GUI development with wxGlade

Johan Vromans
Squirrel Consultancy

<jvromans@squirrel.nl>

In the beginning ...

In the beginning was the command line. To get the computer to do something, the user typed a command to the computer. In the early days, the commands were typed on punched paper tape, or punched cards. Later, the commands could be typed in at a type-writer-like interactive terminal.

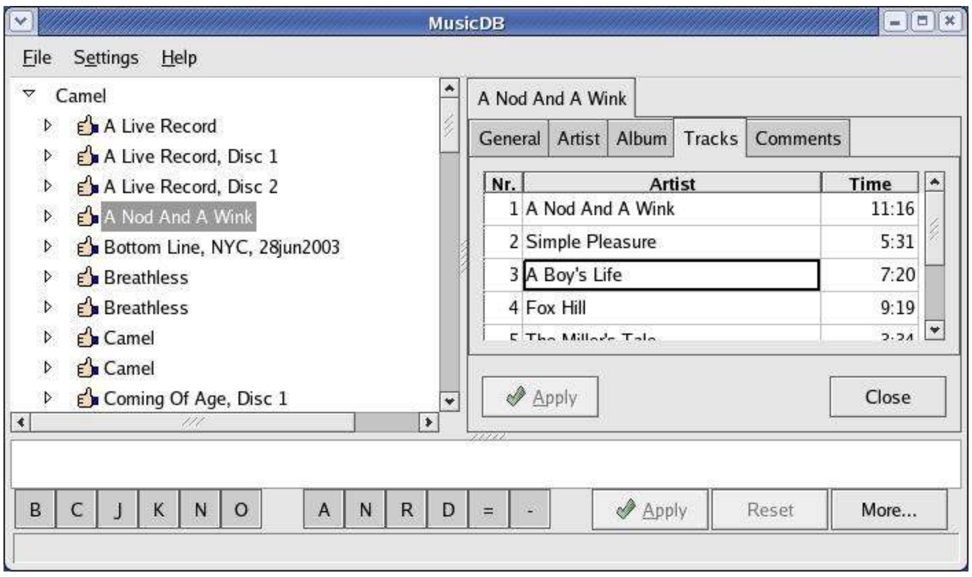
Although powerful, using this method to control a computer required knowledge of a vast collection of commands, with their arguments, options, and variations.

Around 1980 a new method to interact with a computer was developed: the Graphical User Interface, conveniently called GUI. GUIs displays visual elements such as icons, windows, and other gadgets. As we all know, the precursor to the GUI was invented by researchers at the Stanford Research Institute, and later brought to maturity by researchers at Xerox PARC. It was adapted by Apple in 1983, then by Microsoft in 1985, and the rest, as they say, is history.

Examples of systems that currently use GUIs are Mac OS, Microsoft Windows, NEXTSTEP, Palm OS and the X Window System.

GUI Basics

A typical GUI application looks something like this:



Nr.	Artist	Time
1	A Nod And A Wink	11:16
2	Simple Pleasure	5:31
3	A Boy's Life	7:20
4	Fox Hill	9:19

Generally, this is called the application window. It has a title bar at the top, with icons to close and resize it. In the window itself you can see a menu bar with pull-down menus, some windows with information, scrollbars, buttons, and a status bar at the bottom.

Technically, all the items on the screen are windows, some large, some small. Some contain other windows. A button is a small window that has a border and a text in it. A scrollbar is a window that has small triangle-buttons and a slider. The buttons and the slider are also windows. When a button is clicked, it is temporary replaced by another window that looks like a pressed button. Simultaneously, it has to trigger some application function.

To build an application by grouping windows like this would be a tedious job, and a great pain to accomplish. For this purpose, window systems have been providing toolkits with pre-constructed items called widgets. Using the appropriate toolkit, building the application becomes significantly easier. A button is now a widget that knows how to display its text, and automatically handles the changing appearance when clicked. The scrollbar is a single widget that knows how to move the slider, and feeds back on the actual position. But there is more. A scrollable window plus its scrollbars is itself a widget and knows how to deal with content and slider changes. Building a GUI application using a suitable toolkit becomes a feasible job.

Microsoft Windows and Mac OS have standard toolkits to be used for GUI applications.

For the X Window System many popular widget toolkits exist, such as Motif (CDE), Qt (KDE) and GTK+ (GNOME).

Another important toolkit is the wxWidgets[1] toolkit, that provides a platform-independent set of widgets implemented on X, Motif, GTK, Mac OS, Palm OS, Microsoft Windows, and several more.

