Larry Moe Curly Iggy );
my @test_stooges = get_stooges();
is_deeply( \@test_stooges, \@expected_stooges, 'stooge check' );

not ok 1 - stooge check
#   Failed test (stooges.t at line 9)
#   Structures begin differing at:
#       $got->[3] = 'Shemp'
#   $expected->[3] = 'Iggy'
```

Test-First Programming

You may find yourself saying, "All these tests look great, but how am I going to have enough time to write them?" Easy: Write the tests before you write the code. Test-first programming is a key principle of Extreme Programming, and even if you're not an XP fan, it's hard to dispute the value of test-first. A test defines your expectation of the code with simple and absolute clarity.

Say you're writing a function that will tell whether a number is prime. Before you even think about how you'll write the code, write some simple tests. Write tests for the first handful of integers:

```
use Test::More tests => 11;

ok( is_prime( 2 ), "Two is" );
ok( is_prime( 3 ), "Three is" );
ok( !is_prime( 4 ), "Four is not" );
ok( is_prime( 5 ), "Five is" );
ok( !is_prime( 6 ), "Six is not" );
ok( is_prime( 2711 ), "Some big prime" );
```

Those are good tests for when someone passes in nice, well-behaved input, but what about weird cases? What about a negative number? Or a noninteger? Your function should handle all those as well, so write your test cases for them:

```
diag( "Check the weird cases" );
ok( !is_prime( -1 ), "Negatives are never prime" );
ok( !is_prime( 0 ), "Zero is not prime" );
ok( !is_prime( 1 ), "Neither is one" );
ok( !is_prime( 3.14159 ), "Fractions aren't" );
ok( !is_prime( "five" ), "Strings sure aren't" );
```

It doesn't matter what your code does, so long as you define it. Put it in the POD for the function and explicitly describe the behavior. (Maybe you want *is_prime()* to emit a warning: You can check that with *Test::Warn*, available on CPAN.)

Now that you've written tests for some common cases and a few edge cases, you can write your function. Start with the POD documentation for the function. The documentation explains the special cases that we came up with.

```
is_prime( $n )

returns True if $n is a prime number (is positive and has exactly
two factors: 1 and itself). Nonintegers are never primes.
```

Finally, write the code:

```
sub is_prime {
    my $n = shift;
```

```

return 0 if $n <= 0;           # Negatives aren't prime
return 0 unless $n == int($n); # Non-integers aren't prime

my $nfactors = 0;
for my $i ( 1..$n ) {
    ++$nfactors if ($n % $i == 0);
}

return $nfactors == 2;
}

```

TODO & SKIP Blocks

Sometimes you'll write a set of tests, but not have the code completed, and you'll want to check your code back into CVS. You have the subroutine stub created, but it's time to go home for the day. You don't want to check-in code that fails a test, so you wrap the unfinished tests in a TODO block:

```

TODO: {
    local $TODO = "Haven't written pi digitizer yet";

    is( pi_digit(1), "3" );
    is( pi_digit(2), "1" );
    is( pi_digit(100), "7" );
}

```

The TODO block notifies *Test::More* that although the tests should be run, they should all fail. Any tests that succeed will be flagged as having unexpectedly succeeded. This will show as a warning at the end of your run of *prove* or *make test*, but the test run as a whole will still pass.

The counterpart to the TODO block is the SKIP block. SKIP blocks are for sections of code that might not run on all given machines, or under all given circumstances. You might have some

code that shouldn't get run if a certain module isn't installed or if there's not an Internet connection. In these cases, we don't want to run the tests at all.

Say you have a test that uses the *Test::HTML::Lint* module to validate your HTML output and report on it. Unfortunately, not everyone using your module will have *Test::HTML::Lint*:

```

SKIP: {
    eval "use Test::HTML::Lint";
    skip "Test::HTML::Lint not installed", 2 if $@;

