A Model-View-Controller pattern implementation for Pygtk2
Version 0.9.0

Roberto Cavada *

January 15, 2004

1 Introduction

This document contains information about the functionalities and the architecture of a Model-View-Controller Infrastructure (MVC from here on) for *Pygtk* version 2. The aim is to supply the essential information in order to make everyone able to interact, modify and extend the infrastructure, as well as for creating new applications based on this infrastructure. This document is not complete, and it has been extracted from another document which describes the architecture of *gNuSMV* [1], the new GUI for the *NuSMV* model checker[2]. *gnusmv* is strongly based on the MVC pattern implementation for *Pygtk* we developed at ITC-irst.

Section 2 briefly gives an overview of the general architecture for a *gNuSMV* application based on Python and the GTK toolkit, showing all major parts, and how these depend on each other.

Section 3 describes the basement of a GUI application, the Model-View-Controller Infrastructure. An example via a simple Sequence Diagram is also provided, in order to better fix concepts.

Finally, Sections 4 and 5 supply some further details about implementation, via an example. The example aims to make more concrete the ideas described in all previous sections.

2 Architectural Overview

Figure 1 shows the high level software architecture for an application based on Pygtk and the supplied MVC Infrastructure. It shows the functional architecture as well.

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

[1] See at http://nusmv.irst.itc.it/gnusmv

[2] See at http://nusmv.irst.itc.it

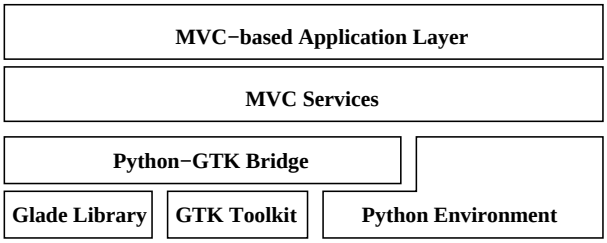

Figure 1: High-Level Block Architecture

In terms of functionalities, at the highest level is located the *Application Layer*, which is partially based on the MVC Infrastructure, and whose implementation depends on the application semantics.

The *MVC Services Layer* supplies a quasi-generic platform which implements the MVC pattern. In the current implementation, the View part partially depends on the GTK graphical toolkit.

Lower layers supply several functionalities concerning the graphical toolkit (GTK and *Glade*) and the Scripting Environment (Python).

3 MVC Infrastructure

The MVC pattern used inside *gNuSMV* is a simplified version of the "official" pattern generally described by Software Engineering Theory (for example, see *http://www.object-arts.com/EducationCentre/Overviews/MVC.htm*).

Figure 2: Simplified Model-View-Controller Pattern

Simplification consists in the fact that in the more general pattern the Constroller-View relationship allows a 1-N cardinality. Also the implementation aims to highlight and address *practical* requirements (w.r.t. the GTK toolkit), rather than to be strictly and formally faithful implementation of the pattern.

Figure 2 shows three interconnected parts:

Model Contains the *state* of the application. Also it provides support to access and modify the state, and knows how to handle dependencies between different parts in the state. For example the application logic could require that changing a variable, causes a changing of another. It is not required the Model's user to be aware about this dependency, because Model autonomously handles it.

Zero, one or more *Controllers* can be connected to one Model (see *Controller*, below). Furthermore, one or more *Views* can be associated with parts of the state; for example a numerical variable could be visualized as a number, as well as a graphic bar. It is important to remark that *a Model does not know that a set of Views are connected to its state.*

View Views show parts of the Model state, and interactively exchange information with the User, via input/output devices. View also interacts with the *Controller* (see below), sending event-associated signals to the Controller, and receiving information to visualize.

A View also associates a set of widget trees, deriving from the *Glade* File, as well as from the ad-hoc View Representation. Since a Widget contains a *state*, this implementation differs from the standard MVC pattern, where generally the View side is completely *stateless.*

As for the Model, a View does not know the semantics concerning what it visualizes, as well as the Model it is connected to.

Controller The Controller realizes the connection between Models and Views. The Controller contains the GUI logic: for example, it stores the information about what happens when a button is clicked (i.e. handlers of signal are located inside the Controller.)

Two particular mechanisms make the isolation between Model and Controller, and between View and Controller (see sections 3.1 and 3.2 below).

A Controller perfectly knows how the connected Model and View are implemented, and knows both the state and presentation semantics. A Controller is associated to one Model (*has a* relationship), and in the current implementation is associated only to one View.

3.1 Observable properties

Models do not know that are connected to a set of controllers, because this knowledge implies the knowledge of the GUI semantics, which should be out-of-scope for Models.

Nevertheless, sometimes it is necessary a Model notifies the GUI logic (generally the Controllers set, but also other models) that the state changed. This practical requirement has been allowed by extending the Model state with a mechanism called *Observable Properties.* An observable property is a part of the Model state which is also externally observable via an *Observer.* Every time an observable property changes, any interested Observer will be notified of the event.

Figure 3 shows a Model containing an observable property (*color*). When *color* changes to red, all connected Observers will be notified. Each Observer will then perform the necessary operation in order to make the View showing the occurred change.

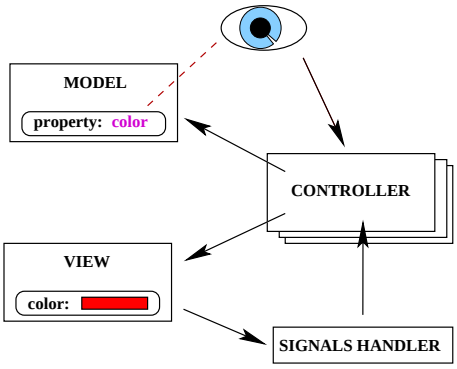

Figure 3: Interaction between Model-Controller and View-Controller

Each Observer declares it is interested in receiving notifications on one or more properties changing, by a mechanism called *Registration*. Once an Observer (for example, a Controller) registered itself among the Model it is associated with, it will be notified of all changes of the observable properties. The Observers will be notified only of the property changes that they are actually interested in observing.

An implicit syntactical rule binds observable properties names to notifications sockets inside Observers. This rule allows an automatic connection, and fixes a sort of "rule" for methods names.

Later in this document, some implementation details are discussed, and further details about observable properties are presented. Finally, an example in the latest part should make all these concepts clearer.

3.2 View Registration

Current implementation allows only a 1-1 relationship between Controller and View. Anyway a registration mechanism has been used to connect those two parts, allowing for more generic relationship in the future, when a Controller could handle more than one Views, or a View can be shared between different Controllers.

After the creation, a Vies must register itself with a Controller. From there on, the Controller can access the state and methods inside the View. When the view registers itself with a Controller, all signals are also automatically connected to the corresponding semantics inside the Controller. Connection in this case is performed by means of an implicit syntax rule, which binds a signal name to a corresponding function name.

4 Details of implementation

This section presents some details regarding the implementation of the MVC framework in Python.

4.1 Models, Controllers and Views

The MVC Infrastructure essentially supplies three base classes which implement respectively a View, a Model and a Controller. Developers must derive custom classes from the base classes set, adding the implementation which depends on the application semantics.

Model base class Supplies servicing for:

- Fully automatic Observable Properties
- Automatic broadcast notification when observable properties change.

Controller base class Supplies servicing for:

- Automatic registration inside the associated Model.
- Easy access to the associated Model and View for any derived class.

View base class Supplies servicing for:

- Automatic widgets tree registration. Input can be a set of root widgets stored inside a *Glade* File, or a completely customized widgets hierarchies.
- Automatic registration inside the associated Controller.
- Automatic signals connection to methods supplied by the associated Controller.
- Widget retrieval inside the set of hierarchy. Widget can be accessed by using the name they have been defined from within *Glade*, at design time.

4.1.1 Model

User's models must derive from this base class. Models must be used to hold the *data* of the application. They can be connected to observers (like Constrollers) by a mechanism discussed by section 4.2. It is important to note that apart from the registration phase, the model do not know that there exists a set observers connected to it.

All the code strictly related to the data of the application (i.e. not related to any view of those data) will live in the model class.

4.1.2 Controller

User's controllers must derive from this class. A controller is always associated with one model, that the controller can monitor and modify. At the other side the controller can control a View. Two members called `model` and `view` holds the corresponding instances.

Typically (but not always!) a Controller instance is also an Observer for some Observable Property inside the controlled model. Indeed, the constructor gets a model instance, that the controller uses to register itself as an observer of the model.

An important method that user can override is `registerView`, that the associated view will call during registration. This can be used to connect custom signals to widgets of the view, or to perform some initialization that can be performed only when model, controller and view are actually connected. `registerView` gets the view instance that is performing its registration within the controller. If user overrides `registerView`, they must call method `registerView` of the base class.

The controller holds all the code that lives between data in model and the view. For example the controller will read a property value from the model, and will send that value to the view, to visualize it. If the property in the model is an Observable Property that the Controller is interested in monitoring, than when somebody will change the property, the controller will be notified and will update the view.

4.1.3 View

User's views derives from base class View. This is the only part specific for the *Pygtk* graphic toolkit.

A View is always associated to a Controller (that gets with its constructor call). When the view is created, it register itself to the controller by calling method `Controller.registerView`.

A View is also associated to a set of widgets. In general, this set can be organized as a set of trees of widgets. Each tree can be optionally be generated by using the *Glade* application (see section 5.1).

The View contructor is quite much complicated:

```
def __init__(self, controller, glade_filename=None,
             glade_top_widget_name=None, parent_view=None,
             register=True)
```

glade_filename can be a string or a list of strings. In any case weach string provided represents the file name of *Glade* output. Typically each glade file contains a tree of widgets.

glade_top_widget_name can be a string or a list of strings. Each string provided is associated to the parameter glade_filename content, and represent the name of the widget in the widgets tree hierarchy to be considered as top level. This let the user to select single parts of the glade trees passed.

parent_view is the view instance to be considered parent of self. Generally this parameter is None.

register is a flag used to delay view's registration. If your derived view class adds some widgets "manually" by creating on the fly them (see below), you want to delay the view registration (performed by the View class contructor) 'till all ad-hoc widgets have been actually created. Since the View's constructor must be called at the beginning of your derived view class, you can avoid the View constructor calling Controller.registerView by setting this flag to False. After your view class contructor built all the widgets, it is responsible for calling Controller.registerView to perform the registration. (This is definitly more complicated to explain than to understand...)

The View class also can be considered a map, that associates widget names to the corresponding widget objects. If file `test.glade` contains a Button you called `start_button` from within *Glade*, you can create the view and use it as follows:

```
from gtkmvc.view import View

class MyView (View):
  def __init__(self, controller):
    View.__init__(self, controller, ‘‘test.glade’’)
    return
  pass

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

v['start_button'] # this returns a gtk.Button object
```

Instead of using only *Glade* files, sometimes the derived views create a set of widgets on the fly. If these widgets must be accessed later, they can be associated simply by (continuing the code above):

```
v['vbox_widget'] = gtk.VBox()
...
```

Typically the creation on the fly of new widgets is performed by the derived view contructor, that will delay the view registration.

Another important mechanism provided by the class View is the signal autoconnection. By using *Glade* users can associate to widget signals functions and methods to be called when associated events happen. When performs the registration, the View searches inside the corresponding Controller instance for methods to associate with signals, and all methods found are automatically connected.

4.2 Observable Properties in details

The mechanism of the Observable Properties (OP) is fully automatic, since its management is carried out by the base class Model.

Basically the user derives from class Model, and adds a class variable called `__properties__` . This variable must be a map, whose elements' keys are names of properties, and the associated values are the intial values.

For example, suppose you want to create an OP called `name` initially associated to the value "Rob":

```
from gtkmvc.model import Model

class MyModel (Model):
  __properties__ = { 'name' : 'Rob' }

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

  pass # end of class
```

That's all. By using a specific metaclass, property `name` will be automatically added, as well as all the code to handle it.

This means that you can use the property in this way:

```
m = MyModel()
print m.name  # prints 'Rob'
m.name = 'Roberto' # changes the property value
```

What's missing is now an observer, to be notified when the property changes:

```
class AnObserver :

  def __init__(self, model):
    model.registerObserver(self)
    # ...
    return

  def property_name_change_notification(self, model, old, new):
    print ``Property name changed from '%s' to '%s''' % (old, new)
    return

  pass # end of class
```

The constructor gets an istance of a Model, and registers the class instance itself to the given model, to become an observer of that model instance.

To receive notifications for the property `name`, the observer must define a method called `property_name_change_notification` that when is automatically called will get the instance of the model containing the changed property, and the property's old and new values.

As you can see, an Observer is not required to derive from a specific class. Anyway, in the MVC framework models and mostly controllers are use also as observers.

Here follows an example of usage:

```
m = MyModel()
o = AnObserver(m)

print m.name  # prints 'Rob'
m.name = 'Roberto' # changes the property value, o is notified
```

Things so far are easy enough, but they get a bit complicated when you derive custom models from other custom models. For example, what happens to OP if you derive a new model class from the class `MyModel`?

In this case the behaviour of the OP trusty follows the typical Object Oriented rules:

1. Any OP in base class are inherited by derived classes
2. Derived class can override any OP in base classes
3. If multiple base classes defines the same OP, only the first OP will be accessible from the derived class

For example:

```
from gtkmvc.model import Model

class Test1 (Model):

    __properties__ = {
        'prop1'  : 1
        }

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

# this class is an observer of its own properties:
        self.registerObserver(self)
        return

    def property_prop1_change_notification(self, model, old, new):
        print "prop1 changed from '%s' to '%s'" % (old, new)
        return
    pass # end of class


class Test2 (Test1):

    __properties__ = {
        'prop2'  : 2,
```

```
        'prop1'  : 3
        }

    def __init__(self):
        Test1.__init__(self)

# also this class is an observer of itself:
        self.registerObserver(self)
        return

    def property_prop2_change_notification(self, model, old, new):
        print "prop2 changed from '%s' to '%s'" % (old, new)
        return
    pass

# test code:
t1 = Test1()
t2 = Test2()


t2.prop2 = 20
t2.prop1 = 30
t1.prop1 = 10
```

When executed, this script generates this output:

```
prop2 changed from '2' to '20'
prop1 changed from '3' to '30'
prop1 changed from '1' to '10'
```

As you can see, `t2.prop1` overrides the OP `prop1` defined in Test1 (they have different intial values). Test2 could also override method `property_prop1_change_notification`:

```
class Test2 (Test1):
  # ... copy from previous definition, and add:

  def property_prop1_change_notification(self, model, old, new):
    print "Test2: prop1 changed from '%s' to '%s'" % (old, new)
    return

  pass
```

As you expect, the output in this case would be:

```
prop2 changed from '2' to '20'
Test2: prop1 changed from '3' to '30'
prop1 changed from '1' to '10'
```

4.2.1 Special members for Observable Properties

Classes derived from Model, that exports OPs, have several special members. Advanced user might be interested can override some of them, but in general they should be considered as private members.

`__properties__` A class (static) member that maps property names and intial values. This must be provided as a map by the user.

`__derived_properties__` Automatically generated static member that maps the OPs exported by all base classes. This does not contain OPs that the class overrides.

_prop_*¡property_name¿* This is an autogenerated variable to hold the property value. For example, a property called `x` will generate a variable called `_prop_x`.

get_prop_*¡property_name¿* This public method is the getter for the property. It is automatically generated only if the user does not define one. This means that the user can change the behaviour of it by defining their own method. For example, for property `x` the method is `get_prop_x`. This method gets only self and returns the corresponding property value.

set_prop_*¡property_name¿* This public method is customizable like `get_prop_<property_name>`. This does not return anything, and gets self and the value to be assigned to the property. The default autogenerated code also calls method `gtkmvc.Model.notify_property_change` to notify the change to all registered observers.

For further details about this topic see metaclasses `PropertyMeta` and `ObservablePropertyMeta` from package `support`.

5 A simple application

This section describes the process of creation of a sample application, from the design with *Glade*, to the integration of views and code inside the MVC Infrastructure.

We want to design and implement a simple application constituted by only one window, containing two string labels. One label shows a text, while the other shows the number of characters displayed (i.e. the length of the string) by the first one. There is also a button the user can press. By pressing the button, the user can change the displayed text, and of course this action might change also the displayed text length. Figure 4 gives an idea on how the application should appear.

Figure 4: The sample Application

5.1 *Glade*

Figure 5 shows *Glade* and a project named `example`. The sample GUI has only one top-level window (named `window1`).

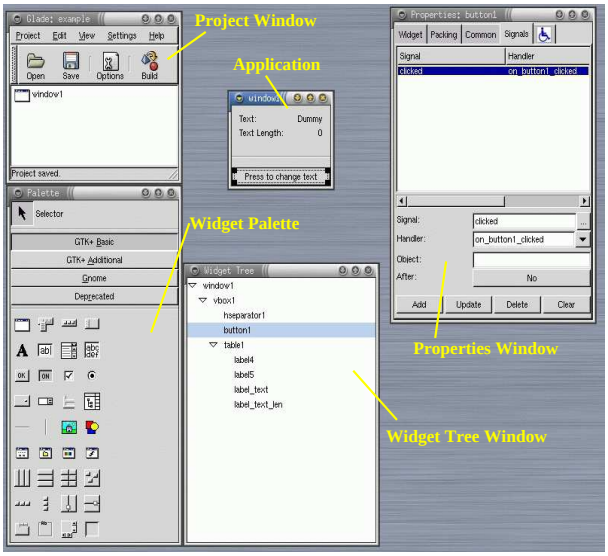

Figure 5: Designing the example by means of *Glade* for GTK2

The *Widget Tree Window* shows the widgets hierarchy. There are essentially the three main components (one button and two labels), grouped inside a set of *containers*, which supplies alignments and resizing capabilities.

On the right side of Figure 5, the *Properties Window* shows that the widget named `button1` has signal `clicked` associated with function `on_button1_clicked`. This means that the Controller will have to supply this function in order to handle the `click` event occurring in `button1`.

5.2 Implementation

The implementation is slightly elaborate for this example, because the goal here is to show how the sample application can be implemented by using the MVC Infrastructure.

A basic knowledge of any Object Oriented programming language is sufficient to understand how this example has been pushed inside the MVC Infrastructure. On the contrary, a fair knowledge of the Python language is required in order to understand the code details.

More description section is 4.

5.2.1 View

In the example, the View is implemented inside the class `ExampleView` shown below.

```
from gtkmvc.view import View
import os.path

GLADE_NAME = "example.glade"
GLADE_PATH = "./glade"
GLADE = os.path.join(GLADE_PATH, GLADE_NAME)

class ExampleView (View):
    """The application view. Contains only the main window1 tree."""

    def __init__(self, controller):
        """Contructor, takes the controller instance to perform registration"""
        View.__init__(self, controller, GLADE, "window1")

        return
    pass # end of class
```

Global variables named `GLADE*` identify the *Glade* File to be used when loading the GUI representation generated by *Glade*.

Class `ExampleView` extends the generic `View` class, which performs most of the job, as described above.

5.2.2 Model

Class `ExampleModel` is as simple as class `ExampleView`. As for `ExampleView`, it extends a base class of the MVC Infrastructure, class `Model`. The state is represented by a set of possible messages, as well as by the current message index. The current message index is also an observable property. A couple of methods are supplied in order to access the state.

```
from gtkmvc.model import Model

class ExampleModel (Model):
    """The model contains a set of messages
    and an observable property that represent the current message
    index"""

    # Observable property: code for that is automatically generated
    # by metaclass constructor. The controller will be the observer
    # for this property
    __properties__ = {
        "message_index" : -1 # -1 is the initial value
        }

    def __init__(self):
```

```
        Model.__init__(self)

        self.messages= ('Initial message',
                        'Another message',
                        'A third message...',
                        'Model changed again')
        return

    def get_message(self, index): return self.messages[index]

    def set_next_message(self):
        # this changes the observable property:
        self.message_index = (self.message_index + 1) % len(self.messages)
        return

    pass # end of class
```

Notice the class' variable `__properties__`, which is a map of (property, value) couples. The base class Model belongs to a metaclass which automatically search for observalbe properties and generates the needed code to handle the notification. When the value of variable `message_index` changes, all registered observers will be notified.

5.2.3 Controller

Class `ExampleController` contains the *logic* of the application. The controller handles two signals and the observable property notification. Signals are the `destroy` event, invoked when the application quits, and the `on_button1_clicked`, fired when `button1` is pressed.

```
from gtkmvc.controller import Controller
from gtk import mainquit

class ExampleController(Controller):
    """The only one controller. Handles the button clicked signal, and
    notifications about one observable property."""

    def __init__(self, model):
        """Contructor. model will be accessible via the member 'self.model'.
        Registration is also performed."""
        Controller.__init__(self, model)
        return

    def registerView(self, view):
        Controller.registerView(self, view) # Calls the overridden method

        # Connects the exiting signal:
        view.get_top_widget().connect("destroy", mainquit)
```

```
        return

    # Signal
    def on_button1_clicked(self, button):
        """Handles the signal clicked for button1. Changes the model."""
        self.model.set_next_message()
        return

    # Observables notifications:
    def property_message_index_change_notification(self, model, old, new):
        """The model is changed and the view must be updated"""
        msg = self.model.get_message(new)

        self.view['label_text'].set_text(msg)
        self.view['label_text_len'].set_text(str(len(msg)))
        return
    pass # end of class
```

The `destroy` signal is connected when the View registers itself inside the controller, by using the method override of `registerView`. Method `on_button1_clicked` calls a method inside the model which changes a part of the state inside the model. Since that part of the state is an observable property, the associated observer (which is the controller itself) is notified of the modification, by calling method `property_message_index_change_notification`. This method updates the view connected to the controller.