wxErlang - Getting Started

This booklet hopes to show you, the Erlang/OTP programmer, how to develop GUIs with wxErlang i.e the wx library that comes bundled with the Erlang distribution.

wx is more than a simple wrapper on top of the wxWidgets C++ library. It extends the interaction mechanism with the GUI elements and makes it seamless with the Erlang/OTP design principles.

On a Windows platform, wxWidgets comes bundled with the Erlang distribution, but on Linux or Mac, you need to first install wxWidgets. This booklet will not go into how to do that.

The approach is practical, hands on. We will tackle wxWidgets with examples, trying to introduce new concepts gradually. The coverage of wxWidgets is naturally incomplete as the intent is to help the Erlang programmer start using the library, not to make him an expert in wxWidgets.

Before delving into wxErlang, let's have a very brief look at what wxWidgets is.

wxWidgets

wxWidgets developers have have this to say:

> wxWidgets is an open source C++ framework allowing to write cross-platform GUI applications with native look and feel in C++ and other languages.
>
> wxWidgets was originally developed by Julian Smart at the Artificial Intelligence Applications Institute, University of Edinburgh, for internal use, and was first made publicly available in 1992, with a vastly improved version 2 released in 1999. The last major version of the library is 3 and was released in 2013. Currently wxWidgets is developed and maintained by Julian Smart, Vadim Zeitlin, Stefan Csomor, Robert Roebling, Vaclav Slavik and many others.
>
> More information about wxWidgets is available on its web site at http://www.wxwidgets.org.

Thanks wxWidgets. And thanks Erlang/OTP team for providing the binding.

wxWidgets takes care of more than just the GUI, but all that is mostly not relevant to Erlang, as Erlang/OTP already provides an adequate platform.

The GUI is made up of a number of windows or widgets, implemented as C++ classes. These classes are arranged in a hierarchy. Classes with more specialized properties and behaviours are derived from more abstract classes. At the top of the class hierarchy we have the wxWindow. I'll be using the term widget to mean an instance of some concrete class in the wxWindow class hierarchy.

Widgets can contain other widgets. In fact, widgets are mostly contained within another widget, the *parent*. However, there is no *pre-existing* widget, so every GUI does have at least one widget with no parents. Such a widget is a Top Level Window. Frames and Dialogs are Top Level Windows.

Every window is identified by a unique integer identifier. It is either provided by the programmer or the programmer requests wx to assign one. There are utility functions which allow you to obtain a reference to a widget whose identity you know.

Most functions (methods in the C++ implementation) that act upon a widget, automatically apply to

the widgets contained in it. For example, if you show a Frame widget, all widgets contained in it will also be shown.

Events are associated with the widgets and get created whenever something significant happens. For example, a *button click* event is generated when a Button is clicked. A *window close* event is generated when the user closes a window. These events are handled by functions connected to those events.

wxErlang

wxErlang is the Erlang binding provided by the Erlang/OTP team in the form of the wx library.

wxWidgets is a huge library. So the binding, and thus its documentation, is mostly machine-generated. The documentation mainly informs you about the available modules and the relative function signatures. It also points to the relative wxWidgets documentation. To get a hang of the meaning of the different functions and the arguments, you will often have to refer to the wxWidgets documentation. The sooner you get over this “inconvenience” and start looking at it, the better off you will be.

wxWidgets is more than just a GUI. It has a lot of support libraries. Those are not part of the wxErlang binding.

I'll assume you are an Erlang/OTP programmer. That you are familiar with Erlang Type Specification, behaviours, in particular gen_server, records, maps and proplists.

We'll first use examples built and run in the shell, and then move on to writing them in modules and running the compiled code.

Here's an outline of what to expect.

In the chapter Hello World, we deal with wx initialization and simple utility functions which do not require an understanding of wxWidgets. We point out the importance of reading both the wx and wxWidgets documentation.

In the chapter Have Your Say, we extend the example to getting input from the user. The chapter points out once again how important it is to read both the wx and the wxWidgets documentation.

In the chapter Countdown, we learn how to create a Frame and put some controls in it. We learn what events are and we see the callback model of handling those events. We also learn what the wx environment is and how to pass it on to other Erlang processes.

In the chapter Countdown Revistied, we introduce sizers. Sizers are pseudo-widgets that allow you to layout widgets based on constraints. They also help reposition and resize widgets when the frame is resized.

In the chapter Countdown Interaction Design Revisited, we introduce event handling in the Erlang process a la gen_server, instead of with callback functions.

In the chapter Chess Clock, we reuse the Countdown timer developed in the previous chapter to come with a chess clock for two players. We design the interaction by breaking up the application into three different processes, implemented in two different modules. We learn about things we have

to look out for such as the wx environment, the need to terminate the processes properly.

In the chapter wx_object, we introduce what I consider to be one of the more interesting abstractions in wxErlang. We see how it is practically an improvement upon the gen_server processes we already developed in the previous chapter. We also learn how the wx demo application is an excellent tool to learn about the whole library.

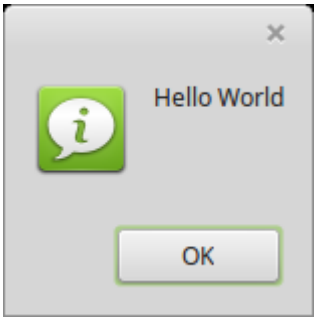
Hello World

Let's start by saying "hello". This is a message we send out to the user. And once she's ok with it, she dismisses the message by the click of a button.

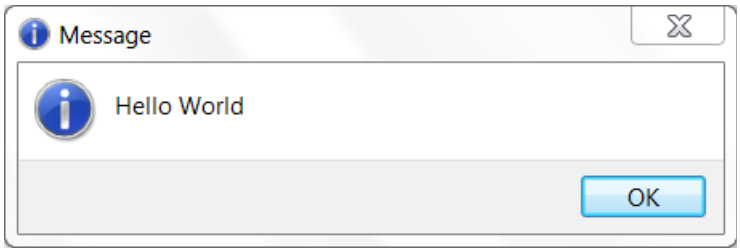
Open an Erlang shell and type the following commands:

```
1> wx:new().
{wx_ref,0,wx,[]}
2> M = wxMessageDialog:new(wx:null(), "Hello World").
{wx_ref,35,wxMessageDialog,[]}
3> wxMessageDialog:showModal(M).
```

If you are running Linux, you'll see something similar to the following:



The look and feel is from the native operating system. For exactly the same set of operations, under Windows you may get:



Click OK to dismiss the message.

Note that the Erlang shell blocked and waited for you to click Ok (or close the window otherwise).

The very first thing we did was to initialize the wx system with wx:new(). If you try creating a window, for example, without doing this, the shell will crash:

```
1> M = wxMessageDialog:new(wx:null(), "Hello World").
** exception error: {wxe,unknown_port}
     in function  wx:get_env/0 (wx.erl, line 133)
     in call from wxe_util:call/2 (wxe_util.erl, line 70)
```

While we are at it, take note of another thing. If you are using the shell, this crash will kill the shell and any processes linked to it. A new instance of the shell will then start handling your input.

If the shell or the process that started wx crashes, the initialization will get invalidated and you will have to do it again.

Next we created a Message Dialog. In the wxWidgets implementation, this would be done with a constructor of the wxMessageDialog class. In wx, we have the functions new/2 or new/3 exported by the module wxMessageDialog.

How do we know what this class does and which function to use? Unfortunately, this is where wxErlang is lacking. In the documentation for wxMessageDialog, you don't see anything about it. But if you do follow the link to the external documentation, you will get to the wxWidgets documentation and there you will see what this class is meant to do:

> This class represents a dialog that shows a single or multi-line message, with a choice of OK, Yes, No and Cancel buttons.

But what is a dialog? In the same wxWidgets documentation we see that wxMessageDialog is derived from the class wxDialog. In the documentation for wxDialog, we see what a dialog does:

> A dialog box is a window with a title bar and sometimes a system menu, which can be moved around the screen. It can contain controls and other windows and is often used to allow the user to make some choice or to answer a question.
>
> The dialog usually contains either a single button allowing to close the dialog or two buttons, one accepting the changes and the other one discarding them

Ok, but which function do we use?

If we look at the wxWidgets documentation for class wxMesageDialog, we'll find that there is only one constructor with five different arguments, three of which are optional, having default values. The module wxMessageDialog, however, exports two constructor functions: new/2 and new/3. new/2 uses all the default values of the optional arguments and new/3 allows you to specify any or all of those options in the third argument. This, IMHO is quite elegant.

So we used new/2, which is specified as:

```
new(Parent, Message) -> wxMessageDialog()
      Types
      Parent = wxWindow:wxWindow()
      Message = unicode:chardata()
```

The first argument is a reference to a widget, the second a unicode string. This is a dialog and thus a top level widget. It can exist without a parent and that's what we have done. We would specify this fact in wxWidgets with a null pointer, but in Erlang we don't have null pointers. So wxErlang provides the function null/0 in module wx to represent it. That's why we used the function new/2 to create a message dialog and set the first argument to wx:null(), indicating it has no parent:

```
M = wxMessageDialog:new(wx:null(), "Hello World").
```

The constructor function returns a reference to a wxMessageDialog widget and we bind the variable M to that reference.

Next we want to display the dialog. The module wxMessageDialog does not show any function that sounds suitable, so we try in the parent module's API. The parent module is wxDialog.

But wait, how do we know all this? The module documentation states:

```
This class is derived (and can use functions) from:
      wxDialog
      wxTopLevelWindow
      wxWindow
      wxEvtHandler
```

That's how!

We have a few functions which seem plausible: show/1, show/2 and showModal/1. But what's the difference and which one shall we use. We go back to the good old wxWidgets documentation of wxMessageDialog and voila! it says quite clearly what to do:

> Use wxMessageDialog::ShowModal to show the dialog.

Thank you. And that's what we did.

Have your say

We'll continue with the previous example to illustrate a few more peculiarities of wxErlang. The previous example used just one button to acknowledge the message. In this example we'll use two buttons to obtain the user's response to a question.

Going back to the wxMessageDialog constructor, we saw that new/3 had three arguments and that the third was a propslist of optional arguments. The wx documentation of new/3 specifies the third argument, Option, as::

```
Option = {caption, unicode:chardata()} | {style, integer()} | {pos,
{X::integer(), Y::integer()}}
```

There is a unicode string, caption. There is an integer, style. And there is a 2-tuple of integers, pos. But what do they mean? We can probably guess what caption and pos mean, but what is style? Especially since it is an integer.

Again, we have to turn to the external documentation, ie wxWidgets, to get an answer. The lesson once again is: refer to wx documentation for the function specification, but to get the details and the meaning, refer to the wxWidgets documentation.