A note on GUI design

The design of the GUI is very important since the GUI guides the user though the application logic. A badly designed GUI will confuse the user, making it hard to know what to do next. Fortunately, there are many good documents and books on this topic. The Macintosh user interface would not have become so successful without a strict style guide that all application were to follow. Projects like Gnome and KDE also have rather strict guidelines. Unfortunately, many application designers do not seem to read this information, or just assume they know better[2].

Introducing wxPerl

Most widget toolkits can be used from several programming languages, and wxWidgets is no exception. It is written in C++ but can be used from virtually any language that has the basic capabilities to interface with C++ libraries. A Perl extension called wxPerl allows Perl

1 Formerly called wxWindows.
2 See, e.g., http://www.frankmahler.de/mshame/

programs to use the wxWidgets toolkit. wxPerl is written and maintained by Mattia Barbon[3], hosted on SourceForge[4], and supported by an active group of users[5].

A typical wxPerl program consists of two parts: the main program, and the wxApplication. This is a user defined class that derives from the wxWidgets class `Wx::App`. The wxPerl program then uses it as follows:

```
# Create an instance of the Wx::App-derived class.
my $app = MyApp->new();

# Start processing events.
$app->MainLoop();
```

After calling `MainLoop`, Wx takes over control and starts listening for events. When an event happens, e.g., the user clicks on a button, Wx will dispatch control to the appropriate event handler and start waiting for the next event, and so on.

A simple application could consist of a simple window, a so called frame, and a piece of text. The definition of the `Wx::App` derived class, in this example `MyApp`, would look like:

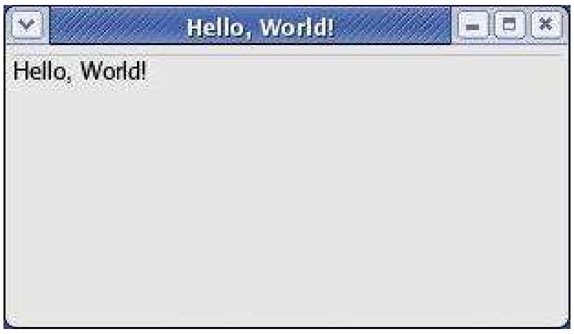
```
package MyApp;
use base qw(Wx::App);

sub OnInit {
  my ($self) = shift;
  # Create a new frame.
  my $frame = MyFrame->new();
  # Set it as top frame.
  $self->SetTopWindow($frame);

  # Show it.
  $frame->Show(1);
}
```

The only method defined in this class is `OnInit`, a method that gets called automatically when an instance of the class is created. In this example, `OnInit` instantiates the `MyFrame` class and displays it. `MyFrame`, as you can guess, implements the application window (Frame) and its contents:

```
package MyFrame;
use base qw(Wx::Frame);

# This imports some constants.
use Wx qw(wxDefaultPosition wxDefaultSize wxHORIZONTAL);

sub new {

  # New frame with no parent, id -1, title 'Hello, world!'.
  # Default position and size 300, 150.
  my $self = shift->SUPER::new(undef, -1, 'Hello, world!',
                               wxDefaultPosition , [350, 100]);

  my $sz = Wx::BoxSizer->new(wxHORIZONTAL);
  my $tx = Wx::StaticText->new($self, -1, 'Hello, World!',
                               wxDefaultPosition, wxDefaultSize);
  $sz->Add($tx);
  $self->SetSizer($sz);
  return $self;
}
```

3 mattia.barbon@libero.it
4 http://wxperl.sourceforge.net
5 wxperl-users@lists.sourceforge.net

Ignoring most details, the constructor of the `MyFrame` class creates a new Frame, a BoxSizer (a kind of generic container) and a StaticText widget. It connects the text widget and sizer to the frame and returns the newly created `MyFrame` object. When run, this is how it looks:



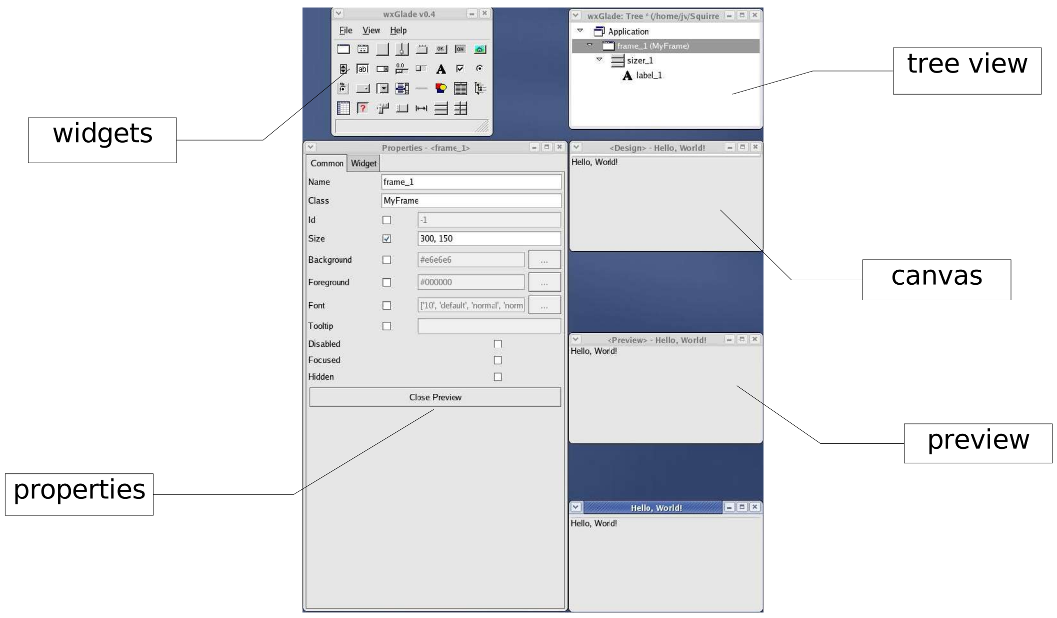
To turn this into a full application would require adding a vast amount of widgets, connect them, add the logic, and so on. While this is certainly doable (I've done it a couple of times ☺) it's a tedious job.

Introducing wxGlade

a GUI builder for wxWidgets

wxGlade is a user interface designer program for wxWidgets. It is written and maintained by Alberto Griggio[6], hosted on SourceForge[7] and actively supported by an active group of users[8]. It is capable of generating wxPerl, wxPython and C++ code.

wxGlade works like building blocks, you start with a blank piece of window, called a canvas, and you put the individual widgets on it.



Name		frame_1
Class		MyFrame
Id	☐	-1
Size	☑	300, 150
Background	☐	#e6e6e6
Foreground	☐	#000000
Font	☐	['10', 'default', 'normal', 'norm
Tooltip	☐	
Disabled	☐	
Focused	☐	
Hidden	☐	

6 alberto.griggio@gmail.com
7 wxglade.sourceforge.net
8 wxglade-general@lists.sourceforge.net

In this overview picture, you can see the most relevant components that play a part in wxGlade.

First, on the top left, the wxGlade widgets panel. It enumerates the widgets supported and you can click and drop widgets from here on the canvas.

Below the widgets panel is the properties panel. It displays a number of characteristics of the widget currently being worked upon.

At the right, on top, is the tree structure of the application. As you can see, the application consists of a frame (`frame_1`), that contains a sizer (`sizer_1`), that on its turn contains the static text widget (`label_1`).

Below the tree structure is the actual canvas being worked on, called the 'design view'. It more or less looks like the actual application, but it is not quite WYSIWYG since it contains some additional handles to play tricks with.

wxGlade provides for an exact preview, you can see it's window below the design view.

Finally, at the bottom, is the window of the actual application. Not surprisingly it looks exactly as the manually crafted application shown earlier (since that is what we tried to achieve). However, this version took just a couple of mouse clicks and keystrokes instead of having to write a lot of lines of code.

Maintenance and re-generation

Generating a program is just one step. As you can guess, the program logic, i.e., what must be done when the buttons are pushed, must still be filled in. And what if you need to change parts of the GUI afterwards?

wxGlade uses a well-known technique of 'guarded regions'. In the generated code, you can see several comments that are actually instructions for wxGlade. For example, a constructor:

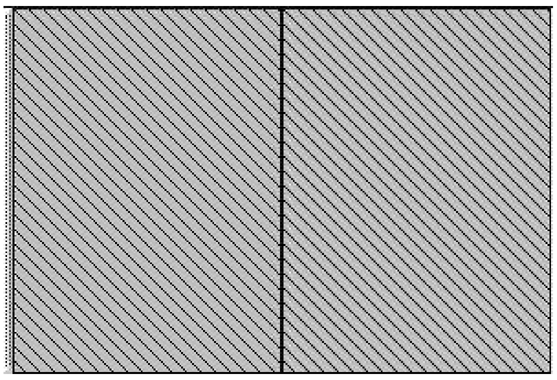
```
sub new {
        my( $self, $parent, $id, $title, $pos, $size, $style, $name ) = @_;
        ...
# begin wxGlade: MyFrame::new
        $style = wxDEFAULT_DIALOG_STYLE
                unless defined $style;

        ...
# end wxGlade
        return $self;
}
```

The parts between '`# begin wxGlade`' and '`# end wxGlade`' are private property. Every time the program is re-generated this part will completely be overwritten. On the other hand, everything outside these regions may be freely modified and augmented. This way it is possible to add application code to the program and still have wxGlade control its own parts.

Using sizers

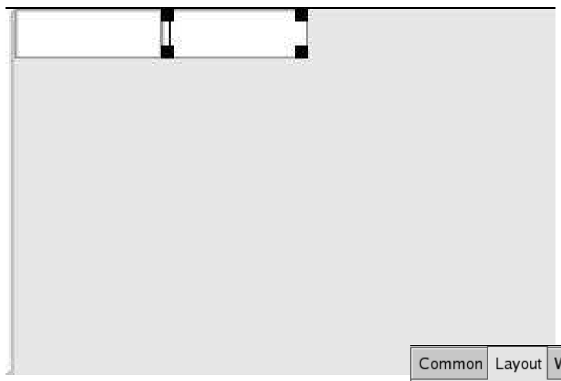
One of the major problems with building GUIs is that the user can, or at least should be able to, move and resize the windows. So it is not just a matter of placing the widgets on the canvas, more important is what to do when the canvas changes size. Several techniques have been developed over the years to solve this problem. Two popular techniques are springs, where the widgets are tied to the canvas, and to each other, using stretchable springs, and layout managers, where the widgets are laid out using external constraints like ‘all on top’, or ‘north east south west’.

wxGlade uses sizers for this purpose. As already mentioned, sizers are some kind of container. They can contain other widgets, including sizers. But sizers have two important properties, called *spread* and *expand*. To demonstrate these, let’s assume we have a canvas that contains a BoxSizer, that’s a sizer that divides the area into one or more horizontal or vertical areas. Initially, wxGlade shows this sizer like this:

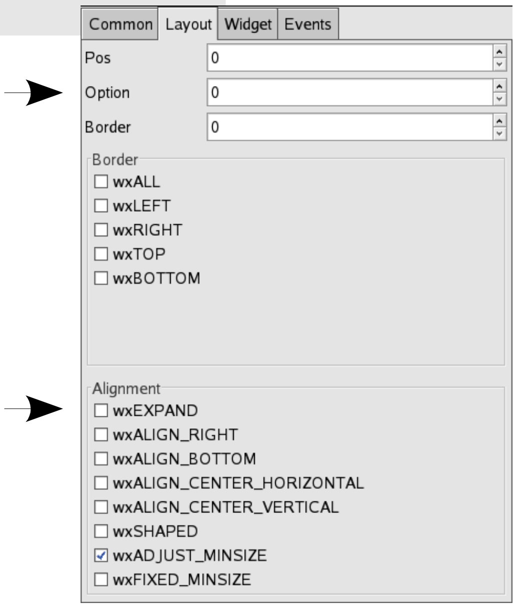


The area is (evenly) divided in two, empty, parts.

Now we drop a text widget into each of the areas. This is how it looks like now:



Since each of the text widgets has its own default size, which is rather small, the sizer shrinks to accommodate this. Now, let’s take a look at the properties of the text widget:



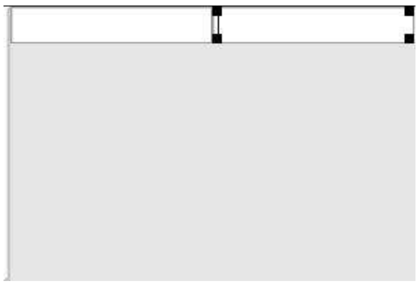
Pos	0
Option	0
Border	0

Border
- ☐ wxALL
- ☐ wxLEFT
- ☐ wxRIGHT
- ☐ wxTOP
- ☐ wxBOTTOM

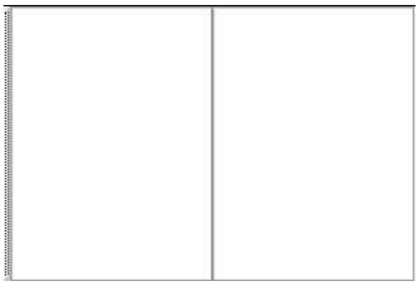
Alignment
- ☐ wxEXPAND
- ☐ wxALIGN_RIGHT
- ☐ wxALIGN_BOTTOM
- ☐ wxALIGN_CENTER_HORIZONTAL
- ☐ wxALIGN_CENTER_VERTICAL
- ☐ wxSHAPED
- ☑ wxADJUST_MINSIZE
- ☐ wxFIXED_MINSIZE