    html_ok( $main_page, "Main page is valid" );
    html_ok( $admin_page, "Admin page is valid" );
}

```

Here, if we're unable to *use* the module, the *skip()* function is called, which then jumps out of the SKIP block. The two *html_ok()* calls are never made. Note that we told *skip()* that there were two tests we were skipping; otherwise, our test numbering would be off and the plan would fail.

Wrapping Up

Automated testing has changed the way I think about programming. I'm confident in the new code I write, and refactoring is a joy, instead of nerve-wracking, because I'm no longer afraid of breaking existing code.

Keep an eye on the *prove* utility that comes with *Test::Harness*. It's a work in progress based on the past two years of testing work in my department. I welcome your comments and suggestions to make Perl's testing tools make your coding life easier.

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/>

Quantifying Popular Programming Languages

For any number of reasons, people always want to know the relative popularity of various programming languages. To provide one measure of language popularity, we focused upon one publicly available and objective measurement—the number of web-based job offers that specify requirements for different programming languages. Our most recent analysis covered the 12-month period from July 2002 to June 2003. We scanned job offers in the category “software” from several employment web sites, eliminating duplicates. Table 1 presents our results.

There are several technical issues: We eliminated duplicate offers on a monthly basis. We used case-insensitive matches. In

“the letter C, preceded by a character that isn’t a letter or digit, and followed by a character that isn’t a letter or a digit or a period or a sharp-sign or a plus”). We then visually scanned a month’s data, finding that about 5 percent of the “C” hits are clearly not C programming jobs: “Bldg C,” “Suite C,” “Unit C,” “A/C” (air conditioning?), “C-Level” (“C-level sales,” “C-level executive”), “4.6.C,” “C-1426,” and so on. Therefore, we prefiltered “C-LANG” and “C-CODE” into plain “C,” we prefiltered “C-SHARP” or “C SHARP” into “C#,” we prefiltered “C ++” or “C PLUS PLUS” into “C++,” then we excluded “period before or after C,” and we excluded “hyphen after C.”

Month	C++	Java	C	C/C++	J*Script	C#	.Net	Cobol	Pascal	Fortran	Ada	RPG	VBasic	VBasic.NET	Perl
2002-07	52.8%	40.9%	33.6%	21.8%	8.0%	2.3%	13.3%	5.4%	0.2%	0.7%	5.6%	2.1%	18.1%	3.2%	12.5%
2002-08	50.3%	44.1%	42.6%	25.9%	10.6%	3.0%	13.2%	5.1%	0.2%	0.8%	4.1%	2.1%	21.3%	4.0%	14.9%
2002-09	50.2%	43.4%	40.6%	26.1%	8.6%	3.2%	15.4%	5.3%	0.2%	1.3%	5.2%	2.3%	20.5%	4.4%	13.6%
2002-10	46.3%	43.2%	35.0%	22.8%	11.5%	4.9%	17.1%	6.9%	0.3%	1.2%	6.1%	2.8%	22.6%	5.0%	11.0%
2002-11	47.2%	44.2%	37.7%	25.1%	10.5%	4.9%	17.5%	5.1%	0.3%	1.4%	4.6%	2.0%	19.0%	4.8%	14.3%
2002-12	47.2%	45.6%	35.5%	23.9%	9.1%	6.0%	18.5%	5.8%	0.4%	1.2%	4.7%	1.7%	21.7%	6.2%	14.1%
2003-01	49.2%	43.6%	36.1%	26.6%	10.0%	6.2%	19.0%	3.7%	0.3%	1.4%	5.7%	1.2%	17.9%	5.9%	11.8%
2003-02	52.2%	41.5%	42.2%	31.3%	9.1%	6.4%	17.2%	3.7%	0.2%	1.5%	6.0%	0.6%	16.8%	5.2%	11.4%
2003-03	50.3%	39.7%	40.5%	28.2%	8.9%	6.9%	17.9%	4.1%	0.1%	0.9%	4.6%	0.8%	18.5%	6.3%	12.6%
2003-04	47.4%	41.0%	39.9%	26.4%	8.4%	6.3%	16.8%	3.9%	0.2%	1.1%	4.9%	0.8%	18.5%	6.3%	14.3%
2003-05	57.2%	39.8%	40.6%	30.3%	9.2%	6.0%	15.5%	2.3%	0.2%	1.0%	8.8%	0.7%	18.4%	6.7%	10.4%
2003-06	55.5%	42.3%	43.0%	31.2%	9.6%	5.8%	15.2%	2.6%	0.2%	1.1%	7.1%	1.2%	17.5%	6.3%	13.3%

Table 1: Jobs per month according to language.

counting “Java” requirements, we had to avoid false hits on “JAVASCRIPT” (obviously). We counted “J2EE,” “J2SE,” and “J2ME” as equivalent to “JAVA.” We added “JAVASCRIPT,” “JSCRIPT,” and “ECMAScript” together to make a “J*script” total. To exclude “VBA” and “VBSCRIPT” from the “Vbasic” total, we matched “VB followed by any letter except A or S,” adding that to the matches for “VISUAL BASIC.” The total indicated as “Vbasic.net” counts job offers that matched “Vbasic” and also “.NET.” We noticed false hits on “PASCAL” when the name “Pascalle” appeared in the job offer, so we counted only “PASCAL followed by a nonletter.” Several web sites could not properly handle “C#” as a lookup keyword, so we scanned the full text of all “software” offers and performed our own keyword search. The number of (nonduplicate) job offers per month varied, but was never less than 4000.

Determining the percentages for “C” was the most challenging. Our first attempts used a simple regular expression like the other matches: “[^A-Za-z0-9]C[^A-Za-z0-9#+]” (which means

We added another category—“C/C++”—which contained all the “C” cases that also contain “C++” somewhere (usually, but not always, the keyword “C/C++”). The “C/C++” percentage was usually more than half of the “C” percentage, and about half of the “C++” percentage.

From early reviewers, we received some comments on our methodology. One reader cautioned that percentages based on published job offers overlook those offers filled internally within the organization and that the percentages of jobs filled internally might be significantly different. We agree, but can’t study internally filled jobs with our methodology. Other readers requested more languages. We’ve added all the languages requested so far; if you want more languages or more detailed analysis, just ask us.

If you believe that some publicly available job sites are biased in favor of, or against, any particular programming language, we would be grateful for your information. (As of today, we are unaware of any such biases.)

My special thanks to Doug Teeple (teeple@wi-fone.com) and John Breeden (jbreenen@plumhall.com), for valuable assistance with the survey software.

Thomas has authored four books on C, and coauthored Efficient C (with Jim Brodie) and C++ Programming Guidelines (with Daniel Saks). Plum Hall (his company) provides test suites for C, C++, Java, and C#. Thomas can be contacted at tplum@plumhall.com.

TPJ



Perl Poker

Simon Cozens

It's not all work and no play at my company. In fact, the Northern Ireland Perl Mongers scene has been swept with somewhat of a poker craze recently. As well as trying to improve our own poker play, we've naturally had a look around at the state of computer poker research.

We found a great number of interesting things: online poker servers where humans and poker-playing robots could play against each other, papers on "pseudo-optimum strategy," evaluation algorithms, and examples in C and in Java—but nothing in Perl.

This needed to be fixed.

Getting Straight to Work

Thankfully, quite a few of the tools out there can be integrated pretty quickly into Perl without much effort. For starters, we can take *Games::Cards* as our card library. We'll make some revisions to it later so that it can handle more of the areas we need.

Another thing we'll need to do is have some means of evaluating what's a winning hand. The good news is, there's a GNU poker library (no, really), which does that for us, at <http://pokersource.sourceforge.net/>; the bad news is, it's in C.

Well, not a problem. It's time to dive into some XS as a simple way of linking these things together. We'll start by creating the framework for our module with *h2xs*:

```
% h2xs -A -n Games::Poker::HandEvaluator
```

and now we need to make sure that *libpoker* gets linked in, by telling the Makefile.PL about the library:

```
LIBS => '-lpoker'
```

Assuming that the poker library is correctly installed, this will be enough to link the library into our Perl module. We run Makefile.PL and do a *make test* to ensure everything's OK. If the library isn't correctly installed, you might see something like this:

```
Note (probably harmless): No library found for -lpoker
```

The "probably harmless" is a complete lie.

Now we have to write a little bit of glue code to expose the parts of the library's functionality that we're interested in. The hand-evaluation function for an ordinary deck of cards in stan-

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.

dard (nonlowball) poker is called *StdDeck_StdRules_EVAL_N*. However, it expects to see a bitmask of the deck, and we don't really want to construct that bitmask ourselves. Instead, we borrow a function from the examples that ship with *libpoker* for turning a string into a bitmask. This function goes above the *XS MODULE* line in *HandEvaluator.xs*, as we're not going to need to call it from Perl.

