Unit: V.
Graphical User Interfaces

Herramientas Avanzadas para el Desarrollo de Aplicaciones

Languages and Computer systems department
University of Alicante

Content

Background (I)

- In this unit we are going to learn how to provide our applications with a *Graphical User Interface* .
- We are going to explain how this is supported by the concepts of *event driven programming* and *deferred code execution* that we have already seen.
- We will briefly introduce the MVC architecture because in order to write our application code we will use it.
- The graphical interface of our desktop applications will use the Gtk+ library.

Background (II)

- We are going to learn how to graphically create the application interface with a program oriented to the design of graphical user interfaces. This program is called Glade .
- The graphical interfaces created using `glade` will be dynamically *loaded* at execution time and shown to the user so he can interact with them.
- Glade generates XML files which contain the description of the designed user interface. These files can be read/loaded using several programming languages: `C`, `C++`, `C#`, `Vala`, `Java`, etc...
- We will use the Vala programming language, used in this subject for the 'desktop' block.

Background (III)

- Vala uses some technologies that are part of the fundamentals of `Gtk+`.
- These are GLib and Gobject .
- This makes possible that is very easy to build applications using Vala with a graphical interface based on `Gtk+`.
- We are going to see a video where in less than five minutes we can see how to create a simple application with a graphical user interface[1].

[1] In this case without using Glade.

MVC (I)

- MVC comes up together with Smalltalk in the 70's.
- It can be applied to the development of any application independently of the programming language we use.
- We do not need an object programming languages to use it, but this methodology makes it easier.

MVC (II)

- The key idea of MVC consists on dividing the code of an application in 3 layers:
  1. **Model**
  2. **View**
  3. **Controller**
- Each of these layers can be substituted in any moment without affecting the others, i.e., having different views for the same model.
- This division of the code guarantees a better potability and adaptation to the user requirements.

MVC: Model. –Application Layer–

- This is the 'software' representation of the problem to solve, its data, functions, etc. . . – persons, cars, entries. . . –
- It provides the needed methods for fulfilling the following tasks:
  - The data of the model can be consulted ($\approx$ *getters*).
  - The data of the model can be modified ($\approx$ *setters*).
- The models do not communicate with the views[2], in this way we get a greater independence among the code of each layer.
- A model can have associated several views.

[2] However in some moment can be interesting.

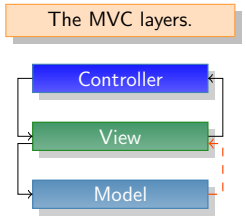
MVC: View. –Presentation Layer–

- It is used to show to the user the 'data' of the model that interest him in any case –the name of a person, the speed of a car, the value of an entry...–
- A view does not have to be only in graphical mode, it can be in text mode...
- Views communicate with the models in a bidirectional way –they request information and can modify information–
- In the original MVC architecture the views can be 'nested' having a so-called main view –*top-view*– composed of sub-views. In this subject we will use simple views, not composed ones.

MVC: Controller. –Interaction Layer–

- It contains the code that behaves as interface among the input devices –keyboard, mouse, etc...– and the View and Model layers.
- This is the code which allows the user interacting with the Views.
- Normally we will not have to write code related with this layer because the code that will go here is the one provided by the graphical library used (Gtk+ in our case).

MVC: Interaction Diagram among layers



The connection between the model and the associated views does not have sense if the data of the model is going to be internally modified by some operation and we want that the views associated to this model are updated.

Gtk+ (I)

- Gtk+ is developed as a free toolkit for the application gimp of image processing. Today is one of the basis of the gnome desktop.
- It is distributed with license LGPL.
- We have many documentation for Gtk+ and for using it iin Vala in electronic format that we can check online.
- It also has a builder for the graphical user interface of the application: glade .

Gtk+ (II)

- Gtk+ is systematically updated twice a year, today we can find the versions 2.$x.y$ (in maintenance mode) and, the currently active, 3.$x.y$. In the EPS labs we have installed Gtk+ 3.$x.y$.
- As we will see later, this affects in the options that we have to specify to the Vala compiler: `--pkg gtk+-2.0` o `--pkg gtk+-3.0`.
- What in a general way we call Gtk+ is composed by a set of libraries: Glib, GdkPixbuf, Gdk, Gtk, Atk y Pango.
- The inner structure of Gtk+ is a hierarchy of classes composed by several trees (different roots) with simple inheritance.
- These trees represent each of the libraries that we have mentioned before (glib, gdk, gtk, etc...).