We see that style can be one of several constants:

wxOK	Show an OK button.
wxCANCEL	Show a Cancel button.
wxYES_NO	Show Yes and No buttons.
wxYES_DEFAULT	Used with wxYES_NO, makes Yes button the default - which is the default behaviour.
wxNO_DEFAULT	Used with wxYES_NO, makes No button the default.

wxICON_EXCLAMATION	Shows an exclamation mark icon.
wxICON_HAND	Shows an error icon.
wxICON_ERROR	Shows an error icon - the same as wxICON_HAND.
wxICON_QUESTION	Shows a question mark icon.
wxICON_INFORMATION	Shows an information (i) icon.

These constants represent bit masks and can be or'ed together, whenever that makes sense.

In wxErlang, such constants are “defined” as macros in the wx.hrl header file of the wx library. In our code, therefore, we will use ?wxOK for the constant wxOK and ?wxICON_EXCLAMATION for the constant wxICON_EXCLAMATION.

To include the header file, we will add

```
-include_lib(“wx/include/wx.hrl”).
```

to our modules.

In the shell, however, we cannot include header files, so we'll have to take a look at the header file itself and see what these constants correspond to.

We find where wx is installed by usign the get_path/0 function of module code:

```
7> code:get_path().
```

For example, on my machine I see:

```
7> code:get_path().
[".","/usr/local/lib/erlang/lib/kernel-5.4/ebin",
 "/usr/local/lib/erlang/lib/stdlib-3.4.2/ebin",
 "/usr/local/lib/erlang/lib/xmerl-1.3.15/ebin",
 "/usr/local/lib/erlang/lib/wx-1.8.2/ebin",
```

So, wx.hrl must be located[1] at /usr/local/lib/erlang/lib/wx-1.8.2/include/wx.hrl

In it, we can see:

```
-define(wxICON_QUESTION, 1024).
-define(wxICON_EXCLAMATION, 256).
-define(wxNO_DEFAULT, 128).
-define(wxYES_DEFAULT, 0).
-define(wxCANCEL, 16).
-define(wxNO, 8).
-define(wxOK, 4).
-define(wxYES, 2).
```

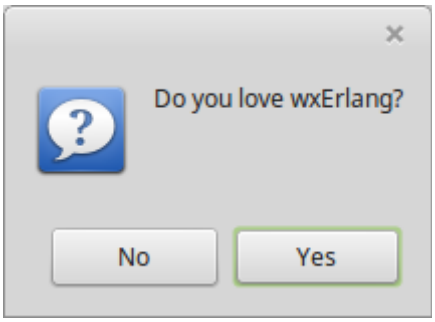
Let's see if we can use this information to display a question. We'll want the Question icon (1024) and we'll want two buttons, one saying yes (2) and one saying no (8), with default button the one saying yes (0). Remember that these integers have to be treated as bitstrings and that the or'ing operation on bitstrings in Erlang is bor.

```
11> M = wxMessageDialog:new(F, "Do you love wxErlang?", [{style, 1024 bor 2 bor
8 bor 0}]).
```

1 Erlang/OTP assumes that all applications will use a conventional directory structure with header files in directory include, source code in directory src and compiled code in directory ebin.

```
{wx_ref,39,wxMessageDialog,[]}
12> wxMessageDialog:showModal(M).
```

And what we get is:



As in any popular GUI, you can click on the buttons or you can hit the Enter key. You can select the button you wish to press by using the arrow keys or the Tab key and you will see the “focus” moves from one key to the other.

Make your choice!

How do we know what your choice was?

```
18> wxMessageDialog:showModal(M).
5103
```

When the dialog closes, a value is returned. In the example above it is 5103. In wx.hrl, we see that that value is associated with the constant wx_YES:

```
-define(wxID_YES, 5103).
-define(wxID_NO, 5104).
```

How did we know the function showModal/1 returns an integer value? It says so in the wxDialog module documentation:

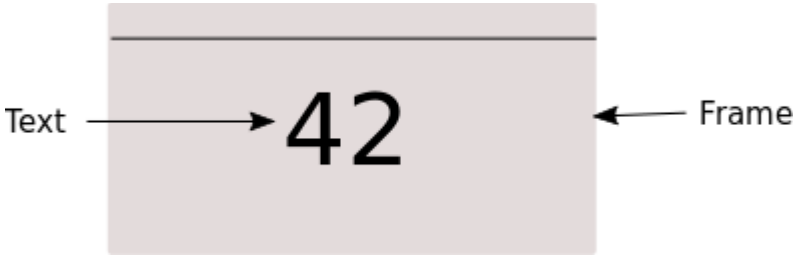
```
showModal(This) -> integer()
      Types
      This = wxDialog()
```

The Message Dialog is just one of several dialogs that are part of the wxWidgets distribution. There's a ColourDialog to select colours. A DirDialog to select a directory. A FileDialog to select a file. A FontDialog to pick a font. Even a PasswordEntryDialog.

Countdown

In this chapter we move on to constructing our own GUI.

To start simple, we shall construct a countdown timer. We will start with a certain number and then count down to zero.

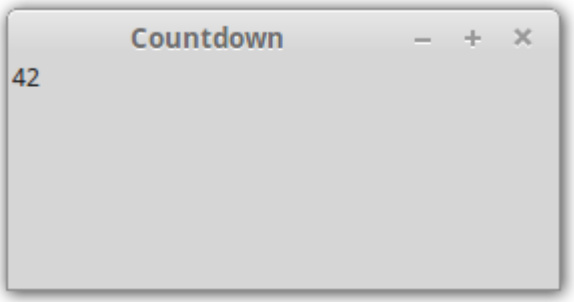


We will create a Frame and we will put a Text inside it. We change the text every second to make it look like a timer which is counting down.

Here is how we can get a Top Level Window that shows the starting count.

```
1> wx:new().
{wx_ref,0,wx,[]}
2> F = wxFrame:new(wx:null(), -1, "Countdown").
{wx_ref,35,wxFrame,[]}
3> T = wxStaticText:new(F, -1, "42").
{wx_ref,36,wxStaticText,[]}
4> wxFrame:show(F).
```

And this is what we might get:



First we examine this bit and then we'll see how to make it do the countdown.

We create a Top Level Window, a Frame, to contain the GUI. We use the function new/3:

```
new(Parent, Id, Title) -> wxFrame()
```

It has no parents, so we use wx:nul(). We could give it any Id or let wxWidgets choose one. We indicate that with the value -1 in the shell. In a module we would use the macro ?wxID_ANY which evaluates to -1 in wx.hrl. The title is any string. We used “Countdown”.

Next we create the counter. This can be achieved with a wxStaticText widget. The constructor function is new/3:

```
new(Parent, Id, Label) -> wxStaticText()
```

This is not a top level window and we want it contained in the frame, so the parent is the frame we

created.

To do the actual countdown, we need to call some function which will decrement the counter value by one, every second, until it gets to zero. We could do it in the shell, but let's start with writing our example programs in modules. Here's module countdown_gui:

```
-module(countdown_gui).
-export([start/1]).
-include_lib("wx/include/wx.hrl").

start(Seconds) when is_integer(Seconds) ->
    wx:new(),
    Frame = wxFrame:new(wx:null(), ?wxID_ANY, "Countdown"),
    Counter = wxStaticText:new(Frame, ?wxID_ANY, integer_to_list(Seconds)),
    wxFrame:show(Frame),
    countdown(Seconds -1, Counter),
    timer:sleep(10000).

countdown(Seconds, _) when Seconds < 0 ->
    ok;
countdown(Seconds, Counter) ->
    timer:sleep(1000),
    wxStaticText:setLabel(Counter, integer_to_list(Seconds)),
    countdown(Seconds-1, Counter).
```

The code is fairly obvious. We include the header file wx.hrl to get the wx macros. We use the function setLabel/2 of module wxStaticText to set the text to the new value after a second.

We can compile this module and run the timer starting with any value.

```
13> c(countdown_gui).
{ok,countdown_gui}
14> countdown_gui:start(3).
ok
```

The GUI should pop up. The counter should go from 3 to 0 and after 10 seconds we should get the prompt back in the shell. The frame is still visible, so we didn't really need that waiting for 10 seconds.

We could, however, spawn the GUI in a separate process.

```
15> spawn(countdown_gui,start, [3]).
<0.126.0>
16>
```

This will give us the prompt immediately and 10 seconds after the countdown, the frame will disappear. Why? The spawned process terminates clearing away whatever it had created, including the frame. It is good practice to terminate wx properly.

The GUI does its job, but is not very practical. We should perhaps be able set a start value from the GUI itself and perhaps be able to stop the timer and then resume the countdown at will. Much like a clock for chess players.

Here's a go at what this might look like:

We start with the counter “editable”, meaning we can enter a different value, and a button labelled Start to start the countdown. Once it starts counting, the counter itself is no longer editable and the button label changes to Stop. On getting to zero, once again the counter value is editable and the button label changes to Start.

We begin with the layout. We need two widgets in the frame this time. One editable text widget and one button widget.

The editable widget is a wxTextCtrl. Here's the description from the wxWidgets documentation:

> A text control allows text to be displayed and edited. It may be single line or multi-line.

We can create one with new/3:

```
Counter = wxTextCtrl:new(Frame, ?wxID_ANY,
                         [{value, integer_to_list(Seconds)}]),
```

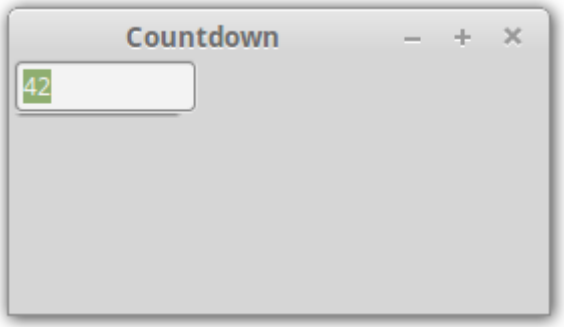
The button widget is a wxButton:

> A button is a control that contains a text string, and is one of the most common elements of a GUI. It may be placed on a dialog box or panel, or indeed almost any other window.

We can create one with new/3:

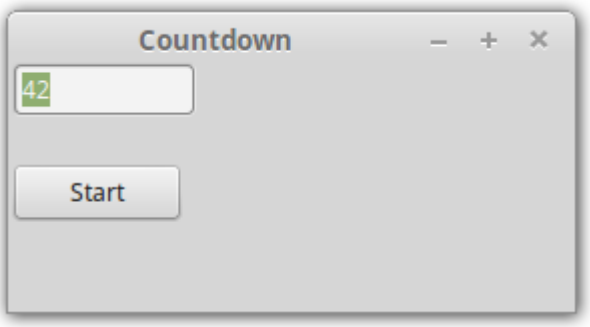
```
Button = wxButton:new(Frame, ?wxID_ANY, [{label, "Start"}]),
```

If we now show the frame, we get something like:



This is not what we wanted. The counter is hiding the button. We can remedy this by positioning the button explicitly in the constructor, or moving it to some other place with move/3 of module wxWindow:

```
Button = wxButton:new(Frame, ?wxID_ANY, [{label, "Start"}, {pos, {0, 50}}]),
```



This works, but is not very elegant. The position has to be determined before hand and may not give the expected result on different platforms. Also, what if we change the font size of the counter (which we shortly will). We'd have to recalculate the position. wxWidgets does provide much more elegant means to lay out our widgets and we will see that in the next example.

Before going any further, let's at least make the counter bigger. We do that by setting the font for the text control. In the wxWindow module, we find the specification:

```
setFont(This, Font) -> boolean()
      Types
      This = wxWindow()
      Font = wxFont:wxFont()
```

However, we must supply a reference to a wxFont widget. Following the link to wxFont, we find a number of constructor functions. We'll use new/4

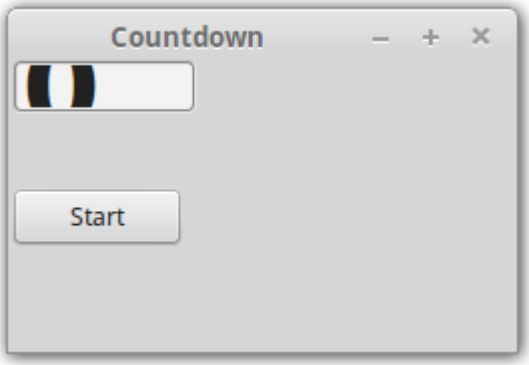
```
new(Size, Family, Style, Weight) -> wxFont()
```

The possible values for Family, Style and Weight are enumerated in wx.hrl as macros (or in the wxWidgets documentation):

```
% From "font.h": wxFontFamily
-define(wxFONTFAMILY_DEFAULT, ?wxDEFAULT).
-define(wxFONTFAMILY_DECORATIVE, ?wxDECORATIVE).
-define(wxFONTFAMILY_ROMAN, ?wxROMAN).
-define(wxFONTFAMILY_SCRIPT, ?wxSCRIPT).
-define(wxFONTFAMILY_SWISS, ?wxSWISS).
-define(wxFONTFAMILY_MODERN, ?wxMODERN).
-define(wxFONTFAMILY_TELETYPE, ?wxTELETYPE).
-define(wxFONTFAMILY_MAX, (?wxTELETYPE+1)).
-define(wxFONTFAMILY_UNKNOWN, ?wxFONTFAMILY_MAX).
% From "font.h": wxFontStyle
-define(wxFONTSTYLE_NORMAL, ?wxNORMAL).
-define(wxFONTSTYLE_ITALIC, ?wxITALIC).
-define(wxFONTSTYLE_SLANT, ?wxSLANT).
-define(wxFONTSTYLE_MAX, (?wxSLANT+1)).
% From "font.h": wxFontWeight
-define(wxFONTWEIGHT_NORMAL, ?wxNORMAL).
-define(wxFONTWEIGHT_LIGHT, ?wxLIGHT).
-define(wxFONTWEIGHT_BOLD, ?wxBOLD).
-define(wxFONTWEIGHT_MAX, (?wxBOLD+1)).
```

We'll try the dafault family, normal style, bold weight and point size 42.

If we try this, we get the following window:



The font is big, but the text control remained the same size. We need to adjust that size either in the text control constructor, or explicitly later with function setSize/3 of module wxWindow.

Let's now see what the use cases are. We start with some value and the button labelled Start. The counter value, whatever it is, should be editable. When we click the button and the counter value is not zero, the timer should start. This means the counter value should no longer be editable, the button label should change to Stop and the count value should start decreasing every second. If we click the button before the count gets to zero, the counting down should stop, the button label should change to Start and the counter value should again be editable. Lastly, if the count goes down to zero, the counting down should stop, the button label should change to Start and the counter value should again be editable.

Before we go into implementing this, let's look into how we'll ever get to know the button has been clicked.

A button click is one of many events wxWidgets handles. Each event is captured in a data structure with information about the event. Widgets generate the events; widgets handle the events. To handle events, widgets must subscribe to events. This is done by using the connect/2 or connect/3 functions of the wxEvtHandler module. All widgets derive from wxEvtHandler, so all widgets can subscribe to events in order to handle them.

In wx, when a widget subscribes to a type of event, it can specify the id of the widget that generated the event and a handler function. When the event is generated, that function, the *callback* function, will be invoked. If no handler function is specified, the event will be sent to the *process* that subscribed to the event. The callback function takes two arguments: an event record and a reference to the event object, in case additional information about the event is needed.

The event record is a wx record, defined in the wxEvtHandler module documentation as:

```
#wx{id=integer(), obj=wx:wx_object(), userData=term(), event=event()}
```

So, when a button is clicked, a command_button_clicked event will be generated. How do we know *that* is the event that will be generated? The documentation of module wxEventHandler lists all the event types (and we can find them in wx.hrl as well). If we search for click, we'll find there is an event command_button_clicked.

I think the documentation on all this is a bit unclear and confusing and wxWidget documentation does not help since the event handling mechanism is improved in wxErlang. When I'm unclear about some things, as in this case I am about what arguments are passed to the callback function, I try it out in the shell. So here's a session in the shell in which I create a button and subscribe to the button

clicked event. I specify a callback function with two arguments. Within the function I just print out those two arguments. When I click the button, I see what the two arguments are:

```
1> wx:new().
{wx_ref,0,wx,[]}
2> F = wxFrame:new(wx:null(), -1, "frame").
{wx_ref,35,wxFrame,[]}
3> B = wxButton:new(F, 1001, [{label, "click me"}]).
{wx_ref,36,wxButton,[]}
4> wxFrame:show(F).
true
5> CB = fun(A,B) -> io:format("called back with A:~n~p~nB:~n~p~n", [A,B]) end.
#Fun<erl_eval.12.99386804>
6> wxButton:connect(B, command_button_clicked, [{callback, CB}, {userData,
some_user_data}]).
ok
called back with A:
{wx,1001,
    {wx_ref,36,wxButton,[]},
    some_user_data,
    {wxCommand,command_button_clicked,[],0,0}}
B:
{wx_ref,38,wxCommandEvent,[]}
7> rr ("/usr/local/lib/erlang/lib/wx-1.8.2/include/wx.hrl").
[wx,wxActivate,wxAuiManager,wxAuiNotebook,wxCalendar,
 wxChildFocus,wxClipboardText,wxClose,wxColourPicker,
 wxCommand,wxContextMenu,wxDate,wxDisplayChanged,wxDropFiles,
 wxErase,wxFileDirPicker,wxFocus,wxFontPicker,wxGrid,wxHelp,
 wxHtmlLink,wxHtmlLinkInfo,wxIconize,wxIdle,wxInitDialog,
 wxJoystick,wxKey,wxList,wxMaximize|...]
8> {wx,1001,
8>     {wx_ref,36,wxButton,[]},
8>     some_user_data,
8>     {wxCommand,command_button_clicked,[],0,0}}.
#wx{id = 1001,
    obj = {wx_ref,36,wxButton,[]},
    userData = some_user_data,
    event = #wxCommand{type = command_button_clicked,
                       cmdString = [],commandInt = 0,extraLong = 0}}
9> wxButton:getLabel({wx_ref,36,wxButton,[]}).
"click me"
10>
```

Let me go through this.

Line 1, we initiate `wx`.

Line 2, we create a frame.

Line 3, we create a button in that frame. Note we've used the id 1001.

Line 4, we show the frame and in it our button. I leave it to you to discover any surprise in the layout.

Line 5, we define an anonymous function that takes two arguments and simply prints them out.

Line 6, we subscribe to the `command_button_clicked` event. We specify a callback function. We also specify some user data as this is probably a good point in time to introduce the concept.

Now we click the button and get the highlighted part. We can see that the first argument was indeed a `wx` record and that the second is a reference to `wxCommandEvent`.

Line 7, The wx record is printed as a tuple, since a record is but a tuple. In the shell we have the possibility of printing out tuples in the form of records if the shell is provided the record information. In this line then, we provide that information using the shell command rr.

Line 8, we copy and paste the printed out wx record to have the shell spew out the same formatted as a record. We can see that the id is the id we assigned to the button. The widget associated with the event is a wxButton. We can use that reference in functions, as we do in line 9, to see what the label of the clicked button is. We can see that the user data is exactly what we provided in the subscription.

One thing still remains to be decided. How do we interrupt the countdown loop if the button is clicked? Actually, we can't. At least not with the way it is implemented. So we actually employ an Erlang timer to update the count every second. The timer will apply some function, say update_gui. Within that function, if the count is not zero and the button is not labelled Start, we'll decrement the count by one and start a fresh Erlang timer to invoke update_gui after one second.

With all that decided, here's our go at two functions: update_gui and handle_click:

```
handle_click(#wx{obj = Button,
            userData = #{counter := Counter}},
          _Event) ->
    Label = wxButton:getLabel(Button),
    case list_to_integer(wxTextCtrl:getValue(Counter)) of
      0 when Label =:= "Start" ->
          ok;
      _ when Label =:= "Start" ->
          wxTextCtrl:setEditable(Counter, false),
          wxButton:setLabel(Button, "Stop"),
          timer:apply_after(1000, ?MODULE, update_gui, [Counter, Button]);
      _ when Label =:= "Stop" ->
          wxTextCtrl:setEditable(Counter, true),
          wxButton:setLabel(Button, "Start")
    end.

update_gui(Counter, Button) ->
    case wxButton:getLabel(Button) of
      "Stop" ->
          Value = wxTextCtrl:getValue(Counter),
          case list_to_integer(Value) of
            1 ->
                wxTextCtrl:setValue(Counter, "0"),
                wxTextCtrl:setEditable(Counter, true),
                wxButton:setLabel(Button, "Start");
            N ->
                wxTextCtrl:setValue(Counter, integer_to_list(N-1)),
                timer:apply_after(1000, ?MODULE, update_gui, [Counter, Button])
          end;
      "Start" ->
          ok
    end.
```

We will use handle_click/2 as the button callback function. A reference to the button will be passed to it through the wx record, but the function will also need a reference to the counter, so we pass that in the user data.

When the countdown timer should be counting down, we program an update of the GUI after a second. The update function too will need references to both the button and the counter, so we pass

those to the function as arguments.