```
int parse_cards(char *handstr, StdDeck_CardMask* cards) {
    char *p;
    int c = 0;
    int ncards = 0;
    char str[80];

    StdDeck_CardMask_RESET(*cards);
    strcpy(str, handstr);
    p = strtok(str, " ");

    do {
        if (DstringToCard(StdDeck, p, &c) == 0)
            return 0;
        if (!StdDeck_CardMask_CARD_IS_SET(*cards, c)) {
            StdDeck_CardMask_SET(*cards, c);
            ++ncards;
        }
    } while ((p = strtok(NULL, " ")) != NULL);
    return ncards;
}
```

Now we can wrap our evaluation function. This will take a string representing a hand of cards—for instance, *Ah 7c Jd As 7h* for two pair aces and sevens—and return an integer. The higher the integer, the better the hand:

```
int
_evaluate( hand );
char* hand;

PREINIT:
    StdDeck_CardMask cards;
    int ncards;

CODE:
    ncards = parse_cards(hand, &cards);
    if (ncards)
        RETVAL = StdDeck_StdRules_EVAL_N(cards, ncards);
    else
        RETVAL = 0;

OUTPUT:
    RETVAL
```


Of course, we often want an explanation of that integer (such as “Two Pair (A 7 J)”). The *libpoker* function *StdRules_HandVal_toString* fills a buffer with such an explanation, so we’ll wrap that in an XS function called *handval*:

```
char*
handval( hval )
{
    int hval;

PREINIT:
    char buf[80];
    int n;

CODE:
    StdRules_HandVal_toString(hval, buf);

    RETVAL = buf;

OUTPUT:
    RETVAL
}
```

We’ll now write a wrapper function that can handle both an ordinary string and a *Games::Card::CardSet* object, so the module will not need changes when we integrate it with the rest of our poker modules. *Games::Card::CardSet* (and hence its derived classes) has a *print* method that nearly does what we want, but not quite. It prints out the hand like so:

```
Player 1: 2S 2C 3S 4D 6D 9C 10S
```

whereas we want

```
2S 2C 3S 4D 6D 9C TS
```

(Notice that *libpoker* expects “T” for ten)

This is nothing that a few regular expressions can’t fix:

```
sub evaluate {
    my $hand = shift;
    if (UNIVERSAL::isa($hand, "Games::Cards::CardSet")) {
        $hand = $hand->print;
        $hand =~ s/\.*/;/; $hand =~ s/\s+/ /g;
        $hand =~ s/10/T/g;
    }
    return 0 unless $hand;
    _evaluate($hand);
}
```

The mistake everyone makes at least once is to say *\$hand->isa*, which fails badly if *\$hand* is just an ordinary string—*UNIVERSAL::isa* is a good way of getting around the problem.

For My Next Trick...

And that’s basically all we need to do for our hand-evaluation library. Now we’ll turn our attention to the Online Poker Protocol, a mechanism designed by the folks at Alberta University, who have been doing a lot of work on computer poker.

Their protocol and servers allow computer players to pit their wits against humans. They play Texas Hold’em, a variant of poker where each player receives two cards of their own (“hold cards”), and then five community cards are dealt face down before all players (“the board”). The board is revealed in three stages, with a round of betting before each stage. The first stage is the “flop,” where three of the board cards are revealed. Then comes a round of betting and the “turn,” where another board card is turned over. After another round of betting, the final card is revealed on the “river.” Then there is a final round of betting before all players’ cards are turned over in the “showdown.”

We’ll begin by writing a simple client to allow us to play on the server, with a view to developing a computer player in the future.

The protocol is documented at the section on <http://games.cs.ualberta.ca/webgames/poker/bots.html> in the *http*: manpage, and is a binary TCP/IP communication protocol. Debugging binary protocols is not terribly easy and I needed some more information about how the protocol dealt with certain conditions, so I wrote a rather nice protocol analyzer, which I’m sure I’ll tell you about next time.

We start by defining the protocol in terms Perl can understand. For instance, it makes sense to have the command names linked to their byte equivalent as constants, like so:

```
use constant JOIN_GAME => 20;
```

And we also maintain an array that specifies how the arguments are to be formatted. The protocol uses 4-byte integers and zero-terminated strings, so it seems reasonable to use the *pack* function to prepare data for transit. For instance, the *JOIN_GAME* command takes two strings (username and password) followed by an integer (protocol version) and another string (client identifier). The *pack* encoding of this would be *Z*Z*NZ**:

```
$protocol[JOIN_GAME] = "Z*Z*NZ*";
```

Now we can write a function that can be called with the right arguments:

```
$self->send_packet(JOIN_GAME,
    "perlribot",
    "sekrit",
    1,
    "Games::Poker::OPP");
```

which packs this into a packet as specified by the protocol and sends it down to the server. We start by making sure that the message number is a valid part of the protocol:

```
sub send_packet {
    my ($self, $message_id, @data) = @_;
    croak sprintf "Protocol error: command 0x%x not recognised",
        $message_id unless exists $protocol[$message_id];
}
```

and now we can simply use *pack* to transform the arguments:

```
if ($protocol[$message_id]) {
    eval { $packed_data = pack($protocol[$message_id], @data); };
    croak sprintf "Problem packing data for %d command",
        $message_id if $@;
}
```

Now we have the arguments packed into *\$packed_data*. The protocol specifies that we first send the message number as a 4-byte integer, followed by the length of the arguments (that’s *\$packed_data*) as a 4-byte integer:

```
my $packet = pack "NN", $message_id, length $packed_data;
```

Finally, we add on the arguments and send the packet out:

```
$packet .= $packed_data;
$self->put($packet);
return $packet;
```

Notice that we use another method, *put*, to actually put the data “on the wire.” This gives us abstraction about how we do this—we might want to use *IO::Socket::INET*, or a *POE* wheel, or something else entirely. Similarly, we can write a helper function to

retrieve a packet from the server, unpack the arguments in the same way, and return a message number and unpacked arguments.

After creating a standard constructor to hold the username, password, and server connection details, we can write an *IO::Socket::INET* implementation of the module like so:

```
sub connect {
    my $self = shift;
    $self->{socket} = IO::Socket::INET->new(
        PeerHost => $self->{server},
        PeerPort => $self->{port},
    );
}

sub put { my ($self, $what) = @_; $self->{socket}->syswrite($what); }
sub get {
    my ($self, $len) = @_;
    my $buf = " " x $len;
    my $newlen = $self->{socket}->sysread($buf, $len);
    return substr($buf, 0, $newlen);
}
```

Now all the groundwork is finally done! Writing functions to actually speak the protocol is now a lot more straightforward:

```
sub joingame {
    my $self = shift;
    $self->send_packet(JOIN_GAME, $self->{username}, $self->{password},
        1, "Games::Poker::OPP");
    my ($status) = $self->get_packet();
    if ($status == GOODPASS) { return 1; }
    elsif ($status == BADPASS) { return 0; }
    else {
        croak sprintf "Protocol error: got %i from server", $status;
    }
}
```

Now we need to think about how to handle the game play itself. There are many ways to do this, but the way I decided upon was to provide a main loop function *playgame*, which calls user-defined callbacks to display status information and decide how to bet. To help the user writing these callbacks, we export some of the protocol constants, particularly those that represent actions that the player can make (e.g., *FOLD*, *CHECK*, *RAISE*, *CALL*).

I also decided to create an object to hold the game state. Since this is generic to Texas Hold'em and not specific to our protocol, I created a separate module called *Games::Poker::TexasHold'em*. (Yes, this is a gratuitous abuse of apostrophe-as-package-separator.) In the future, this module will be extended with functions to analyze hand potential, but for now, it just keeps track of the game.

I won't go into much detail about the *TexasHold'em* module, except to say that it can be used quite independently of the On-line Poker Protocol module:

```
use Games::Poker::TexasHold'em;
my $game = Games::Poker::TexasHold'em->new(
    players => [
        { name => "lathos", bankroll => 500 },
        { name => "MarcBeth", bankroll => 500 },
        { name => "Hectate", bankroll => 500 },
        { name => "RichardIII", bankroll => 500 },
    ],
    button => "Hectate",
    bet => 10,
    limit => 50
);
```

```
$game->blinds;
$game->check; $game->bet(10); $game->call; $game->fold;
$game->fold;
$game->next_stage;
$game->check; $game->bet(20); $game->raise(40); $game->call;
$game->next_stage;
print $game->status;
```

This will print out:

```
Pot: 80 Stage: turn
?           Name Bankroll InPot
.           lathos $   470 $   30
F           MarcBeth $   490 $   10
F           Hectate $   490 $   10
           RichardIII $   470 $   30
```

Showing that all but *lathos* and *RichardIII* have folded, these two started madly throwing cash into the pot. (Which is, for those in the know, what usually happens in games...) We're currently in the turn, it's *lathos*' bet, and there's \$80 in the pot.

playgame sits in a loop, looking for messages from the server and determining how to respond to them:

```
sub playgame {
    my $self = shift;
    $self->{game} = undef;

    while (my ($cmd, @data) = $self->get_packet()) {
```

A *PING* packet (which doesn't often get sent) needs to be replied immediately with a *PONG*, and that's all we need to do with it:

```
if ($cmd == PING) { $self->send_packet(PONG); next; }
```

And a *GOODBYE* packet should end the loop:

```
if ($cmd == GOODBYE) { last }
```

Anything that is purely advisory, such as chatter from other players or information from the server, gets handed to a user-defined status routine:

```
if ($cmd == CHATTER ||
    $cmd == INFORMATION) {
    $self->{status}->($self, $cmd, @data); next;
}
```

playgame has a concept of the current game. If we join the room in the middle of an existing game, we may get sent messages that are not actually for our consumption. If we don't have anything in *\$self->{game}*, then we ignore the message, unless, of course, it's the start of a new game:

```
next unless $self->{game} or $cmd == START_NEW_GAME;
```

Now we dispatch any command off to its appropriate handler, and also call the *status* routine so that the client gets the opportunity to display some message to the user:

```
if (exists $handlers[$cmd]) {
    $handlers[$cmd]->($self, $cmd, @data);
}
$self->{status}->($self, $cmd, @data);
```

These handlers are responsible for updating the *Games::Poker::TexasHold'em* object, and also for making the all-important callback to determine how the client should play!

But where do these handlers come from? Well, when we fill the *@protocol* array with the *pack* formats of the expected arguments, we also fill the *@handlers* array:

```
map {
    $protocol[$->[0]] = $->[1];
    $handlers[$->[0]] = $->[2] if $->[2];
} (
    [ START_NEW_GAME , "N5(Z*NN)**", \&new_game_handler ],
    [ HOLE_CARDS , "NZ*", \&hole_card_handler ],
    [ NEW_STAGE , "NZ*", \&next_stage_handler ],
    [ NEXT_TO_ACT , "N4", \&next_turn_handler ],
    [ FOLD , "NN", \&fold_handler ],
    [ CALL , "NN", \&call_handler ],
    ...
)
```

This ensures that, for instance, when a player folds, the *fold_handler* routine is called:

```
sub fold_handler { shift->{game}->fold() }
```

And when it's someone's turn to play, the *next_turn_handler* gets called:

```
sub next_turn_handler {
    my ($self, $cmd, $who, $to_call, $min_bet, $max_bet) = @_;
    my $game = $self->{game};
```

It might be our turn to play—*\$who* is set to a seat number, and if that seat number matches ours, then we need to call the callback and do what it tells us to:

```
# If it's me, make the callback
if ($who == $game->{seats}->{$self->{username}}) {
    my $action = $self->{callback}->($self, $to_call, $min_bet, $max_bet);
    return $self->send_packet(ACTION, $action);
}
```

And there is a slight discrepancy between how *Games::Poker::TexasHold'em* works and how the game is played on the servers—after a round of betting, the servers want to start the next round by making the player who was to the left of the dealer in the last round into the dealer for this round, whereas my module handles it by continuing betting from the player after the one who last bet. Of course, the servers are correct, and you might call this discrepancy a bug that I haven't fixed yet, but we use this opportunity to ensure that the status object and the server both agree on who's to play next:

```
$game->{next} = $who;
```

And that's basically all there is to the game play. Let's now turn to building a client with this module.

Upping the Ante

There's a simple, text-based poker client that ships with the module. The bulk of it is made up of the two callbacks that are sent to *Games::Poker::OPP*, the status callback and the main action callback.

This is a very simple callback—it displays the status table, shows your hole cards and the cards on the board, and prompts for what you want to do:

```
callback => sub {
    my $game = shift->state();
    print $game->status;
    print "Hole cards: ", $game->hole, "\n";
    print "Board cards: ", $game->board, "\n";
```

```
print "[F]old, [C]all/check [B]et/[R]aise\n";
print "Your turn: ";
```

And then it reads your reaction, and sends it back:

```
my $action = <STDIN>;
if ($action =~ /\f/i) { $action = FOLD; }
elsif ($action =~ /\b/i) { $action = RAISE; }
else { $action = CALL; }
return $action;
}
```

(The *FOLD*, *RAISE*, and *CALL* constants are exported by the *Games::Poker::OPP* module.)

The status callback is equally simple. It gets a command:

```
status => sub {
    my ($self, $cmd, @stuff) = @_;
```

If that's informational, it just prints out the arguments:

```
if ($cmd == CHATTER || $cmd == INFORMATION) {
    print @stuff, "\n";
    return;
}
```

The server information messages tell you a good deal of what's happening in the game (such as *lathos has folded*), so we don't need to display any specific status information for these commands:

```
return if $cmd == FOLD || $cmd == RAISE || $cmd == CALL
    || $cmd == BLIND;
```

If we were writing a graphical client, though, these would be good opportunities to update the display.

And in all other circumstances, we just print out the game state:

```
my $game = $self->state;
return unless $game;
print "\n--\n";
print $game->status;
print "Hole cards: ", $game->hole, "\n";
print "Board cards: ", $game->board, "\n";
print "--\n";
}
```

That's all it takes to create a poker-playing robot in Perl with *Games::Poker::OPP*—now all that's left to do is add some analysis, a little artificial intelligence, test it out on the servers for a few weeks, and then...Las Vegas, here we come.

TPJ





Free as in Music

Randal Schwartz

I've heard a lot of noise being generated lately about the Record Industry Association of America (better known as the RIAA) cracking down on Internet music-file sharing. They claim that such activity takes the profits away from their artists and the record labels who produce those artists and distribute the music. Critics suggest that most of the money never gets to the artists anyway, and that the artist would often do better by simply publishing directly on the Internet.

One of the articles I read on the subject suggested that we ignore RIAA-member labels and artists completely, and listen only to songs freely available on the Internet, putting our ears where our mouth is, so to speak. But what songs, and where? And how do I keep from downloading so much junk, and find the gems within?

Enter the collaborative filtering model and the power of the masses. In that same article, I found a pointer to iRate, an "internet radio" that learns my preferences, feeding me more and more of what I like. iRate works by downloading a few MP3s to my disk, playing them using a simple embedded player, and then letting me vote on the songs. The songs are selected from publicly available free song sites like IUMA (<http://iuma.com/>) and Artist Direct (<http://artistdirect.com/>), so I can feel safe and legal in downloading and listening to them. The player then uploads my votes, and using a collaborative filtering system, sends me more of what I like. As I vote on more songs, the selections more accurately reflect my taste and provide a variety that far exceeds any commercial station I've frequented.

Try it yourself: Visit <http://irate.sourceforge.net/> and start your personalized Internet radio station today! I've found that setting it on "continuous download" lets me grab as many songs as I can while I'm attached to a fast Internet connection, and then rate songs even when I'm offline later. Be sure to rate quite a few songs before letting it get ahead of you, though, or you'll find the mix to be somewhat unpleasant.

So where does Perl fit in to this? Well, after downloading and playing a few dozen songs using the built-in miniplayer, I longed for the flexibility of my Mac's iTunes player to play the songs, and the ability to listen to these new free songs on my iPod when I was away from my computer.

But I didn't just want to take the entire download directory out of iRate and import it to iTunes because I had already given the

lowest vote to some of those songs ("This sux") and didn't want to waste my disk space on them, and I wanted to preserve my vote rating on the rest of the songs. Luckily, I noticed that the iRate player maintains an XML file, and after a bit of examination, I saw that it contained everything I needed to know to move the files into iTunes, including filename and rating, organized something like this:

```
<Track
  artist="MOTION PICTURE"
  url="http://www.insound.com/media/motion_picture_a_paper_gift.mp3"
  file="/Users/merlyn/irate/download/motion_picture_a_paper_gift.mp3"
  rating="5.0"
  last="10/9/03 5:11 PM"
  played="10"
  title="A Paper Gift"/>
```

From this record, I could simply tell iTunes to add the songfile and give it a particular rating, using *Mac::Glue* and a little help from Chris Nandor to work out the messiness. And parsing the XML was easy using *libxml2* through the *XML::LibXML* interface. The result is found in Listing 1.

Lines 1 through 3 begin nearly every program I write, turning on warnings, enabling the usual compiler and runtime restrictions for large programs, and disabling the output buffering.

Lines 5 and 6 pull in the two modules found in CPAN. *Mac::Glue* is fun to install because it makes my laptop go through a lot of steps as the AppleScript interface gets exercised. Unfortunately, *XML::LibXML* is a bit finicky, but seemed to work fine with the *finx*-installed *libxml2* on my machine.

Line 9 creates a *Mac::Glue* handle for iTunes. Method calls against this object will send messages to iTunes, and the responses get mapped back into values and objects. We'll be using this handle to add the songs as we find them.

Lines 11 and 12 change to the "irate" subdirectory below my home directory. Using an empty *chdir*, I first end up in my home directory without having to explicitly look up the home. The next *chdir* is then relative to the home directory.

Line 14 creates an XML parser using the *XML::LibXML* library. Using the default settings for how things get parsed, I then create the document object in line 15, parsing the "track-database.xml" file, which contains many elements similar to the one presented earlier.

From watching the XML file, I was able to determine that a track goes through a few different stages. First, when a song gets

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.

suggested by the central server, the song record is added with no rating or local filename attributes: just a remote URL. Next, when a song has been successfully downloaded, the *file* attribute is updated to reflect the location on local disk. Finally, when I've listened to the song (or enough of the song to rate it) and provide a rating, the *rating* attribute also gets added. In the current release, the rating seemed to be always one of "0.0," "2.0," "5.0," "7.0" or "10.0." Additionally, if a song couldn't be downloaded, it would have a rating attribute (of 0.0), but no file attribute or a file attribute of an empty string.

All of the songs I want move into iTunes are therefore songs that have a rating and a nonempty filename. I can select those using an XPath expression, as shown in line 17. The resulting list consists of all the nodes that have been rated and are ready to move. Each node ends up one-by-one in *\$track*.

The next step is to pick out the file to see if we've already moved this one. Line 18 looks for the value of the *@file* attribute of the given node, using an XPath-ish way of describing the attribute. A DOM-ish way of getting at the same value might have looked like:

```
my $file = $track->getAttribute("file");
```

You have the option of using whatever you feel is clearer, which is one of the nice things about *XML::LibXML*.

If this string names a nonempty file, then we have a candidate for adding to iTunes; this is tested in line 19. Why would the file be empty? Well, I discovered that if I simply remove the music file once I've moved it, iRate gets mad and redownloads the file to replace it. (If only my real CD collection worked that way...) But if I replace the music file with an empty file, iRate "plays" the file very quickly, and moves on to the next one, causing only a slight pause in the scan for new music. (Perhaps iRate's behavior will change eventually. That'd be nice.)

Line 20 pulls up the iRate rating for this track. Lines 22 to 27 map this rating into the "star" rating for iTunes. A 5-star song in iTunes is an iTunes rating of 100, while no stars (or unrated) is a 0, and everything else is evenly mapped in between. I decided to map the four nonsucking levels of iRate to iTunes levels of five stars through two stars.

If the file has an iRate rating of 0.0, I don't even want to waste the disk space, so line 29 checks this value, and does nothing to the song unless it has a nonzero rating. Otherwise, it's time to move the file, which we ask iTunes to do in line 31. If iTunes isn't

running, this starts it up and sends it an AppleScript to add the given file to its library. (To make this work, I have my preference set to always copy played music files into the iTunes folder.) In the iTunes status window, I see "Copying..." with my music file.

The returned object of *\$s* represents a song in the library. In line 32, we further ask iTunes to set the rating for that song to our new value. This preserves the iRate rating all the way through to the iTunes rating.

Finally, lines 38 and 39 null out the file by opening it for writing and closing it. We're left with an empty file.

Running this program, I get a series of lines like:

```
adding with 60 for /Users/merlyn/irate/download/31_Capricorn_-_She_Says.mp3
adding with 80 for /Users/merlyn/irate/download/Welcome To Tuesday-None-Love
    Crime.mp3
adding with 100 for /Users/merlyn/irate/download/B.C._Powers_-_Baby_I_Do.mp3
adding with 40 for /Users/merlyn/irate/download/Danny Baker Band-Mama%27s
    Cookin%27-Heaven In Your Eyes.mp3
tossing /Users/merlyn/irate/download/brusta-Fresh Interpretation-Fresh
    Interpretation.mp3
```

And whenever it says "adding...", I see iTunes copying the file into the archive. Because these files are now empty, the test in line 19 now skips them, so it's safe to keep rerunning this program as often or as infrequently as I want. The only place I've found a problem is when iRate is playing the file that I'm also moving into iTunes, causing iRate to get a bit upset. The workaround is to have iRate be closed, or to be playing an "unrated" song.

Another problem I noticed is that iTunes grabs the internal MP3 tags from the file, falling back to the filename for artist and title, but I still sometimes get songs called "track 01" in my playlist. Although I have the artist and title information from iRate, I'm not (yet) using it. Maybe in a future edition of this program I'll add those as well. I'd also like to put the URL into a comment for later reference, and maybe even put all such added music into a separate playlist. But this is good enough for now.

So, the next time someone asks "Where is all this free music I keep hearing about on the Internet?", you can now point at your legitimate personally tailored collection and enjoy!

TPJ

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

Listing 1

```
=1=    #!/usr/bin/perl -w
=2=    use strict;
=3=    $|++;
=4=
=5=    use Mac::Glue;
=6=    use XML::LibXML;
=7=
=8=
=9=    my $itunehandle = Mac::Glue->new("iTunes");
=10=
=11=    chdir or die $!;
=12=    chdir "irate" or die $!;
=13=
=14=    my $x = XML::LibXML->new or die;
=15=    my $d = $x->parse_file("trackdatabase.xml") or die;
=16=
=17=    for my $track ($d->findnodes(q{//Track[@rating and @file != ""]})) {
=18=        my $file = $track->findvalue('@file');
=19=        next unless -s $file;
=20=        my $rating = $track->findvalue('@rating');
=21=
=22=        my $irating =
=23=            ($rating >= 10) ? 100 :
=24=            ($rating >= 7) ? 80 :
=25=            ($rating >= 5) ? 60 :
=26=            ($rating >= 2) ? 40 :
```

```
=27=        0;
=28=
=29=        if ($irating) {
=30=            print "adding with $irating for $file\n";
=31=            my $s = $itunehandle->add($file);
=32=            $s->prop("rating")->set(to => $irating);
=33=        } else {
=34=            print "tossing $file\n";
=35=        }
=36=
=37=        ## next part is to fool it from downloading again
=38=        open F, ">$file" or warn "Cannot create $file: $!";
=39=        close F;
=40=    }
```

TPJ



Linux and the UNIX Philosophy

Jack J. Woehr

In 1995, Digital Press published Mike Gancarz's *The UNIX Philosophy*. Gancarz has followed up in the new millennium with a greatly revised, essentially new work: *Linux and the UNIX Philosophy*. As Gancarz explains in the preface:

...[I]t finally dawned on me. [My book] as a description of the UNIX way of thinking was a 'first system.' It was now being drawn into becoming a 'second system,' a...fuller, more developed, more relevant edition of the first.

Sounds deep. Actually, it is.

Linux and the UNIX Philosophy is a book that conceivably could have an impact on the young computer scientist of today comparable to the impact that a book like *Thinking Forth* (Prentice Hall, 1984, ISBN 0-13-917568-7) had on us then-young computer scientists back in '80s. Gancarz explores the philosophical basis of UNIX and ends up expounding the computer science equivalent of what Huxley, Watts, and Isherwood called in the humanities the "perennial philosophy." Simplicity is a virtue. Directness is a virtue. If you keep it clean enough, even users can be taught to use it. We've all spent hours, days, and man-months struggling with managers and team members who couldn't grasp those simple principles!

On another level, this book is painful to read: The pain is nostalgia. "Yes, yes," I want to shout along with the text, "Tell them how it happened so they don't forget. We were there!" Actually, that's pretty much the gist of the foreword by Jon "maddog" Hall. I found the cascade of memories triggered by Gancarz's insightful recital overwhelming—memories of decisions made, of paths taken, of paths not taken, ideas good and ideas not so good succeeding or failing according to merit or whims of fate, as our craft, our industry, our art and world usage of the [Li]U[n]ix operating system(s) blossomed and grew to gargantuan proportions.

Yet, it's more than just a nostalgic trip. *Linux and the UNIX Philosophy* is a useful book because it recounts, with charming brevity, how an immense corpus of business-mission-critical soft-

Jack J. Woehr is an independent consultant and team mentor practicing in Colorado. His website is <http://www.softwoehr.com>.

Linux and the UNIX Philosophy

Mike Gancarz

Digital Press, 2003

200 pp., \$39.99

ISBN 1-55558-273-7

Automating UNIX and Linux Administration

Kirk Bauer

Apress, 2003

592 pp., \$49.99

ISBN 1-59059-212-3

ware emerged in conjunction with the then-industry-novel business model of community-shared intellectual property as represented by GNU/Free Software/Berkeley/Open Source. The inference is that there is some sort of platonic ideal that unites the evolution of the design philosophy of UNIX itself with the advent of the social philosophy of open software.

Not that Gancarz idealizes the operating system itself, as this excerpt from section 7.9 ("Worse is Better") shows:

[T]here is the right way, the wrong way, and the military way ... The 'UNIX way' is akin to the military way. If you listen to the purists, UNIX should have withered and died 20 years ago. Yet, here it is in its entire parasitic splendor, feeding off the criticisms leveled at it...and growing stronger every day.

This book is not just a windy newsgroup rant: *Linux and the UNIX Philosophy* is a first-class folk-history of UNIX and Linux. Many of us might tell the story differently, or may have already told the story differently, drawing at the same time somewhat



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's.



ORDER TODAY!

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

U.S. and Canada

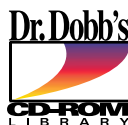
785-841-1631 International

Fax: 785-841-2624

Dr. Dobbs's CD-ROM Library

1601 West 23rd Street, Suite 200

Lawrence, KS 66046-2703



different conclusions based on our individual roles. But if you're willing to listen to a lover of software coding and design tell the story concisely and engagingly, you couldn't ask for better than this book.

You can view the typical publisher info about *Linux and the UNIX Philosophy* at the Digital Press website, <http://www.digitalpressbooks.com/>. I recommend a search by author for Gancarz.

*I want to shout along with the
text, "Tell them how it happened
so they don't forget!"*

Automating UNIX and Linux Administration

Descending from the ideal to the workaday, Kirk Bauer has delivered a worthwhile volume in *Automating UNIX and Linux Administration*. In a literary style resembling a rich series of workshops/lectures, Bauer takes the reader through admin automation starting at the right place—secure shells, of course, to zip and zap your commands around the network—and ending in the realm of automated backups and restores and the user interfaces for them. Sounds very basic, but maybe you've noticed that:

1. When you suddenly decide you need to automate admin tasks, it can take weeks if not months to get right.
2. If it's so basic, why do all the packaged toolchains for these purposes give such magnificent headaches?

Uh-huh. Now you're getting it. It's not a question of whether you are interested in what Mr. Bauer is selling. It's a question of whether you need it explained to you how this stuff is done. In my experience, the answer is, "Yes, the majority of Linux and UNIX sysadmins in the industry need this refresher."

Now, this sort of exposition can be done well or done ill. Bauer does it well. His only visible ideological commitment is to the simple, practical, and inexpensive. So in the end, perhaps this book could be said to manifest the same platonic ideal as *Linux and the UNIX Philosophy* where the rubber meets the road. The book exhibits good writing, good production values, several quick starts (e.g., cvs) for the reader, and a good deal of mostly reusable code.

You can view the typical publisher info (including table of contents, source code, and sample chapter) at <http://www.apress.com/book/bookDisplay.html?bID=211>.

TPJ

Source Code Appendix

Gavin Brown “Programming Graphical Applications with Gtk2-Perl, Part 1”

Listing 1

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

my $quit = sub { exit };

my $window = Gtk2::Window->new;
$window->signal_connect('delete_event', $quit);
$window->set_position('center');
$window->set_border_width(8);
$window->set_title('Hello World!');
$window->set_default_size(200, 100);

my $label = Gtk2::Label->new('Hello World!');

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

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

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

$window->add($vbox);

$window->show_all;

Gtk2->main;
```

Listing 2

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

my $dialog = Gtk2::Dialog->new;

$dialog->set_title('Example Dialog');
$dialog->set_border_width(8);
$dialog->vbox->set_spacing(8);
$dialog->set_position('center');
$dialog->set_default_size(200, 100);

$dialog->add_buttons(
    'gtk-cancel' => 'cancel',
    'gtk-ok' => 'ok'
);

$dialog->signal_connect('response', \&response);

my $label = Gtk2::Label->new('Example dialog');

$dialog->vbox->pack_start($label, 1, 1, 0);

$dialog->show_all;

$dialog->run;

sub response {
    my ($button, $response) = @_;
    print "response code is $response\n";
    exit;
}
```

Listing 3

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

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

my $label = Gtk2::Label->new;

my $markup = <<"END";
This is some plain text.
```



```

<b>This is some bold text.</b>

<span size="small">This is some small text.</span>

<span size="large">This is some large text.</span>

<span foreground="blue">This is blue.</span> <span style="italic">This is italic.</span> <tt>This is monospace.</tt>
END

$label->set_markup($markup);

$window->add($label);

$window->show_all;

Gtk2->main;

exit;

```

Listing 4

```

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

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

my $label = Gtk2::Label->new("Here's a label");
my $button = Gtk2::Button->new("Here's a button");
my $image = Gtk2::Image->new_from_stock('gtk-dialog-info', 'dialog');

$button->signal_connect('clicked', \&clicked);

my $box = Gtk2::HBox->new;
$box->set_spacing(8);

$box->pack_start($label, 0, 0, 0);
$box->pack_start($button, 0, 0, 0);
$box->pack_start($image, 0, 0, 0);

$window->add($box);

$window->show_all;

Gtk2->main;

exit;

sub clicked {

    $window->remove($box);          # remove from the window
    foreach my $child ($box->get_children) {
        $box->remove($child);      # remove all the children
    }
    $box->destroy;                  # destroy

    # create a new box:
    my $new_box = (ref($box) eq 'Gtk2::VBox' ? Gtk2::HBox->new :
        Gtk2::VBox->new);

    # re-pack the widgets:
    $new_box->pack_start($label, 0, 0, 0);
    $new_box->pack_start($button, 0, 0, 0);
    $new_box->pack_start($image, 0, 0, 0);
    $new_box->show_all;

    $box = $new_box;
    $window->add($box);              # add the new box
    return 1;
}

```

Listing 5

```

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

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

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

my @values = (

```

```

        [ 0, 0 ],
        [ 1, 0 ],
        [ 1, 1 ],
    );

    foreach my $args (@values) {
        my $hbox = Gtk2::HBox->new;
        $hbox->set_spacing(8);
        $hbox->pack_start(Gtk2::Button->new('First Item'), @{$args}, 0);
        $hbox->pack_start(Gtk2::Button->new('Second'),    @{$args}, 0);
        $hbox->pack_start(Gtk2::Button->new('Third'),      @{$args}, 0);
        $vbox->pack_start($hbox, 0, 0, 0);
    }

    $window->add($vbox);

    $window->show_all;

    Gtk2->main;

    exit;

```

Listing 6

```

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

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

my $table = Gtk2::Table->new(4, 2);
$table->set_col_spacings(8);
$table->set_row_spacings(8);

# two buttons side-by-side:
$table->attach_defaults(Gtk2::Button->new('0,0 -> 1, 0'), 0, 1, 0, 1);
$table->attach_defaults(Gtk2::Button->new('1,0 -> 2, 0'), 1, 2, 0, 1);

# one button spanning two columns:
$table->attach_defaults(Gtk2::Button->new('0,1 -> 2, 1'), 0, 2, 1, 2);

# one button spanning two rows:
$table->attach_defaults(Gtk2::Button->new('0,3 -> 1, 4'), 0, 1, 2, 4);

# two more buttons to fill the space:
$table->attach_defaults(Gtk2::Button->new('1,3 -> 2, 3'), 1, 2, 2, 3);
$table->attach_defaults(Gtk2::Button->new('1,4 -> 2, 4'), 1, 2, 3, 4);

$window->add($table);
$window->show_all;

Gtk2->main;

exit;

```

Listing 7

```

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

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

my $frame = Gtk2::Frame->new('Frame Title');

$frame->set_shadow_type('etched_in');

my $button = Gtk2::Button->new('Frame Contents');
$button->signal_connect('clicked', sub { Gtk2->main_quit });
$button->set_border_width(8);

$frame->add($button);

$window->add($frame);

$window->show_all;

Gtk2->main;

```

Listing 1

```
=1=      #!/usr/bin/perl -w
=2=      use strict;
=3=      $|++;
=4=
=5=      use Mac::Glue;
=6=      use XML::LibXML;
=7=
=8=
=9=      my $itunehandle = Mac::Glue->new("iTunes");
=10=
=11=     chdir or die $!;
=12=     chdir "irate" or die $!;
=13=
=14=     my $x = XML::LibXML->new or die;
=15=     my $d = $x->parse_file("trackdatabase.xml") or die;
=16=
=17=     for my $track ($d->findnodes(q{//Track[@rating and @file != ""]})) {
=18=         my $file = $track->findvalue('@file');
=19=         next unless -s $file;
=20=         my $rating = $track->findvalue('@rating');
=21=
=22=         my $irating =
=23=             ($rating >= 10) ? 100 :
=24=             ($rating >= 7) ? 80 :
=25=             ($rating >= 5) ? 60 :
=26=             ($rating >= 2) ? 40 :
=27=             0;
=28=
=29=         if ($irating) {
=30=             print "adding with $irating for $file\n";
=31=             my $s = $itunehandle->add($file);
=32=             $s->prop("rating")->set(to => $irating);
=33=         } else {
=34=             print "tossing $file\n";
=35=         }
=36=
=37=         ## next part is to fool it from downloading again
=38=         open F, ">$file" or warn "Cannot create $file: $!";
=39=         close F;
=40=     };
```