Gtk+ + Vala + signal/handler (I)

- The use of Gtk+ in Vala is based on what we have seen in the previous units: events/signals and handlers/callbacks.
- The elements of the user interface (widgets, controls)that Gtk+ provides define a set of signals that can be triggered.
- We have to connect them with the methods or functions of our code that behave as handler or callback, i.e. checking the documentation of the Button class we find a section dedicated to signals:

```
1   public virtual signal void activate ()

3   The activate signal on GtkButton is an action signal and
    emitting it causes the button to animate press then
5   release....

7   public virtual signal void clicked ()

9   Emitted when the button has been activated (pressed and
    released).
11
    ...
```


Gtk+ + Vala + signal/handler (II)

Let's see a complete example:

```
    // File: gtk-hello.vala
2   using Gtk;

4   int main (string[] args) {
      Gtk.init (ref args);
6
      var window = new Window (); //Gtk.Window (using Gtk)
8     window.title = "First GTK+ Program";
      window.border_width = 10;
10    window.window_position = WindowPosition.CENTER;
      window.set_default_size (350, 70);
12    window.destroy.connect (Gtk.main_quit);

14    var button = new Button.with_label ("Click me!");
      button.clicked.connect (() => {button.label = "Thank you";});
16
      window.add (button);
18    window.show_all ();

20    Gtk.main ();
      return 0;
22  }
```

It is compiled in this way: `valac --pkg gtk+-3.0 gtk-hello.vala`.

Gtk+ + Vala + signal/handler (III)

In the previous code:

- using Gtk: is equivalent to a `import` in Java or `#include` in C or C++ to have access to the declarations/definitions of Gtk+.
- Gtk.init (ref args): Initiates the Gtk+ library. It is needed to do this always in the beginning of the main program.
- var window = new Window (): Creates an object of the `Gtk.Window` class, this means, a window over which we can add other elements of the user interface.
- var button = new Button.with_label("Click me!"): Creates an object of the `Gtk.Button` class.

Gtk+ + Vala + signal/handler (IV)

- button.clicked.connect(() => {button.label = "Thank you";}): The `Button`class has the `clicked` signal, here we connect it to a handler. In this case it is a λ function.
- window.add (button): We add the created button to the window.
- window.show_all(): The window makes visible all the widget that it contains.
- Gtk.main(): It is the event waiting loop, we only go out from it to finalize the application.

Gtk+ Widgets (I)

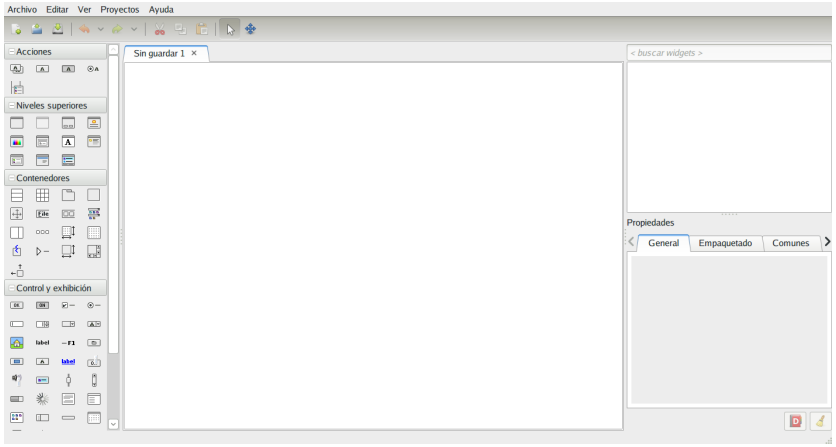
- Gtk+ offers us an important collection of predefined widgets organized as a hierarchy of classes with simple inheritance. We can consult it here
- The base class of any *element of the interface* is the class GtkWidget .
- In Vala the `Gtk` prefix of any identifier, i.e. 'GtkWidget' is interpreted as a *namespace*, so the identifier in Vala would be: 'Gtk.Widget'.
- We have equivalent documentation for the adaptation of Gtk+ to Vala here .