It looks all set, but it won't work. The reason is that the Erlang timer runs the functions programmed to execute after some time in a separate process. We all know how jealous Erlang processes are with their variables. One process cannot access variables from another process. So to permit this, wxErlang provides a mechanism. It is the wx *environment*. You get the wx environment, with get_env/0, in the process in which you started wx with new/0, and then you pass it on to other processes. Those processes will then set *their* wx environment, with set_env/1, to the one passed, in order to be able to use the references to the widgets in the parent environment.

We therefore modify the two functions to make use of the wx environment and subscribe to button clik events when setting up the countdown timer. The complete module with all these modifications is:

```
-module(countdown_gui2).

-export([start/0]).
-export([handle_click/2, update_gui/3]).

-include_lib("wx/include/wx.hrl").

start()  ->
    wx:new(),
    Frame = wxFrame:new(wx:null(), 1, "Countdown"),

    %% build and layout the GUI components
    Counter = wxTextCtrl:new(Frame, ?wxID_ANY, [{value, "42"}, {size, {150,
50}}]),
    Font = wxFont:new(42, ?wxFONTFAMILY_DEFAULT, ?wxFONTSTYLE_NORMAL, ?
wxFONTWEIGHT_BOLD),
    wxTextCtrl:setFont(Counter, Font),

   Button = wxButton:new(Frame, ?wxID_ANY, [{label, "Start"}, {pos,{0, 64}},
{size, {150, 50}}]),

    wxButton:connect(Button, command_button_clicked, [{callback, fun
handle_click/2}, {userData, #{counter => Counter, env => wx:get_env()}}]),

    wxFrame:show(Frame).

handle_click(#wx{obj = Button, userData = #{counter := Counter, env := Env}},
          _Event) ->
    wx:set_env(Env),
    Label = wxButton:getLabel(Button),
    case list_to_integer(wxTextCtrl:getValue(Counter)) of
      0 when Label =:= "Start" ->
          ok;
      _ when Label =:= "Start" ->
          wxTextCtrl:setEditable(Counter, false),
          wxButton:setLabel(Button, "Stop"),
          timer:apply_after(1000, ?MODULE, update_gui, [Counter, Button, Env]);
      _ when Label =:= "Stop" ->
          wxTextCtrl:setEditable(Counter, true),
          wxButton:setLabel(Button, "Start")
    end.

update_gui(Counter, Button, Env) ->
    wx:set_env(Env),
    case wxButton:getLabel(Button) of
      "Stop" ->
```

```
          Value = wxTextCtrl:getValue(Counter),
          case list_to_integer(Value) of
            1 ->
                wxTextCtrl:setValue(Counter, "0"),
                wxTextCtrl:setEditable(Counter, true),
                wxButton:setLabel(Button, "Start");
            N ->
                wxTextCtrl:setValue(Counter, integer_to_list(N-1)),
                timer:apply_after(1000, ?MODULE, update_gui, [Counter, Button,
Env])
          end;
       "Start" ->
          ok
    end.
```

Go ahead. Compile it and run the start/0 function. It should work the way we expect.

There still remain several problems with this. In the following sections we will first try handling the events directly without the callback function and we will learn how to layout the widgets in a much more elegant way.

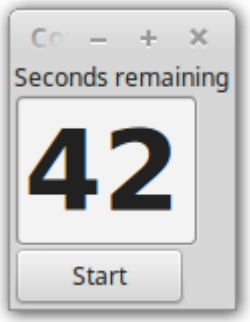
Countdown Revisited

In this chapter, we will modify the layout of the widgets to make it more pleasing to the eye. We'll see that we won't need complicated calculations and we'll see that wxWidgets will handle the resizing of the frame for us. By the way, we haven't talked about that yet, but do try to resize the frame by pulling one its corners to see how, if at all, the layout changes.

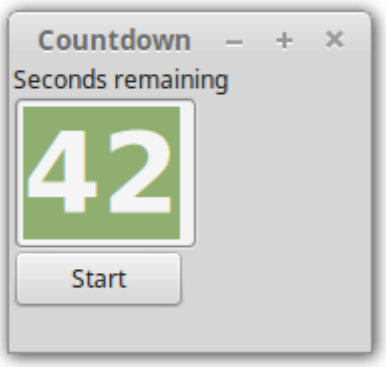
We change the start/0 function in the countdown module as follows:

```
start()  ->
    wx:new(),
    Frame = wxFrame:new(wx:null(), 1, "Countdown"),

    %% build and layout the GUI components
    Label = wxStaticText:new(Frame, ?wxID_ANY, "Seconds remaining"),
    Counter = wxTextCtrl:new(Frame, ?wxID_ANY, [{value, "42"}]),
    Font = wxFont:new(42, ?wxFONTFAMILY_DEFAULT, ?wxFONTSTYLE_NORMAL, ?
wxFONTWEIGHT_BOLD),
    wxTextCtrl:setFont(Counter, Font),

    Button = wxButton:new(Frame, ?wxID_ANY, [{label, "Start"}]),

    MainSizer = wxBoxSizer:new(?wxVERTICAL),
    wxSizer:add(MainSizer, Label),
    wxSizer:add(MainSizer, Counter),
    wxSizer:add(MainSizer, Button),

    wxWindow:setSizer(Frame, MainSizer),
    wxSizer:setSizeHints(MainSizer, Frame),

    wxButton:connect(Button, command_button_clicked, [{callback, fun
handle_click/2}, {userData, #{counter => Counter, env => wx:get_env()}}]),

    wxFrame:show(Frame).
```

And if we compile and run it, we get a frame like:



If we try to resize it, we see something like:



We can hardly see the caption without resizing the frane, and if we do try to resize it, we can't make it any smaller. We have added a label above the counter as its presence will be useful in the explanation of the layout, and not because it is needed. Note that we haven"t specified the position of the button or the size of the counter.

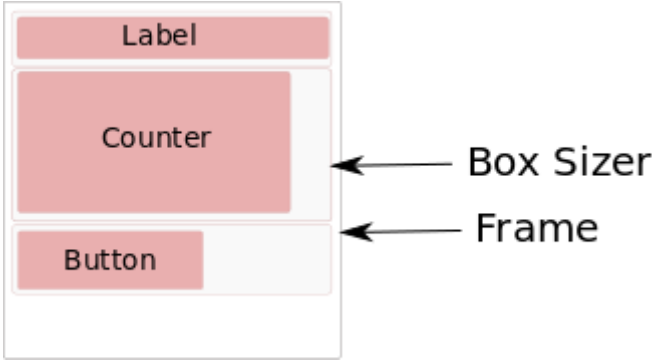
wxWidgets introduces so-called Sizer widgets to help with the layout of all the widgets. Sizers are lightweight widgets, we dare say pseudo-widgets, whose only function is to help with the layout of the contained widgets or sizers. There are several type of sizers, the simplest being the Box Sizer.

The Box Sizer lays out the contained widgets either vertically, one on top of the other, or horizontally, one next to the other. We specify the orientation in the new/1 function with the macros ?wxVERTICAL or ?wxHORIZONTAL.

When we add widgets to the sizer, they are created with some “natural” size which depends on the platform and on default values, including the default font size, etc. Imagine the sizer as a sort of wireframe. As we add widgets, new cells are added to the wireframe to accommodate the added widgets. A vertically-oriented box sizer adds cells to its bottom. A horizontally-oriented box sizer adds cells to its right. The overall size of the box sizer itself is the size of the containing widget. So if the containing widget is resized, the sizer will be resized too and may force the resizing or repositioning of the contained widgets. We shall see how this works.

For a vertical box sizer, all cells of the sizer will have the same width - the width of the containing widget. For the horizontal box sizer, the opposite holds; the height of all the cells is the same as the height of the containing widget.

By default, the height of all the cells in a vertical box sizer is the same as the height of the contained widgets. This may leave out some space at the bottom, which we will shortly see how to fill up. For our countdown GUI, the sizer will be something like the following figure:



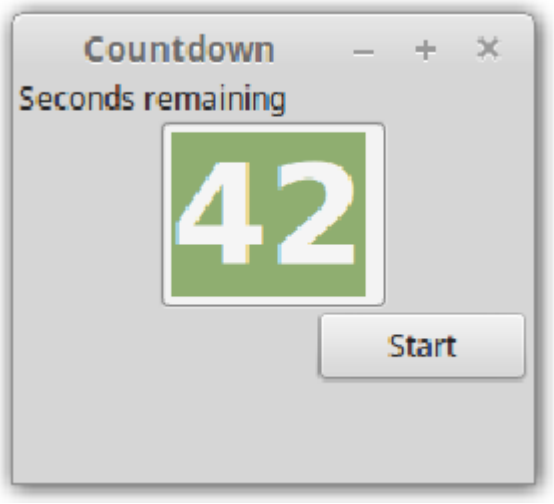
For the horizontally-oriented box sizer, it is the other way around. All cells have the same hieght,

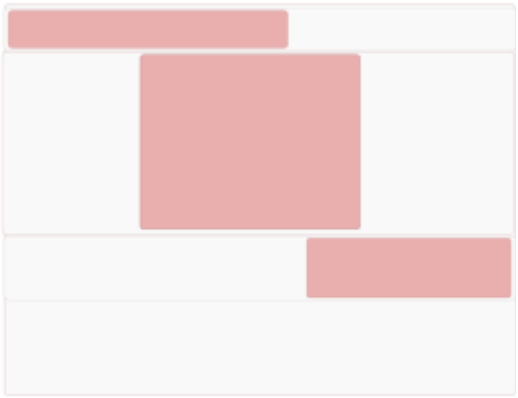
but are as wide as the contained widgets. If the containing widget's width is greater than the sum of all the widths of the contained widgets, some space will be left on the right.

The default alignment of the widgets is to the top-left. That explains why the Button and Counter are where they are. If we change the alignment of the button to right and that of the counter to centre, by specifying it in the add/3 function of the box sizer by means of some flags:

```
wxSizer:add(MainSizer, Counter, [{flag, ?wxALIGN_CENTRE}]),
wxSizer:add(MainSizer, Button, [{flag, ?wxALIGN_RIGHT}]),
```

our frame will change to:



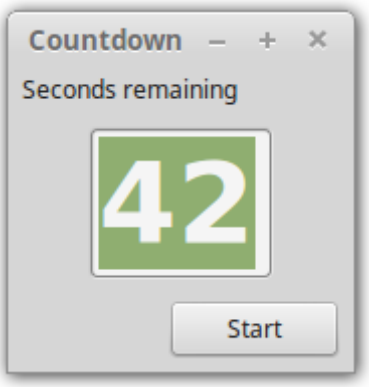


Please see the wxWidgets documentation for the available flag constants.

