Building GTK apps with Glade

By Grant McLean

One advantage of the GTK GUI toolkit is the 'Glade' user interface builder. Glade allows you to build your UI and associate event handlers with widgets using point and click. This allows a clean separation of user interface definition and 'business logic'. It also completely eliminates much of the code required to build an interface.

If you've looked at a Perl program that uses a GUI toolkit then you may have found yourself looking at big ugly mess of code. It's common to find code for creating and laying out widgets mixed in with the callback code that will be triggered by different GUI events. As well as being a nightmare to maintain, it's also fairly tedious to write in the first place. If you're using the Gtk2 toolkit, the Glade GUI builder can save you from all that.

Recently, I needed some GUI code fast. The local Perl Mongers group was having a lightning talk evening and I wanted a simple application to time the talks and trigger sound effects to warn the speakers when their time was up. What I had in mind was something like Figure 1 (actually, what I had in mind was the slightly more complex version at the end of the article, but we have to start somewhere).

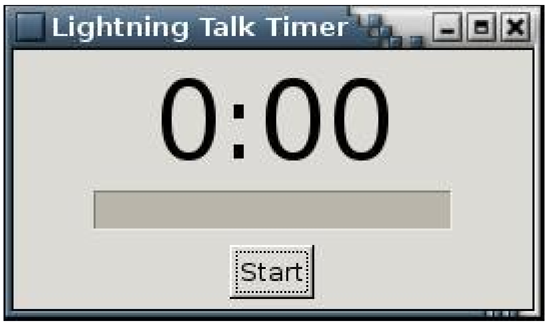


Figure 1 - A simple application window

This simple interface provides one button which will toggle between start and stop (just like a real stopwatch). It also provides a label and a progress bar to show the elapsed time.

Building the Interface

When I launched the Glade interface designer, three windows appeared - the project, palette and properties windows. I used the 'New' button on the toolbar in the main project window (Figure 2) to create a new project.

At this point Glade offered me the choice of creating a 'New GTK+ Project' or a 'New GNOME Project'. Since I didn't need any GNOME-specific widgets, I selected the GTK+ option.

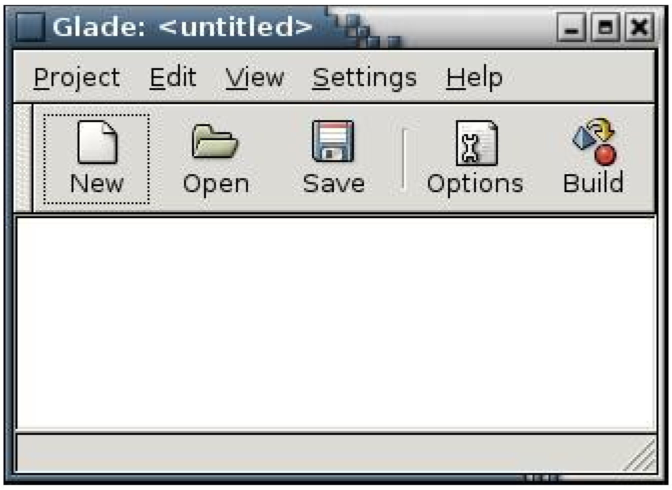
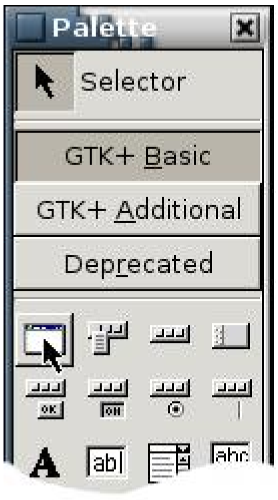


Figure 2 - The Glade project window

Next, I used the very first icon in Glade's widget palette to add an application window to the new project. This gave me a new blank window called 'window1' (Figure 3) to serve as the container for the user interface elements.



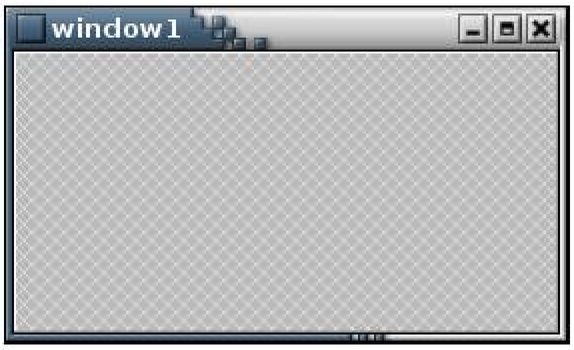
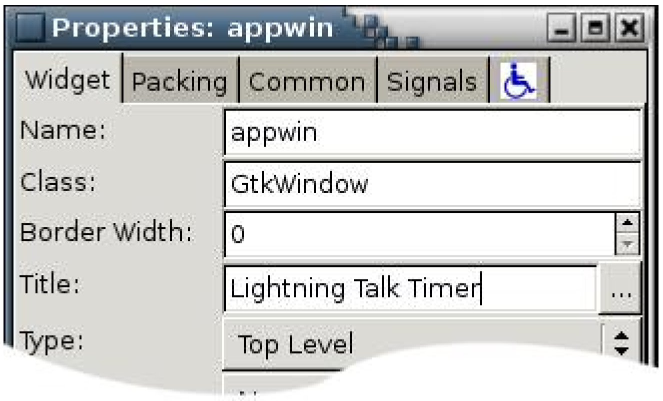


Figure 3 - Glade palette window and application window

Before adding any more widgets, I used Glade's properties window to change the widget name from 'window1' to 'appwin' and the title to 'Lightning Talk Timer' (Firgure 4).

Although it is possible to place widgets at fixed positions within a window, it's not a particularly good idea. An arrangement that looks good on your system may well look a mess on a system with different fonts installed or a different desktop theme. Instead, Gtk provides a number of layout widgets that act as containers for other widgets. The layouts automatically adjust for different font sizes and window sizes.



Name: appwin
Class: GtkWindow
Border Width: 0
Title: Lightning Talk Timer
Type: Top Level

Figure 4 - Glade properties window

I chose to use a 'Table' widget with one column of three rows. A 'VBox' widget would give a similar result, but I like to add padding to prevent the layout looking cramped, and that's easier with a table.

To add the table, I clicked once on the table icon in the palette and once in the application window (unfortunately you can't drag and drop). Using the same technique I added a label widget into the first row of the table, a progress bar into the second and a button into the third. The only gotcha in that sequence was that the icon for the progress bar widget appears on the 'GTK+ Additional' tab of the palette. This gave me the widgets I wanted but the layout needed to be smartened up (Figure 5).

I clicked to select the label widget. Then back over in the properties window, I set the 'Name' to 'timer_label' and the initial text to '0:00'. I also set the 'X Align' property to '0.50' to make the text horizontally centered within the allocated space ('0.00' would be left aligned and '1.00' would be right aligned). Using the 'Packing' properties tab I set the 'H Padding' and 'V Padding' to 4 pixels each and the 'X Expand', 'Y Expand', 'X Fill' and 'Y Fill' all to 'Yes'.

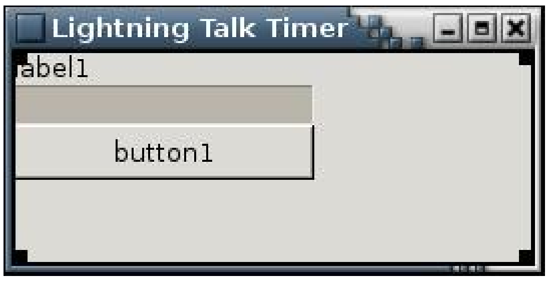


Figure 5 - Widgets added to application window

The progress bar required less work. I set the widget name to 'progress_bar', the 'H Padding' to 40 pixels and the 'X Fill' to yes. I left the 'Y Fill', 'X Expand' and 'Y Expand' properties set to 'No'.

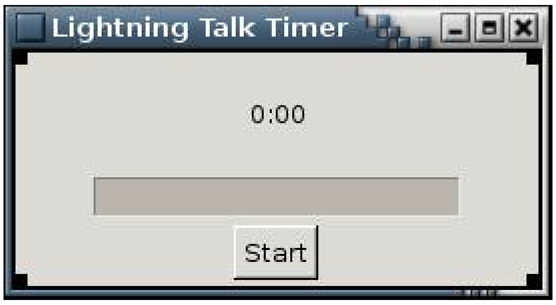


Figure 6 - After setting widget properties

Finally, I set the button's name to 'start_stop', label to 'Start', 'V Padding' to 4 pixels and 'X Fill' to 'No'.

The font on the label widget was still too small, but apart from that, the interface looked just the way I wanted (Figure 6). I saved the project to a file called `talk-timer.glade`.