Gtk+ Widgets (II)

- In Gtk+ a widget normally can only contain another widget.
- In order to create functional user interfaces we need to solve this limitation.
- There exist a special type of widgets that are the *containers*, specifically the classes deriving from Gtk.Box . You can find more information from its derived classes here and Gtk.HBox and Gtk.VBox .
- It is also recommended that you know the *container in table form*: Gtk.Table . You have more information about it in the Gtk documentation.
- Visually they do not have any appearance but they have as a feature that can contain more than one widget, and can manage the space they take up and what occurs when the size of that space changes.

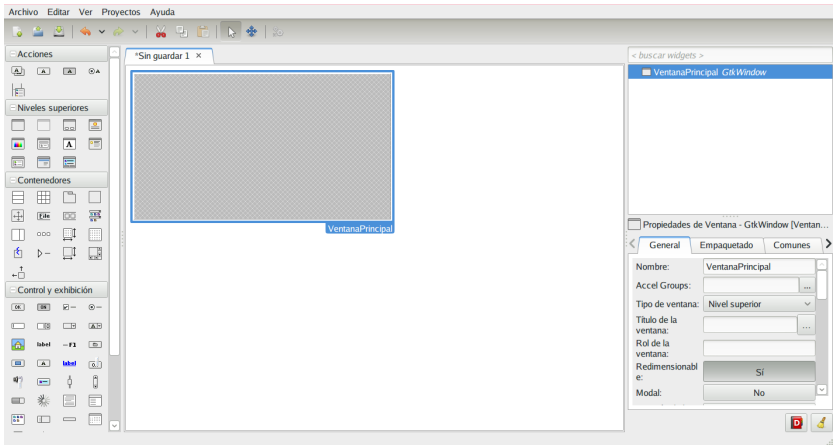
Glade (I)

- Glade is the official graphical constructor of user interfaces for Gtk+.
- It looks like this:



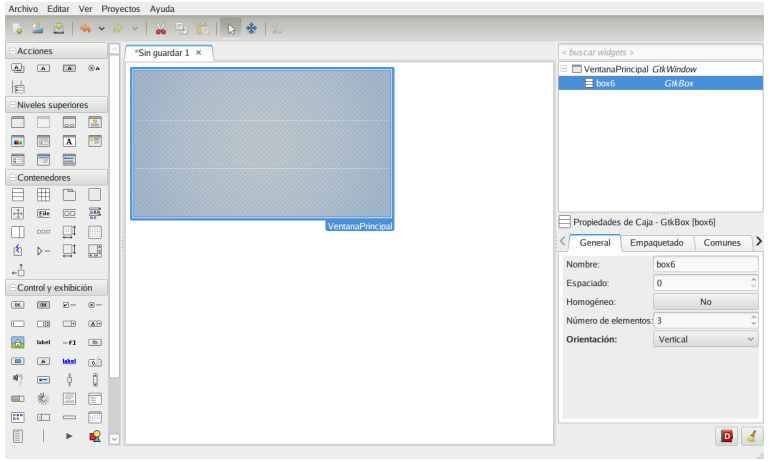
Glade (II)

We can create initial windows, dialogs, etc... picking them from the list of 'Niveles superiores':



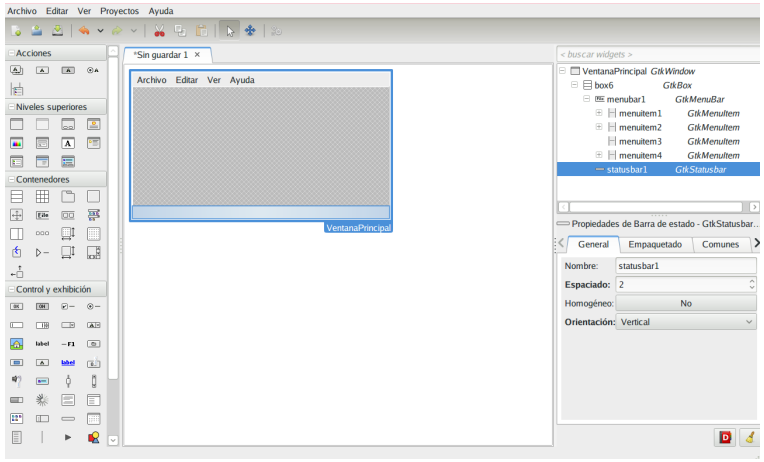
Glade (III)