Our widgets look too packed together and this we can change by forcing some space around them. This space is called the border. We can specify the border on the four sides independently, or we can specify the border for all the sides with the flag ?wxALL. The size of the border is specified by the border option. Let's add a 5 pixel border to all three widgets:

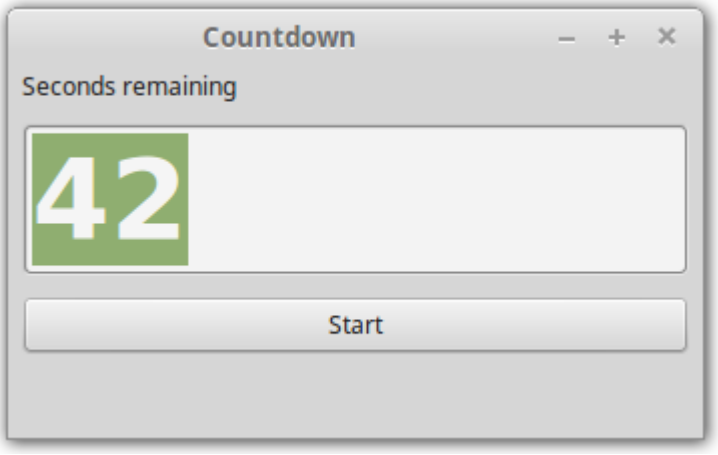
```
wxSizer:add(MainSizer, Label, [{flag, ?wxALL}, {border, 5}]),
wxSizer:add(MainSizer, Counter, [{flag, ?wxALIGN_CENTRE bor ?wxALL}, {border,
5}]),
wxSizer:add(MainSizer, Button, [{flag, ?wxALIGN_RIGHT bor ?wxALL}, {border,
5}]),
```

The result is:



Other than aligning the widgets, wxWidgets also allows them to expand and fill the space. This is done with the flag ?wxEXPAND. Here's what we'll get by expanding all three widgets.

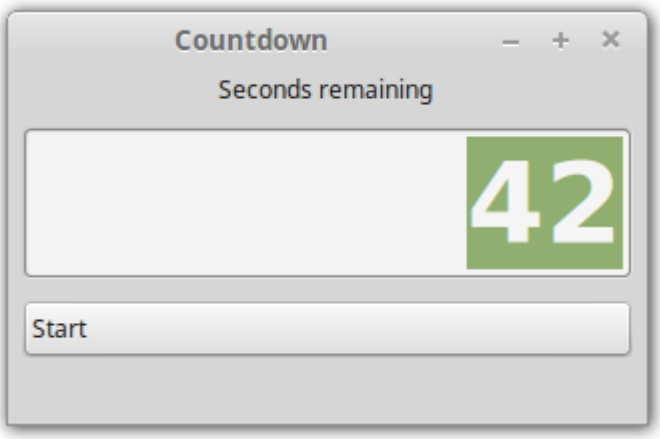
```
wxSizer:add(MainSizer, Label, [{flag, ?wxALL bor ?wxEXPAND}, {border, 5}]),
wxSizer:add(MainSizer, Counter, [{flag, ?wxEXPAND bor ?wxALL}, {border, 5}]),
wxSizer:add(MainSizer, Button, [{flag, ?wxEXPAND bor ?wxALL}, {border, 5}]),
```



Note that although the counter widget expanded to fill the space, the count is still on the left. That is the default alignment of text inside a text control. The same is true of the label. The button's label is aligned to the centre by default. We can change these alignments explicitly, but please note that it is part of the widget's style, not a flag for the sizer!

Here's what we get by aligning the label to the centre, the counter to the right and the button to the left:

```
Counter = wxTextCtrl:new(Frame, ?wxID_ANY, [{value, "42"}, {style, ?
wxTE_RIGHT}]),
Font = wxFont:new(42, ?wxFONTFAMILY_DEFAULT, ?wxFONTSTYLE_NORMAL, ?
wxFONTWEIGHT_BOLD),
wxTextCtrl:setFont(Counter, Font),
Button = wxButton:new(Frame, ?wxID_ANY, [{label, "Start"},{style, ?
wxBU_LEFT}]),
MainSizer = wxBoxSizer:new(?wxVERTICAL),
wxSizer:add(MainSizer, Label, [{flag, ?wxALL bor ?wxALIGN_CENTRE}, {border,
5}]),
wxSizer:add(MainSizer, Counter, [{flag, ?wxEXPAND bor ?wxALL}, {border, 5}]),
wxSizer:add(MainSizer, Button, [{flag, ?wxEXPAND bor ?wxALL}, {border, 5}]),
```

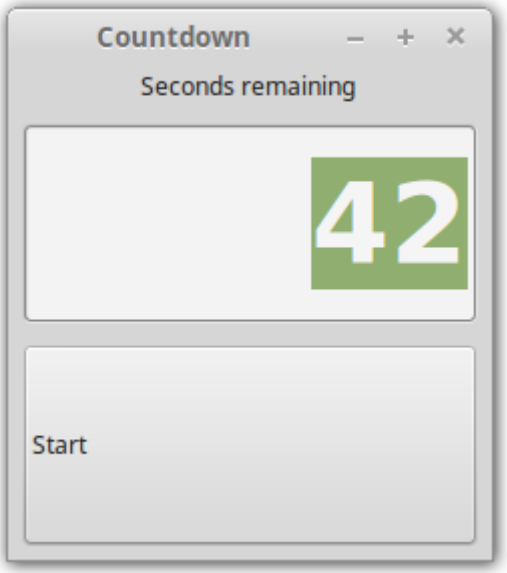


We cheated a bit by aligning the label to the centre, instead of expanding it and giving it a center style.

Now we examine the vertical dimension, and the space left at the bottom if we resize the window to be higher than the sum of the heights of all the contained widgets.

When we add a widget to a box sizer, a proportion value is associated to the cell. The default value is zero, but we can specify any positive integer. What we said above about each cell being as high as the widget itself is actually true only when the associated proportion values are zero. For any cells that have a proportion value greater than zero, the box sizer adds up those numbers and then allocates space proportionally to each of those cells.

Let's assume we set the proportions 0, 1, and 1 for the cells containing the label, the counter and the button respectively. The label will get a space equal to its “natural” height. Then the box sizer sums up the proportions and gets 1+1=2. It then divides its vertical space into two cells with heights in proportion. So the counter gets 1/2 of the space and the button gets 1/2 of the space. If the proportion values had been 2 and 3 for the counter and button, counter would have got 2/5 of the space and button 3/5 of it. Here's what we get with proportions 0, 1, 1:



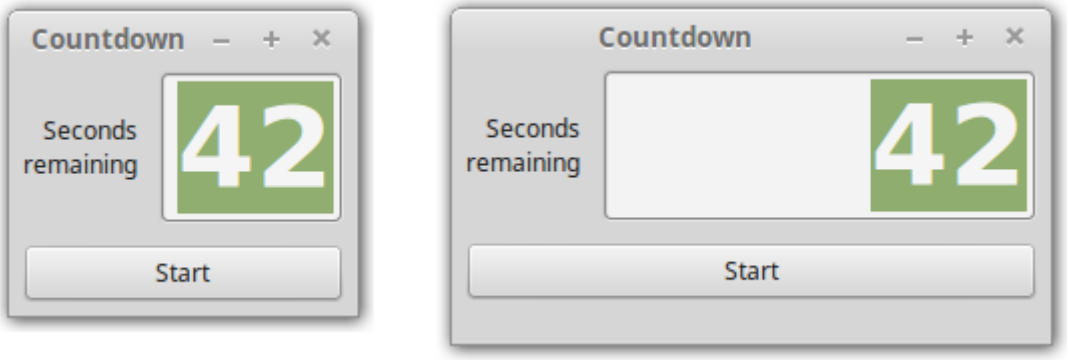
Please note that within that vertical space, the button label and the counter value are centered. These are style values. wxButton does allow the label to be aligned to the top or the bottom. wxTextCtrl, instead, has styles that allow multiline text entry, word wrapping, horizontal scroll bar, etc.

Finally, a word about why we can't make the frame any smaller than the initial size. This is due to the setSizeHints function on the sizer. It instructs the sizer to make itself as small as possible by calling the function fit. However, it also instructs the window to set its minimum size to whatever the natural minimum size of the widgets is. In fact, if you didn't give the size hints, or explicitly set the frame size to some value, you would be able to resize the frame so small, some of the widgets would disappear. We'll have more to say on this later.

Here's what the wxWidgets documentation says about it:

> This method first calls wxSizer::Fit and then SetSizeHints on the window passed to it. This only makes sense when window is actually a wxTopLevelWindow such as a wxFrame or a wxDialog, since SetSizeHints only has any effect in these classes. It does nothing in normal windows or controls.
>
> This method is commonly invoked in the constructor of a toplevel window itself if the toplevel window is resizable.

As an exercise, try to figure out how you can change the layout to get:



The frame on the left is the minimum size. The one on the right is after resizing.

Hints: we use a horizontal box sizer to contain the label and the counter. To get the label to split across two rows, we use the function wrap/2 of module wxStaticText. The wxWidgets documentation of the function states:

> This functions wraps the controls label so that each of its lines becomes at most width pixels wide if possible (the lines are broken at words boundaries so it might not be the case if words are too long). If width is negative, no wrapping is done.

Here's the code, but do try to figure it out yourself before looking at it.

```
Label = wxStaticText:new(Frame, ?wxID_ANY, "Seconds remaining", [{style, ?
wxALIGN_RIGHT}]),
wxStaticText:wrap(Label,100),
Counter = wxTextCtrl:new(Frame, ?wxID_ANY, [{value, "42"}, {style, ?
wxTE_RIGHT}]),
Font = wxFont:new(42, ?wxFONTFAMILY_DEFAULT, ?wxFONTSTYLE_NORMAL, ?
wxFONTWEIGHT_BOLD),
wxTextCtrl:setFont(Counter, Font),
Button = wxButton:new(Frame, ?wxID_ANY, [{label, "Start"}]),

CounterSizer = wxBoxSizer:new(?wxHORIZONTAL),
wxSizer:add(CounterSizer, Label, [{flag, ?wxALL bor ?wxALIGN_CENTRE}, {border,
5}]),
```

```
wxSizer:add(CounterSizer, Counter, [{proportion,1}, {flag, ?wxEXPAND bor ?
wxALL}, {border, 5}]),

MainSizer = wxBoxSizer:new(?wxVERTICAL),
wxSizer:add(MainSizer, CounterSizer, [{flag, ?wxEXPAND}]),
wxSizer:add(MainSizer, Button, [{flag, ?wxEXPAND bor ?wxALL}, {border, 5}]),

wxWindow:setSizer(Frame, MainSizer),
wxSizer:setSizeHints(MainSizer, Frame),
```

The box sizer is the simplest, but the concepts we have learnt will enable you to make sense of the others. There's a Grid Sizer, with all cells having the same width and same height. Then there's a Flexible Grid Sizer, which allows for different heights for the single rows and different widths for the single columns. Finally, there is a Grid Bag Sizer, which is like the Flexible Grid Sizer, but in addition allows you to place widgets in specific locations and allows the widgets to span more than one column or row.

Countdown Interaction Design Revisited

We're done with the aesthetics, but the interaction part, the handling of events, etc is a bit ugly. And this is a pretty simple application. What with more complex stuff?

In this section we will learn how to treat the events differently, without using callback functions.

If you recall the introduction to wx events, we said we have to subscribe to those events by means of the connect/2 or connect/3 functions and we could supply a callback function in that subscription. We also said that if we didn't provide a callback function, the event would be delivered to the process from which the subscription to the event was made.

Let's try it out in the shell:

```
3> B = wxButton:new(F, 1001, [{label, "Click me"}]).
{wx_ref,36,wxButton,[]}
4> wxFrame:show(F).
true
5> wxButton:connect(B, command_button_clicked).
ok
6> flush().
Shell got {wx,1001,
              {wx_ref,36,wxButton,[]},
              [],
              {wxCommand,command_button_clicked,[],0,0}}
ok
7>
```

After subscribing to button click events in line 5, we click the button. To see if the shell got any message, we use flush/0, and indeed, the shell has got the wx event that would have been the first argument to the callback function had we provided one in the subscription.

We also know that the Erlang timer can be used to send the process a message after some time, instead of invoking a function.

So, since the countdown timer is now beginning to look more like a server that does stuff in response to messages it receives, we can use a gen_server. Also, we don't need to get the state of our countdown timer from the label of the button, we can keep that info in the gen_server state.

Here's the module to do it based on the gen_server template in emacs:

```
-module(countdown_server).

-behaviour(gen_server).
-export([start_link/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
       terminate/2, code_change/3]).

-include_lib("wx/include/wx.hrl").

-define(SERVER, ?MODULE).

-record(state, {counter, button, counting_down, tref}).


start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).
```

```
init([]) ->
    wx:new(),
    Frame = wxFrame:new(wx:null(), 1, "Countdown"),

    %% build and layout the GUI components
    Label = wxStaticText:new(Frame, ?wxID_ANY, "Seconds remaining", [{style, ?
wxALIGN_RIGHT}]),
    wxStaticText:wrap(Label,100),
    Counter = wxTextCtrl:new(Frame, ?wxID_ANY, [{value, "42"}, {style, ?
wxTE_RIGHT}]),
    Font = wxFont:new(42, ?wxFONTFAMILY_DEFAULT, ?wxFONTSTYLE_NORMAL, ?
wxFONTWEIGHT_BOLD),
    wxTextCtrl:setFont(Counter, Font),
    Button = wxButton:new(Frame, ?wxID_ANY, [{label, "Start"}]),

    CounterSizer = wxBoxSizer:new(?wxHORIZONTAL),
    wxSizer:add(CounterSizer, Label, [{flag, ?wxALL bor ?wxALIGN_CENTRE},
{border, 5}]),

    wxSizer:add(CounterSizer, Counter, [{proportion,1}, {flag, ?wxEXPAND bor ?
wxALL}, {border, 5}]),

    MainSizer = wxBoxSizer:new(?wxVERTICAL),
    wxSizer:add(MainSizer, CounterSizer, [{flag, ?wxEXPAND}]),
    wxSizer:add(MainSizer, Button, [{flag, ?wxEXPAND bor ?wxALL}, {border,
5}]),

    wxWindow:setSizer(Frame, MainSizer),
    wxSizer:setSizeHints(MainSizer, Frame),
    wxWindow:setMinSize(Frame, wxWindow:getSize(Frame)),

    wxButton:connect(Button, command_button_clicked),

    wxFrame:show(Frame),
    {ok, #state{counter = Counter, button = Button, counting_down = false}}.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(#wx{obj = Button, event = #wxCommand{type =
command_button_clicked}}, #state{counter = Counter, counting_down = false} =
State) ->
    case list_to_integer(wxTextCtrl:getValue(Counter)) of
      0 ->
          {noreply, State};
      _ ->
          wxTextCtrl:setEditable(Counter, false),
          wxButton:setLabel(Button, "Stop"),
          TRef = erlang:send_after(1000, self(), update_gui),
          {noreply, State#state{tref = TRef, counting_down = true}}
    end;

handle_info(#wx{obj = Button, event = #wxCommand{type =
command_button_clicked}}, #state{counter = Counter, counting_down = true, tref
= TRef} = State) ->
    erlang:cancel_timer(TRef),
    wxTextCtrl:setEditable(Counter, true),
    wxButton:setLabel(Button, "Start"),
```

```
    {noreply, State#state{tref = undefined, counting_down = false}};

handle_info(update_gui, #state{button = Button, counter = Counter,
counting_down = true} = State) ->
    Value = wxTextCtrl:getValue(Counter),
    case list_to_integer(Value) of
      1 ->
          wxTextCtrl:setValue(Counter, "0"),
          wxTextCtrl:setEditable(Counter, true),
          wxButton:setLabel(Button, "Start"),
          {noreply, State#state{counting_down = false}};
      N ->
          wxTextCtrl:setValue(Counter, integer_to_list(N-1)),
          TRef = erlang:send_after(1000, self(), update_gui),
          {noreply, State#state{tref = TRef}}
    end.

terminate(_Reason, _State) ->
    wx:destroy(),
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

There is something new in this:

```
wxWindow:setMinSize(Frame, wxWindow:getSize(Frame)),
```

When we call setSizeHints/2, wxWidgets will not only lay out the widgets, it will also set the minimum size of the frame. However, the minimum size that it calculates uses the default sizes of the widgets involved. So it may be preferable to set the minimum size to what setSizeHints calculated as the current best size.

Also, instead of handling the Erlang timer when the application is not counting down, we simply cancel it if the countdown is stopped.

One last thing. If we close the countdown window, it will close alright, but the gen_server will keep running. If we want the gen_server to exit as well, which normally we would, we need to subscribe to the window closing event, close_window, and handle it.

We can subscribe to the event thus:

```
wxFrame:connect(Frame, close_window),
```

And then handle the resulting wxClose event, by requesting the gen_server to stop:

```
handle_info(#wx{event=#wxClose{}}, State) ->
    {stop, normal, State}.
```

Chess clock

The chess clock is a countdown timer used by chess players. Players decide on a certain amount of time for each player. Here's a digital version (courtesy http://chessgeeks.com)

Two clocks that count down are used, and are set to the same value initially. This is the time each player has for his moves. As soon as a player makes his move, he hits the buton on top of the clock. This stops his clock, but starts the clock of the opponent. And vice versa. If the players are unable to win during the maximum time, the one who runs out of time first, looses.

We will reuse the countdown timer we developed in the previous sections to make a chess clock.

The use case is slightly different. The two clocks have to be set to the same value to start with and once the game starts, the white player's clock has to start counting down. Only the button of the player who has to move a piece will count down. As soon as he moves, he hits his button, stopping his own count, but starting the countdown of the opponent. At this point, the opponent's clock starts counting down. There is no stopping the game or changing the value of the counter. Once the counter of any of the players goes to zero, the game ends and the player whose count is above zero, wins.

The two clocks of the two players are just the same from a GUI point of view, and although the two will have to interact with each other at some point, the handling of the countdown timer and detecting the button being clicked is still the same in both. So we will factor this part out into a separate process and have the main process start two instances of it. Let's call the main process, arbiter, as it is where we will create the frame and as we will let it play the role of the arbiter, who decides what value to start the game with, and when to start it. We'll call the other processes player.

To let the player processes build the counter and button, we will pass the process a reference to the Frame and to the wx environment. The player process will create a panel, a top level window in wx, and place the controls on it. Later, the arbiter process can fetch the panel from the player process and add it to its sizer.

Here's how the construction of the Frame will look like in the arbiter process.

```
Env = wx:get_env(),
Frame = wxFrame:new(wx:null(), 1, "Countdown"),
{ok, _} = player:start_link(player1, Env, Frame, ?SERVER),
{ok, _} = player:start_link(player2, Env, Frame, ?SERVER),
Player1 = gen_server:call(player1, get_panel),
Player2 = gen_server:call(player2, get_panel),
```

```
MainSizer = wxBoxSizer:new(?wxHORIZONTAL),
wxSizer:add(MainSizer, Player1, [{proportion, 1}, {flag, ?wxALL}, {border,
5}]),
wxSizer:add(MainSizer, Player2, [{proportion, 1}, {flag, ?wxALL}, {border,
5}]),
```

And here's the construction of the player:

```
start_link(Name, Env, Frame, Arbiter) ->
    gen_server:start_link({local, Name}, ?MODULE, [Name, Env, Frame, Arbiter],
[]).

init([Name, Env, Frame, Arbiter]) ->
    wx:set_env(Env),

    Panel = wxPanel:new(Frame),

    Label = wxStaticText:new(Panel, ?wxID_ANY, "Seconds remaining", [{style, ?
wxALIGN_RIGHT}]),
    wxStaticText:wrap(Label,100),

    Counter = wxTextCtrl:new(Panel, ?wxID_ANY, [{value, "42"}, {style, ?
wxTE_RIGHT}]),
    Font = wxFont:new(42, ?wxFONTFAMILY_DEFAULT, ?wxFONTSTYLE_NORMAL, ?
wxFONTWEIGHT_BOLD),
    wxTextCtrl:setFont(Counter, Font),
    wxTextCtrl:setEditable(Counter, false),

    Button = wxButton:new(Panel, ?wxID_ANY, [{label, "Moved"}]),
    wxButton:disable(Button),

    CounterSizer = wxBoxSizer:new(?wxHORIZONTAL),
    wxSizer:add(CounterSizer, Label, [{flag, ?wxALL bor ?wxALIGN_CENTRE},
{border, 5}]),
    wxSizer:add(CounterSizer, Counter, [{proportion,1}, {flag, ?wxEXPAND bor ?
wxALL}, {border, 5}]),

    MainSizer = wxBoxSizer:new(?wxVERTICAL),
    wxSizer:add(MainSizer, CounterSizer, [{flag, ?wxEXPAND}]),
    wxSizer:add(MainSizer, Button, [{flag, ?wxEXPAND bor ?wxALL}, {border,
5}]),

    wxWindow:setSizer(Panel, MainSizer),
    wxSizer:setSizeHints(MainSizer, Panel),
    wxWindow:setMinSize(Panel, wxWindow:getSize(Panel)),

    wxButton:connect(Button, command_button_clicked),

    {ok, #state{panel = Panel,
            counter = Counter,
            button = Button,
            whoami = Name,
            arbiter = Arbiter}}.
```

We keep a reference to the panel in the state. That way we can provide it to the arbiter when it asks for it:

```
handle_call(get_panel, _From, #state{panel = Panel} = State) ->
    {reply, Panel, State};
```

We also keep the names of the player and the arbiter processes in the state. We'll soon see why.

As preparation to starting the game, we need to set the counters to a certain value. We do this by

sending the arbiter process the message {reset, N}, where N is the value we want the counters set to.

```
handle_info({reset, N}, State) ->
    player1 ! {reset, N},
    player2 ! {reset, N},
    {noreply, State};
```

The players, upon receiving the message, will set the counter to that value and disable their buttons.

```
handle_info({reset,N}, #state{counter = Counter, button = Button} = State) ->
    wxTextCtrl:setValue(Counter, integer_to_list(N)),
    wxButton:disable(Button),
    {noreply, State};
```

To actually start the game, one of the players, the white player, will have to move a piece. So we simply send the message move to the white player:

```
handle_info(move, #state{button = Button} = State) ->
    wxButton:enable(Button),
    TRef = erlang:send_after(1000, self(), update_gui),
    {noreply, State#state{tref = TRef}};
```

When the player has moved his piece, he hits the button. This causes his button to be disabled and the countdown to be stopped. It also informs the arbiter he has moved his piece:

```
handle_info(#wx{obj = Button, event = #wxCommand{type =
command_button_clicked}}, #state{tref = TRef, whoami = Name, arbiter = Arbiter}
= State) ->
    erlang:cancel_timer(TRef),
    wxButton:disable(Button),
    Arbiter ! {moved, Name},
    {noreply, State#state{tref = undefined}};
```

The arbiter, upon receiving the moved notification, tells the opponent to move his piece:

```
handle_info({moved, player1}, State) ->
    player2 ! move,
    {noreply, State};
handle_info({moved, player2}, State) ->
    player1 ! move,
    {noreply, State};
```

In updating the GUI, the player will normally decrement the counter by one and program an update of the GUI again after one second. However, when the count reaches zero, he knows he has lost. So, he will disable the button and notify the arbiter he has lost.

```
handle_info(update_gui, #state{button = Button, counter = Counter, whoami =
Name, arbiter = Arbiter} = State) ->
    Value = wxTextCtrl:getValue(Counter),
    case list_to_integer(Value) of
      1 ->
          wxTextCtrl:setValue(Counter, "0"),
          wxButton:disable(Button),
          Arbiter ! {ilose, Name},
          {noreply, State};
      N ->
          wxTextCtrl:setValue(Counter, integer_to_list(N-1)),
          TRef = erlang:send_after(1000, self(), update_gui),
          {noreply, State#state{tref = TRef}}
    end.
```

The arbiter will then inform the opponent he has won:

```
handle_info({ilose, player1}, State) ->
    player2 ! youwin,
    {noreply, State};
handle_info({ilose, player2}, State) ->
    player1 ! youwin,
    {noreply, State};
```

The winner may display his victory anyhow he pleases. eg by displaying some text.

```
handle_info(youwin, #state{counter = Counter} = State) ->
    wxTextCtrl:setValue(Counter, "win"),
    {noreply, State};
```

One last thing we need to take care of is when we close the arbiter window. We will need to terminate the player processes as well.

```
terminate(Reason, _State) ->
    sys:terminate(player1, Reason),
    sys:terminate(player2, Reason),
    wx:destroy(),
```

This is the complete arbiter module:

```
-module(arbiter).

-behaviour(gen_server).
-export([start_link/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
       terminate/2, code_change/3]).

-include_lib("wx/include/wx.hrl").

-define(SERVER, ?MODULE).

-record(state, {}).


start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

init([]) ->
    wx:new(),
    Env = wx:get_env(),

    %% build and layout the GUI components
    Frame = wxFrame:new(wx:null(), 1, "Countdown"),
    {ok, _} = player:start_link(player1, Env, Frame, ?SERVER),
    {ok, _} = player:start_link(player2, Env, Frame, ?SERVER),
    Player1 = gen_server:call(player1, get_panel),
    Player2 = gen_server:call(player2, get_panel),

    MainSizer = wxBoxSizer:new(?wxHORIZONTAL),
    wxSizer:add(MainSizer, Player1, [{proportion, 1}, {flag, ?wxALL}, {border,
5}]),
    wxSizer:add(MainSizer, Player2, [{proportion, 1}, {flag, ?wxALL}, {border,
5}]),

    wxWindow:setSizer(Frame, MainSizer),
    wxSizer:setSizeHints(MainSizer, Frame),
    wxWindow:setMinSize(Frame, wxWindow:getSize(Frame)),

    wxFrame:connect(Frame, close_window),
```

```
    wxFrame:show(Frame),
    {ok, #state{}}.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info({reset, N}, State) ->
    player1 ! {reset, N},
    player2 ! {reset, N},
    {noreply, State};

handle_info({moved, player1}, State) ->
    player2 ! move,
    {noreply, State};
handle_info({moved, player2}, State) ->
    player1 ! move,
    {noreply, State};

handle_info({ilose, player1}, State) ->
    player2 ! youwin,
    {noreply, State};
handle_info({ilose, player2}, State) ->
    player1 ! youwin,
    {noreply, State};

handle_info(#wx{event = #wxClose{}}, State) ->
    {stop, normal, State};

handle_info(Msg, State) ->
    io:format("frame got unexpected message ~p~n", [Msg]),
    {noreply, State}.

terminate(Reason, _State) ->
    sys:terminate(player1, Reason),
    sys:terminate(player2, Reason),
    wx:destroy(),
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

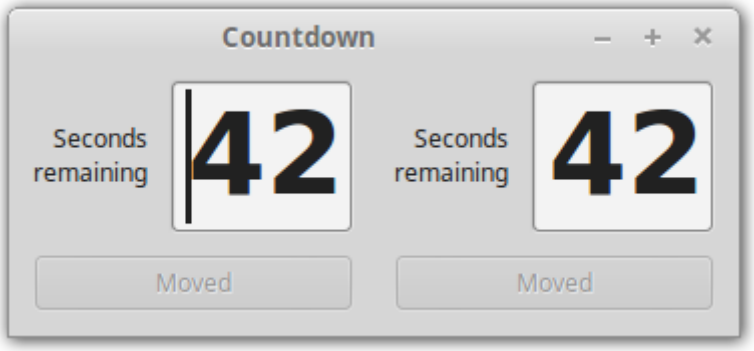
And this, the complete player module:

```
-module(player).

-behaviour(gen_server).
-export([start_link/4]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
       terminate/2, code_change/3]).

-include_lib("wx/include/wx.hrl").

-record(state, {panel, counter, button, tref, whoami, arbiter}).


start_link(Name, Env, Frame, Arbiter) ->
    gen_server:start_link({local, Name}, ?MODULE, [Name, Env, Frame, Arbiter],
[]).
```

```
init([Name, Env, Frame, Arbiter]) ->
    wx:set_env(Env),
    Panel = wxPanel:new(Frame),

    %% build and layout the GUI components
    Label = wxStaticText:new(Panel, ?wxID_ANY, "Seconds remaining", [{style, ?
wxALIGN_RIGHT}]),
    wxStaticText:wrap(Label,100),
    Counter = wxTextCtrl:new(Panel, ?wxID_ANY, [{value, "42"}, {style, ?
wxTE_RIGHT}]),
    Font = wxFont:new(42, ?wxFONTFAMILY_DEFAULT, ?wxFONTSTYLE_NORMAL, ?
wxFONTWEIGHT_BOLD),
    wxTextCtrl:setFont(Counter, Font),
    wxTextCtrl:setEditable(Counter, false),

    Button = wxButton:new(Panel, ?wxID_ANY, [{label, "Moved"}]),
    wxButton:disable(Button),

    CounterSizer = wxBoxSizer:new(?wxHORIZONTAL),
    wxSizer:add(CounterSizer, Label, [{flag, ?wxALL bor ?wxALIGN_CENTRE},
{border, 5}]),
    wxSizer:add(CounterSizer, Counter, [{proportion,1}, {flag, ?wxEXPAND bor ?
wxALL}, {border, 5}]),

    MainSizer = wxBoxSizer:new(?wxVERTICAL),
    wxSizer:add(MainSizer, CounterSizer, [{flag, ?wxEXPAND}]),
    wxSizer:add(MainSizer, Button, [{flag, ?wxEXPAND bor ?wxALL}, {border,
5}]),

    wxWindow:setSizer(Panel, MainSizer),
    wxSizer:setSizeHints(MainSizer, Panel),
    wxWindow:setMinSize(Panel, wxWindow:getSize(Panel)),

    wxButton:connect(Button, command_button_clicked),

    {ok, #state{panel = Panel,
            counter = Counter,
            button = Button,
            whoami = Name,
            arbiter = Arbiter}}.

handle_call(get_panel, _From, #state{panel = Panel} = State) ->
    {reply, Panel, State};
handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info({reset,N}, #state{counter = Counter, button = Button} = State) ->
    wxTextCtrl:setValue(Counter, integer_to_list(N)),
    wxButton:disable(Button),
    {noreply, State};

handle_info(youwin, #state{counter = Counter} = State) ->
    wxTextCtrl:setValue(Counter, "win"),
    {noreply, State};

handle_info(#wx{obj = Button, event = #wxCommand{type =
command_button_clicked}}, #state{tref = TRef, whoami = Name, arbiter = Arbiter}
= State) ->
    erlang:cancel_timer(TRef),
```

```
    wxButton:disable(Button),
    Arbiter ! {moved, Name},
    {noreply, State#state{tref = undefined}};

handle_info(move, #state{button = Button} = State) ->
    wxButton:enable(Button),
    TRef = erlang:send_after(1000, self(), update_gui),
    {noreply, State#state{tref = TRef}};

handle_info(update_gui, #state{button = Button, counter = Counter, whoami =
Name, arbiter = Arbiter} = State) ->
    Value = wxTextCtrl:getValue(Counter),
    case list_to_integer(Value) of
      1 ->
          wxTextCtrl:setValue(Counter, "0"),
          wxButton:disable(Button),
          Arbiter ! {ilose, Name},
          {noreply, State};
      N ->
          wxTextCtrl:setValue(Counter, integer_to_list(N-1)),
          TRef = erlang:send_after(1000, self(), update_gui),
          {noreply, State#state{tref = TRef}}
    end.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

Please compile and run it to see how it works.



wx_object

We have come a long way to finally talk of wx_object. I find it is perhaps the most interesting abstraction in wxErlang. It is practically the gen_server implementation of the previous section brushed up to make life easier for the wxErlang programmer. In talking about it, let's say the arbiter process is like a master wx_object and the player processes are like wx_object slaves.

Here's the things it takes care of for us:

- It passes the wx environment from the master to the slaves
- It fetches the panel, or whatever else the slave may have constructed. This is done by requiring the slave init/1 function to return {WxObject, State}, instead of the gen_server {ok, State} tuple. In this way, the wx_object:start_link function can return a widget, instead of the {ok, Pid} returned by the gen_server start functions
- It takes care of terminating the slaves when the master terminates
- It requires a different callback, handle_event, for wx events. In other words wx events are not handled by handle_info. This requirement is enforced by the wx_object behaviour.

So, making these changes to our arbiter and player modules, we get the following wo_arbiter and wo_player modules:

```
-module(wo_arbiter).

-include_lib("wx/include/wx.hrl").
-behaviour(wx_object).

-export([start_link/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
      handle_event/2,
      terminate/2, code_change/3]).

-define(SERVER, ?MODULE).

-record(state, {}).

start_link() ->
    wx_object:start_link({local, ?SERVER}, ?MODULE, [], []).

init([]) ->
    wx:new(),
    %% Env = wx:get_env(),

    %% build and layout the GUI components
    Frame = wxFrame:new(wx:null(), 1, "Countdown"),
    Player1 = wo_player:start_link(player1, Frame, ?SERVER),
    Player2 = wo_player:start_link(player2, Frame, ?SERVER),
    %% Player1 = gen_server:call(player1, get_panel),
    %% Player2 = gen_server:call(player2, get_panel),

    MainSizer = wxBoxSizer:new(?wxHORIZONTAL),
    wxSizer:add(MainSizer, Player1, [{proportion, 1}, {flag, ?wxALL}, {border,
5}]),
    wxSizer:add(MainSizer, Player2, [{proportion, 1}, {flag, ?wxALL}, {border,
5}]),
```

```
    wxWindow:setSizer(Frame, MainSizer),
    wxSizer:setSizeHints(MainSizer, Frame),
    wxWindow:setMinSize(Frame, wxWindow:getSize(Frame)),

    wxFrame:connect(Frame, close_window),

    wxFrame:show(Frame),
    {Frame, #state{}}.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info({reset, N}, State) ->
    player1 ! {reset, N},
    player2 ! {reset, N},
    {noreply, State};

handle_info({moved, player1}, State) ->
    player2 ! move,
    {noreply, State};
handle_info({moved, player2}, State) ->
    player1 ! move,
    {noreply, State};

handle_info({ilose, player1}, State) ->
    player2 ! youwin,
    {noreply, State};
handle_info({ilose, player2}, State) ->
    player1 ! youwin,
    {noreply, State};

handle_info(Msg, State) ->
    io:format("frame got unexpected message ~p~n", [Msg]),
    {noreply, State}.

handle_event(#wx{event = #wxClose{}}, State) ->
    {stop, normal, State}.


terminate(_Reason, _State) ->
    %% sys:terminate(player1, Reason),
    %% sys:terminate(player2, Reason),
    wx:destroy(),
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

```
-module(wo_player).

-behaviour(wx_object).
-export([start_link/3]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
       handle_event/2,
       terminate/2, code_change/3]).

-include_lib("wx/include/wx.hrl").

-record(state, {panel, counter, button, tref, whoami, arbiter}).
```

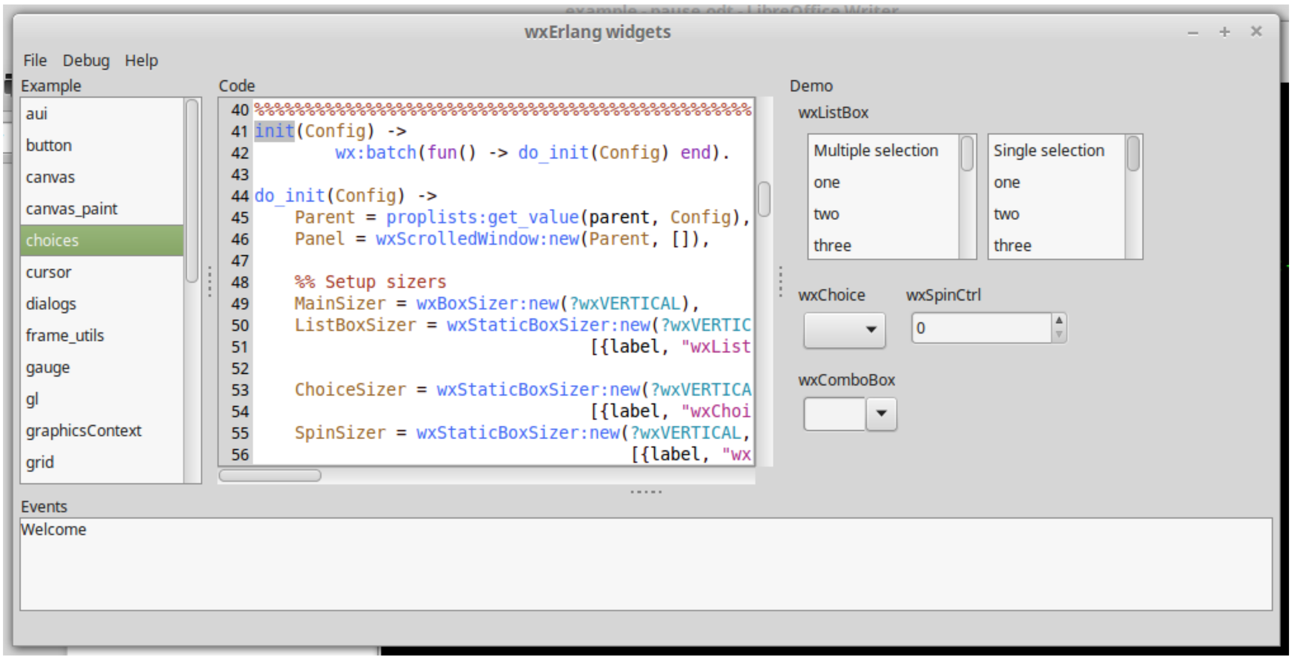
```
start_link(Name, Frame, Arbiter) ->
    wx_object:start_link({local, Name}, ?MODULE, [Name, Frame, Arbiter], []).

init([Name, Frame, Arbiter]) ->
    %% wx:set_env(Env),
    Panel = wxPanel:new(Frame),

    %% build and layout the GUI components
    Label = wxStaticText:new(Panel, ?wxID_ANY, "Seconds remaining", [{style, ?
wxALIGN_RIGHT}]),
    wxStaticText:wrap(Label,100),
    Counter = wxTextCtrl:new(Panel, ?wxID_ANY, [{value, "42"}, {style, ?
wxTE_RIGHT}]),
    Font = wxFont:new(42, ?wxFONTFAMILY_DEFAULT, ?wxFONTSTYLE_NORMAL, ?
wxFONTWEIGHT_BOLD),
    wxTextCtrl:setFont(Counter, Font),
    wxTextCtrl:setEditable(Counter, false),

    Button = wxButton:new(Panel, ?wxID_ANY, [{label, "Moved"}]),
    wxButton:disable(Button),

    CounterSizer = wxBoxSizer:new(?wxHORIZONTAL),
    wxSizer:add(CounterSizer, Label, [{flag, ?wxALL bor ?wxALIGN_CENTRE},
{border, 5}]),
    wxSizer:add(CounterSizer, Counter, [{proportion,1}, {flag, ?wxEXPAND bor ?
wxALL}, {border, 5}]),

    MainSizer = wxBoxSizer:new(?wxVERTICAL),
    wxSizer:add(MainSizer, CounterSizer, [{flag, ?wxEXPAND}]),
    wxSizer:add(MainSizer, Button, [{flag, ?wxEXPAND bor ?wxALL}, {border,
5}]),

    wxWindow:setSizer(Panel, MainSizer),
    wxSizer:setSizeHints(MainSizer, Panel),
    wxWindow:setMinSize(Panel, wxWindow:getSize(Panel)),

    wxButton:connect(Button, command_button_clicked),

    {Panel, #state{panel = Panel,
           counter = Counter,
           button = Button,
           whoami = Name,
           arbiter = Arbiter}}.

handle_call(get_panel, _From, #state{panel = Panel} = State) ->
    {reply, Panel, State};
handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info({reset,N}, #state{counter = Counter, button = Button} = State) ->
    wxTextCtrl:setValue(Counter, integer_to_list(N)),
    wxButton:disable(Button),
    {noreply, State};

handle_info(youwin, #state{counter = Counter} = State) ->
    wxTextCtrl:setValue(Counter, "win"),
    {noreply, State};
```

```
handle_info(move, #state{button = Button} = State) ->
    wxButton:enable(Button),
    TRef = erlang:send_after(1000, self(), update_gui),
    {noreply, State#state{tref = TRef}};

handle_info(update_gui, #state{button = Button, counter = Counter, whoami =
Name, arbiter = Arbiter} = State) ->
    Value = wxTextCtrl:getValue(Counter),
    case list_to_integer(Value) of
      1 ->
          wxTextCtrl:setValue(Counter, "0"),
          wxButton:disable(Button),
          Arbiter ! {ilose, Name},
          {noreply, State};
      N ->
          wxTextCtrl:setValue(Counter, integer_to_list(N-1)),
          TRef = erlang:send_after(1000, self(), update_gui),
          {noreply, State#state{tref = TRef}}
    end.

handle_event(#wx{obj = Button, event = #wxCommand{type =
command_button_clicked}}, #state{tref = TRef, whoami = Name, arbiter = Arbiter}
= State) ->
    erlang:cancel_timer(TRef),
    wxButton:disable(Button),
    Arbiter ! {moved, Name},
    {noreply, State#state{tref = undefined}}.


terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

Please do compile and run it. You will see it is exactly the same as the version with gen_server. Only that this is cleaner.

Once you have understood what wx_object does, you can understand what the wx demo is doing:

```
2> wx:demo().
ok
```



You select the type of control in the list in the left panel and it shows some examples in the right panel. This panel is actually a wx_object and the middle panel shows the code to create it.

This is a great demo as it readily shows the practical way of doing wxErlang. Once you understand it, you will hardly need this booklet anymore. In the frame_utils example you will find how to create other windows, how to create menus, menu bars and toolbars and how to set up a status bar in a frame. For advanced GUI work, wxErlang has support for GL and AUI (https://wiki.wxwidgets.org/WxAUI).

The code for demo/0 in the Erlang distribution's wx library is also instructive as it makes good use of the wx_object abstraction.

Have fun!