Testing the Interface

To test the interface, I started with this simple Perl wrapper:

```
#!/usr/bin/perl -w

use strict;

use Gtk2 -init;
use Gtk2::GladeXML;

my $gui = Gtk2::GladeXML->new('talk-timer.glade');

Gtk2->main;
```

When I ran this script, my talk timer window popped up and looked almost useful. Unfortunately, when I closed the window, my script did not exit and I had to interrupt it with Ctrl-C. This is the default behaviour for Gtk2 scripts and the correct way to deal with it is to set up a handler which quits from the main event loop in response to the application window's 'delete_event' signal.

It might have been tempting at this point to start extending the wrapper script to set up the necessary event handling, but with Glade, there is a better way. By refactoring the wrapper script to represent the running application as an object, I can use Glade to turn GUI events into method calls.

I created `TalkTimer.pm` to contain the logic for an instance of the application:

```
package TalkTimer;

use strict;
use warnings;

use Glib qw(TRUE FALSE);
use Gtk2 -init;
use Gtk2::GladeXML;

sub run { my $self = shift->new; Gtk2->main; }

sub new { return bless({}, shift)->_init; }
sub _init {
  my $self = shift;

  my $glade_file = __FILE__;
  $glade_file =~ s/TalkTimer.pm/talk-timer.glade/;
  my $gui = Gtk2::GladeXML->new($glade_file);

  $gui->signal_autoconnect_from_package($self);

  return $self;
}

sub on_appwin_delete_event {
  Gtk2->main_quit;
}

1;
```

The `TalkTimer` class defines a `run` method which creates a `TalkTimer` object and then enters the Gtk2 event loop. The object constructor uses the `_init` method to locate and load the `.glade` file containing the user interface definition. Then `_init` uses GladeXML's `signal_autoconnect_from_package` method to turn each signal handler defined in the `.glade` file into a method call on the application object. The `on_appwin_delete_event` will be called when the window is closed and its job is to tell the event loop to quit.

Next, I altered the wrapper script to use the new application class:

```
#!/usr/bin/perl -w

use strict;

use FindBin;
use lib $FindBin::Bin;

use TalkTimer;

TalkTimer->run;
```

Finally, back in Glade, I set up the 'delete_event' signal handler. The first step was to select the application window widget. This is a little tricky since clicking anywhere in the application window either selects one of the three interface widgets or selects the table widget they are contained in. One solution is to right-click anywhere in the window and choose 'appwin->Select' from the pop-up menu. Another option is to use Glade's 'View' menu to show the 'Widget Tree' and use that to select 'appwin'.

Once the 'appwin' was selected, I went to the 'Signals' tab in the properties window. When I selected 'delete_event' from the pop-up list of signals, Glade automatically generated the `on_appwin_delete_event` method name, so all I needed to do was click 'Add' and then save the project.

This new version of the script exited cleanly when the window was closed.

Adding the Application Logic

To make the application fully functional, only one more signal needed to be hooked up. Back in Glade, I selected the 'start_stop' button and associated the 'clicked' event with the `on_start_stop_clicked` method (and saved the project again).

Then back in `TalkTimer.pm` I added some constants to define when the sounds should be played (I used smaller values during testing) and which sound files to use:

```
  use constant TIMER_MAXIMUM  => 5 * 60;
  use constant TIMER_WARNING  => 4 * 60;

  use constant WARNING_SOUND  => 'WARNING.WAV';
  use constant TIMES_UP_SOUND => 'GONG.WAV';
```

I also added some accessor methods for storing the application state:

```
  use base 'Class::Accessor::Fast';

  __PACKAGE__->mk_accessors(qw(
    label progress start_stop
    start_time running warned times_up
  ));
```

and fleshed out the `_init` method to save references to the interface widgets and to set the font on the timer label:

```
  sub _init {
    my $self = shift;

    my $glade_file = __FILE__;
    $glade_file =~ s/TalkTimer.pm/talk-timer.glade/;
    my $gui = Gtk2::GladeXML->new($glade_file);

    $gui->signal_autoconnect_from_package($self);

    $self->label     ($gui->get_widget('timer_label') );
    $self->progress  ($gui->get_widget('progress_bar'));
    $self->start_stop($gui->get_widget('start_stop')  );

    $self->label->modify_font(
      Gtk2::Pango::FontDescription->from_string("Sans 40")
    );

    return $self;
  }
```