We add the 'containers' needed to build our interface:



Glade (IV)

We insert in the 'empty spaces' of the 'containers' the widgets that we need, i.e. buttons, labels, textboxs, etc... :



Glade (V)

- Once we have created the interface we save it using the menu 'Archivo'.
- It saves it as a text file in `xml` format.
- These files usually have the extension '`.ui`'.
- We can load them from our application dynamically with an object of the '`Gtk.Builder` class'.
- Using the method '`add_from_file`' we can read the file '`.ui`'.
- Using the method '`get_object`' we can load one by one by its name the widgets that interest us of the '`.ui`' file.

Glade + Gtk+ (I)

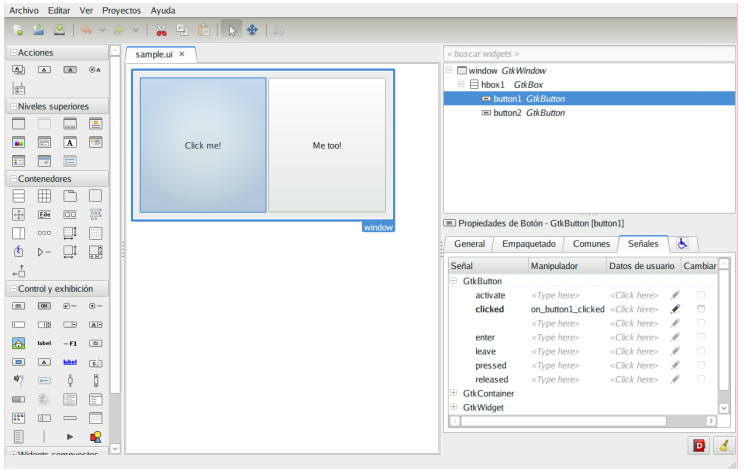
- Let's see a code example similar to the one of the slide 15.

```
// valac --pkg gtk+-3.0 --pkg gmodule-2.0 gtk-builder-sample.vala
using Gtk;
public void on_button1_clicked (Button source) {
  source.label = "Thank you!";
}
public void on_button2_clicked (Button source) {
  source.label = "Thanks!";
}
int main (string[] args) {
  Gtk.init (ref args);
  try {
    var builder = new Builder ();   // loader of Glade files
    builder.add_from_file ("sample.ui");  // load of the interface
    builder.connect_signals (null);       // auto conection of signals
    var window = builder.get_object ("window") as Window;
    window.show_all ();
    Gtk.main ();
  } catch (Error e) {
    stderr.printf ("Could not load UI: %s\n", e.message);
    return 1;
  }
  return 0;
}
```

The user interface can be downloaded from sample.ui

Glade + Gtk+ (II)

The `sample.ui` shown using Glade looks like this:



Glade + Gtk+ (III)

- We can connect methods as signal handlers.
- In this case we have to follow some rules to give names to the signals in Glade.
- Let's see it with an example:

```
using Gtk;

namespace Foo {
  public class MyBar {

    [CCode (instance_pos = -1)]
    public void on_button1_clicked (Button source) {
      source.label = "Thank you!";
    }

    [CCode (instance_pos = -1)]
    public void on_button2_clicked (Button source) {
      source.label = "Thanks!";
    }
  }
}

// ...
var object = new Foo.MyBar ();
builder.connect_signals (object);
// ...
```


Glade + Gtk+ (IV)

- If we declare the methods that will act as callbacks inside a class and/or inside a namespace...
- ... in Glade we have to write the namespace or the class to which the method belongs to, followed by the name of the callback method, using lower-case letters and separated by underline symbols.
- For instance: '`Foo.MyBar.on_button1_clicked`' in Glade would be: '`foo_my_bar_on_button1_clicked`', as we can see in the next image: