A short tutorial for *gtkmvc*

Roberto Cavada *

June 11, 2004

1 What is this?

gtkmvc is a practical implementation of *Model-View-Controller* and *Observer* patterns. The goal of this tutorial is to provide readers with sufficient awareness about what *gtkmvc* is and what it can do, in order to allow them to quickly decide whether it might be useful or not for their needs.

The tutorial is thought to be minimal. For any further information the user should refer to the manual pages.

2 The example in few words

The sample application must provide a single window, with a label showing the value of a numeric counter. The window contains also a button, which increments the counter by one every time it is pressed.

Since this tutorial must fit into few pages, the example is extremely simple. Moreover, a more convoluted example would not help to better understand what *gtkmvc* can be used for.

3 The framework

The implementation of this example is split into two parts. In the first one, the GUI is automatically constructed from a glade file. In the second one, the GUI is constructed *'by hand'*, by creating and connecting manually all widgets.

The latter solution is mainly presented to make this tutorial more complete, but the readers should keep in mind that they are going to adopt the former most of the times, or even more often a mixture of them, where many parts come from one or more glade files, and some others are built manually.

The use of glade files is also the reason why the *gtkmvc* framework is split into *three* distinct parts. It is a matter of fact that in practice the Model-View-Controller pattern will almost always collapse to a two-level framework, where

*ITC-irst, Trento, Italy, cavada@irst.itc.it

the View and the Controller are represented by a unique monolithic entity (let's call it *V&C*), and the Model is still separated by the rest.

The *gtkmvc* framework provides three well-distinguishable levels, to allow the pure-glade parts to go into the View side (in a direct and very natural way), and all the remaining parts that would be put in the *V&C* part, to go either in the Controller part, or in the View part, depending on how much close to the GUI stuff are.

For example, all the widgets signal handlers must go in the Controller side, whereas the code that sets some attributes of a specific widget might live either in the Controller or in the View, depending on how much those attributes are bounded to the application logic.

The more some code depends on the logic of the application, the farther it lives from the View side. If some code depends only on the logic without any relation with the GUI stuff, it must live in the Model.

4 The implementation *glade-based*

4.1 The model

The model is represented by class `MyModel`, derived from class `Model`, that in turn is provided by the framework.

The class `MyModel` contains a field called `counter` to hold the value of a numeric counter. Since we are interested in monitoring and show any change of this counter, we declare it as an *observable property*.

```
from gtkmvc.model import Model

class MyModel (Model):
    # observable properties:
    __properties__ = { 'counter' : 0 }

    def __init__(self):
        Model.__init__(self)
        return

    pass # end of class
```

All that it is required to do, is calling class `Model`'s constructor from within the derived class' constructor, and defining a class variable `__properties__` containing the name of the observable properties, and the associated initial values. The class `Model` will do all the boring work automatically.

4.2 The glade-based view

glade-2 while editing the example is depicted in figure 1. The names for the main window, the label and the button are significant, and signal `clicked` of the button has been associated with a function called `on_button_clicked`.

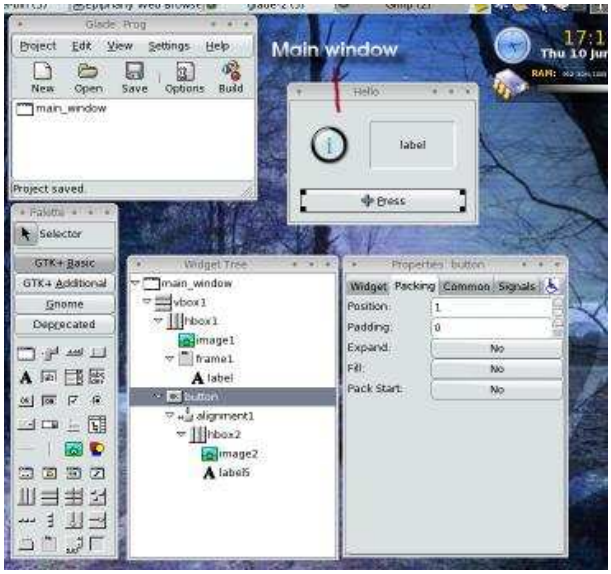

Figure 1: *glade-2* in action

The result is saved in `gtkmvc-example.glade`.

The view is represented by class `MyView`, that derives from class `View` provided by *gtkmvc*. The class `View` can be thought as a container that holds a set of widgets, and may associate each widget with a string name. When a glade file is used to build the view, each widget will be associated automatically inside the view with the corresponding name occurring in the glade file.

Moreover, each `View` instance is connected to a corresponding *Controller*, and when built from a glade file, methods inside the Controller will be scanned to try to connect automatically all signals declared in the glade file.

from gtkmvc.model import Model

```
# This is file model.py
from gtkmvc.view import View

class MyView (View):

    def __init__(self, ctrl):
        View.__init__(self, ctrl, 'gtkmvc-example.glade')
        return

    pass # end of class
```

Class `MyView` calls simply `View`'s class constructor from within its constructor, by passing the `Controller` instance which it belongs to, and the glade file name. All the hard work is carried out by class `View`.

4.3 The controller

The controller - so to speak - is the most complicated part of this example. It is the only part of the MVC pattern which knows the model and the view instances which it is linked to. These are accessible via members `self.model` and `self.view` respectively.

```
# This is file ctrl_glade.py
from gtkmvc.controller import Controller
import gtk

class MyController (Controller):

    def __init__(self, model):
        Controller.__init__(self, model)

        # The controller is an observer for properties contained in
        # the model:
        self.model.registerObserver(self)
        return

    def registerView(self, view):
        """This method is called by the view, that calls it when it is
        ready to register itself. Here we connect the 'pressed' signal
        of the button with a controller's method. Signal 'destroy'
        for the main window is handled as well."""

        Controller.registerView(self, view)

        # connects the signals:
        self.view['main_window'].connect('destroy', gtk.mainquit)

        # initializes the text of label:
        self.view['label'].set_text("%d" % self.model.counter)
        return

    # signals:
    def on_button_clicked(self, button):
        self.model.counter += 1  # changes the model
        return

    # observable properties:
    def property_counter_change_notification(self, model, old, new):
        self.view['label'].set_text("%d" % new)
        print "Property 'counter' changed from %d to %d" % (old, new)
        return


    pass # end of class
```

In the class constructor, at first base class constructor is called, passing the

`Model` instance this `Controller` instance belongs to. From that moment on, class member `self.model` will be accessible.

Then method `Model.registerObserver` is called, in order to make the controller an observer for the observable property `counter` in the model. After this, every change applied to `MyModel` class' member `counter` will make method `property_counter_change_notification` of class `MyController` be called automatically.

Method `registerView` is called when a class `View` instance requires to be registered to the controller it belongs to. This method is mainly used to connect signals and initialize the GUI side that depends on the application logic. In the example, signal `destroy` of the main window is connected to `gtk.mainquit` to close the application when the user closes the window. Notice here the use of member `self.view` and how a class `View` can be used as a map to retrieve widgets from their names.

Also, the text label is initialized to the initial value of the counter.

Method `on_button_clicked` is called as a callback every time the user clicks the button. The corresponding signal is automatically connected to this method when class `MyView` registers itself within the controller.

Finally, method `property_counter_change_notification` is called when the property `counter` in class `MyModel` changes. The model containing the property, the old value and the new value are passed to this method. Notice that the model is passed since the controller might be an observer for more than one models, even different from the model it is directly connected to in the MVC chain.

4.4 The main code

Main code is really trivial:

```
# This is file main_glade.py
import gtk
from model import MyModel
from ctrl_glade import MyController
from view_glade import MyView

m = MyModel()
c = MyController(m)
v = MyView(c)

gtk.mainloop()
```

5 The implementation *without glade*

5.1 The model

The model does not depend on the controller+view sides, so it is exactly the same as for the implementation *glade-based*.

5.2 The view

Using manually constructed views is slightly less intuitive that using glade-based views, since the architecture of the view-side *gtkmvc* is mainly designed to be used with glade files.

```
# This is file view_no_glade.py
from gtkmvc.view import View
import gtk

class MyViewNoGlade (View):

    def __init__(self, ctrl):

        # The view here is not constructed from a glade file.
        # Registration is delayed, and widgets are added manually,
        # later.
        View.__init__(self, ctrl, register=False)

        # The set of widgets:
        w = gtk.Window()
        h = gtk.VBox()
        l = gtk.Label()
        b = gtk.Button("Press")
        h.pack_start(l)
        h.pack_end(b)
        w.add(h)
        w.show_all()

        # We add all widgets we are interested in retrieving later in
        # the view, by giving them a name. Suppose you need access
        # only to the main window, label and button.  Widgets are
        # added like in a map:
        self['main_window'] = w
        self['label'] = l
        self['button'] = b

        # View's registration was delayed, now we can proceed.
        # This will allow the controller to set up all signals
        # connections, and other operations:
        ctrl.registerView(self)

        return

    pass # end of class
```

The entire work is carried out by the class constructor. At the beginning base class `View` is called like in glade-based view class, but now parameter `register` is set to False, to delay the registration of the view within the controller. This to allow manual construction of the widgets set, that later during registration

the controller will be able to access.

Following lines are used to build the widgets set, and to associate a few of them with string names.

Finally, last line calls method `registerView` of the controller, in order to at last allow the controller to know about this view.

Notice that here glade file has not been used at all. Nevertheless, a mixed solution where glade file(s) and manually constructed widgets sets is fully supported.

5.3 The controller

The controller is the same that has been used for the glade-based version, a part from a further signal connection that is performed to connect the button `clicked` event to class method `self.on_button_clicked`. For this reason, class `MyControllerNoGLade` is derived from class `MyController` to reduce typing.

```
# This is file ctrl_no_glade.py
from ctrl_glade import MyController

class MyControllerNoGlade (MyController):

    def __init__(self, model):
        MyController.__init__(self, model)
        return

    def registerView(self, view):
        MyController.registerView(self, view)

        # connects the signals:
        self.view['button'].connect('clicked', self.on_button_clicked)
        return

    pass # end of class
```

5.4 The main code

Like previous version, main code for manually built view is very short:

```
# This is file main_no_glade.py
import gtk
from model import MyModel
from ctrl_no_glade import MyControllerNoGlade
from view_no_glade import MyViewNoGlade

m = MyModel()
c = MyControllerNoGlade(m)
v = MyViewNoGlade(c)

gtk.mainloop()
```

6 More convoluted main code

Finally, this example shows the powerful of the *Observer* pattern.

Here both the glade-based and manually built versions are being run at the same time, with a single instance of class `MyModel` shared between those two versions. The execution of this example results in two windows being displayed; by clicking the button of one of them, the counter is incremented, and the labels in both of them are updated.

```
# This is file main_mixed.py
import gtk
from model import MyModel
from ctrl_no_glade import MyControllerNoGlade
from ctrl_glade import MyController
from view_no_glade import MyViewNoGlade
from view_glade import MyView

m = MyModel()

c1 = MyControllerNoGlade(m)
c2 = MyController(m)

v1 = MyViewNoGlade(c1)
v2 =  MyView(c2)

gtk.mainloop()
```

7 Conclusions

The author does hope that this tutorial will be useful to help those who are unsettled about whether gtkmvc can fit their needs or not.

Even if very simple, from this tutorial should result clear to the reader that both the MVC and Observer patterns can strongly improve the quality of middle and big size GUI applications, especially if combined with the use of glade-based views.

gtkmvc has been extensively used to produce a few large GUI applications based on *Python* and *Pygtk-2*. In this scenario, many design choices that led to *gtkmvc* had been determined from practical needs, and this made easiness and transparency the most appreciated quality of the framework.