The important properties are ‘Option’ and ‘Expand’. ‘Option’ controls how the available space is distributed (spread) horizontally over the participating widgets. Currently, it is set to zero, indicating that no spread is applied for this widget, and its default size must be used. ‘Expand’ controls whether the widget will expand to the available space vertically.

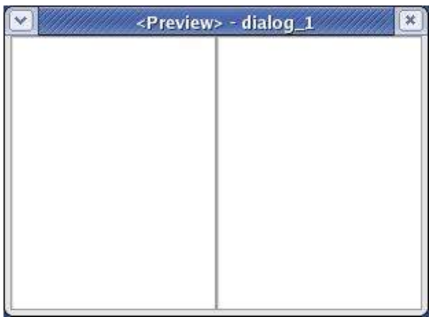
If we want both widgets to get equal space, we need to set 'Option' to an equal value, e.g., '**1**'.



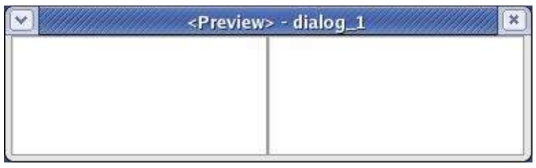
Finally, by checking 'Expand' for both widgets:



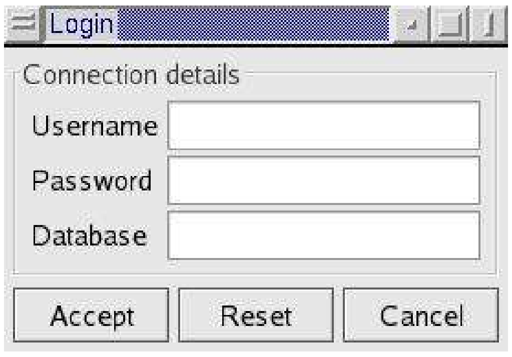
When run, or previewed, this is how it now looks like:



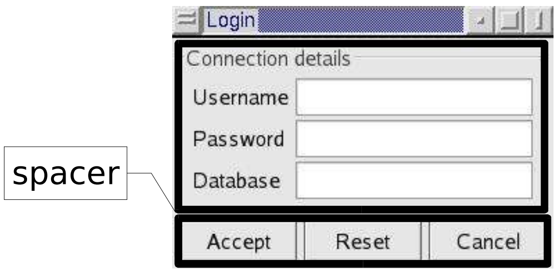
Now, we can resize the window, keeping the placement of the widgets in tact:



Complex situations can be accomplished by carefully crafting sizers within sizers. For example, this dialog:

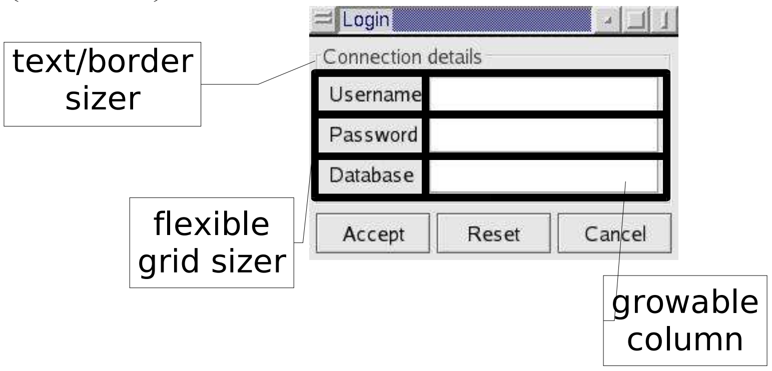


It is constructed as follows:

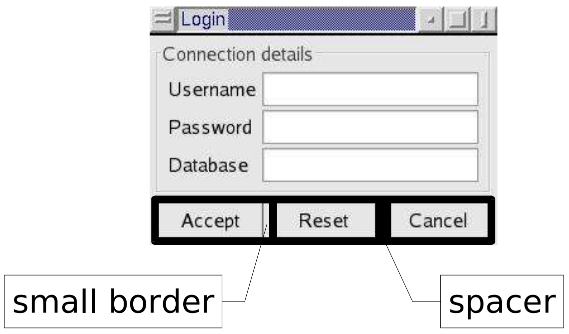


Just a vertical sizer with three slots, the middle slot contains stretchable space.