Finally, I added the methods triggered by the Start/Stop button.

```
  sub on_start_stop_clicked {
    my $self = shift;

    if($self->running) {
      $self->stop_timer;
    }
  else {
    $self->start_timer;
  }
}

sub start_timer {
  my $self = shift;

  $self->start_time(time);
  $self->running(TRUE);
  $self->warned(FALSE);
  $self->times_up(FALSE);
  $self->start_stop->set_label('Stop');
  Glib::Timeout->add(1000, sub { $self->tick });
}

sub stop_timer {
  my $self = shift;

  $self->running(FALSE);
  $self->label->set_text('0:00');
  $self->progress->set_fraction(0);
  $self->start_stop->set_label('Start');
}

sub tick {
  my $self = shift;

  return FALSE unless $self->running;

  my $secs = time - $self->start_time;
  $self->label->set_text(
    sprintf('%d:%02d', int($secs/60), $secs % 60)
  );

  my $frac = $secs > TIMER_MAXIMUM ? 1 : $secs/TIMER_MAXIMUM;
  $self->progress->set_fraction($frac);

  if($secs >= TIMER_MAXIMUM  and  !$self->times_up) {
    $self->times_up(TRUE);
    $self->play_times_up_sound;
  }
  elsif($secs >= TIMER_WARNING  and  !$self->warned) {
    $self->warned(TRUE);
    $self->play_warning_sound;
  }

  return TRUE;
}
```

```
sub play_warning_sound  { play_sound(WARNING_SOUND);  };
sub play_times_up_sound { play_sound(TIMES_UP_SOUND); };

sub play_sound {
  my $filename = shift;

  system("play $filename >/dev/null 2>&1 &");
}
```

The `start_timer` method arranges for the `tick` method to be called after a one second timeout. As long as `tick` returns TRUE, the timeout callback will repeat at one second intervals. As the configured intervals are are reached, the `tick` method will arrange for the appropriate sounds to be played.

On my Linux laptop, I used the `play` command that comes with the SOX sound file translation package. On Windows, the `Win32::Sound` module provides a similar service using the standard Win32 APIs.

Extending the Timer App

Of course real life is never as tidy as a well written Perl script. The real version of the script I used had an interface that looked like Figure 7.

With this version, the sound effects could be switched to manual mode without interrupting the timer display. So at the operator's discretion, a particularly interesting speaker could be given a little extra time or a floundering speaker could be 'saved by the bell'.

If you want to try your hand at extending the app in this direction, you can add the extra widgets and set signal handler names using Glade. Then define your handlers in `TalkTimer.pm`. I used the `set_sensitive` method on the extra button widgets to enable/disable them as the checkbox was toggled.

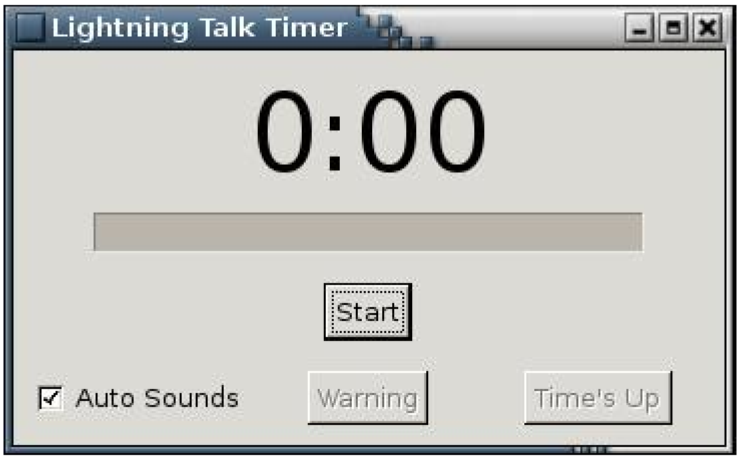


Figure 7 - Application with manual override widgets added

Packaging the App

Having the application split into three separate files is very convenient during development, but might not be ideal for deployment. You can move the definition of the `TalkTimer` package directly into the wrapper script. You could also put the contents of the `talk-timer.glade` file into a `__DATA__` section or a 'here doc' and initialize the UI with the `Gtk2::GladeXML->new_from_buffer` method.

Of course even this simple application requires a couple of extra media files for the sound effects. Larger and more complex applications might need custom icons and other image files. In those cases, you might want to use a platform specific packaging method such as .deb or RPM, or a Perl-specific format such as PAR.

Glade on Win32

The Gtk2::GladeXML module is available in PPM format for Microsoft Windows from the Gtk2-Perl project on SourceForge.

The Glade interface designer itself is available from the GladeWin32 project, also on SourceForge.

About the Author

Grant McLean `<grantm@cpan.org>` lives with his wife and two children in Wellington, New Zealand. He writes Perl code for a living and works on the Sprog project (http://sprog.sourceforge.net) for fun. He may never again be allowed to time lightning talks at Wellington.pm.