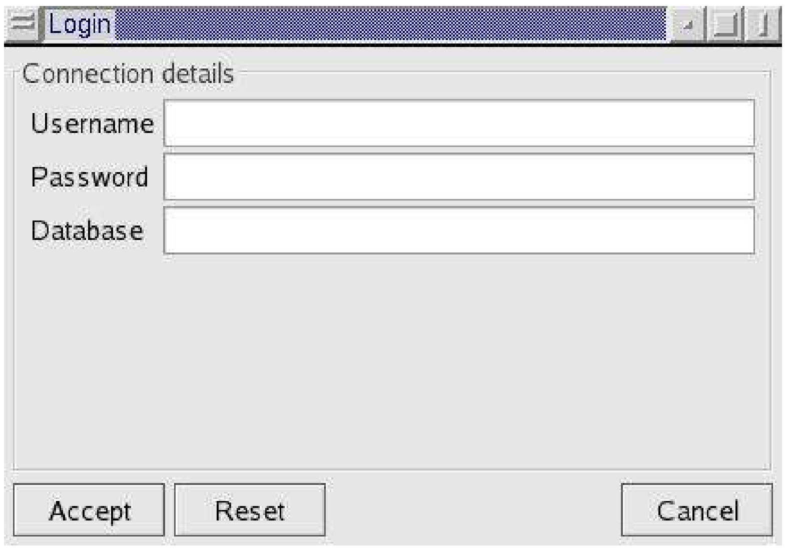
The top area contains a sizer with a text and border. This sizer is further divided using a GridSizer. It has rows and columns and you can specify one or more rows and columns to be growable (stretchable):



The bottom area is divided horizontally using a sizer, where the third slot is a spacer:



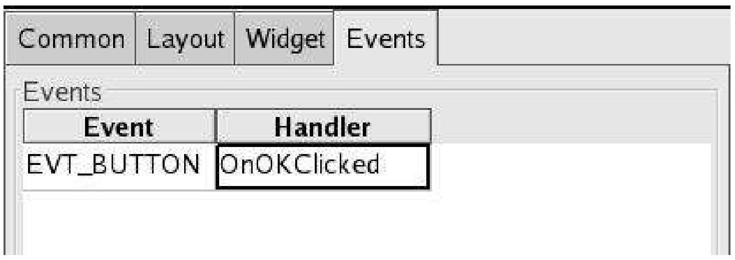
When we resize the window, we see all the parts play together:



Events

In a GUI application events play a crucial role. When the user clicks a button, or types some text in a text area, events are generated that must be acted upon by the application. There are many types of events, and complex widgets can generate even more complex event sequences, but the basic principle is the same: the application program has to designate an event handler for a particular event, i.e., a subroutine that gets called by the Wx framework when the event happens. And wxGlade can help with event handlers as well.

Examine a trivial button. On the property sheet, the 'Event' tab, there's a single event entry: '`EVT_BUTTON`'. This is the event that happens when the user clicks the button. Here we can supply the name of the callback routine that is to handle the event. Traditionally, these handlers have names like '`OnClicked`', '`OnClose`' and so on. We'll choose '`OnOKClicked`' here.



Event	Handler
EVT_BUTTON	OnOKClicked

When wxGlade has re-generated the code for the application, it has taken care of setting up the event handler, and also generated a piece of code for the handler, to be used as a starting point.

```
# wxGlade: MyDialog2::OnOKClicked <event_handler>
sub OnOKClicked {
        my ($self, $event) = @_;
        $event->Skip;
}
```

This subroutine effectively does nothing, '`$event->skip`' just means: not for me, pass this event to someone else who may be interested. A trivial thing to add would be a '`warn`' message:

```
# wxGlade: MyDialog2::OnOKClicked <event_handler>
sub OnOKClicked {
        my ($self, $event) = @_;
        warn("User pressed the OK button\n");
}
```

Conclusions

While building a GUI application still remains a time consuming task, much of the more tedious parts of the job can be handled by wxGlade. In fact, it would be a very bad idea to manually code the widgets, sizers, and events since the smallest change in design or layout might require enormous modifications. On the other hand, there's a limit to what a tool like wxGlade can do. In the end, it will always be the programmer who decides.