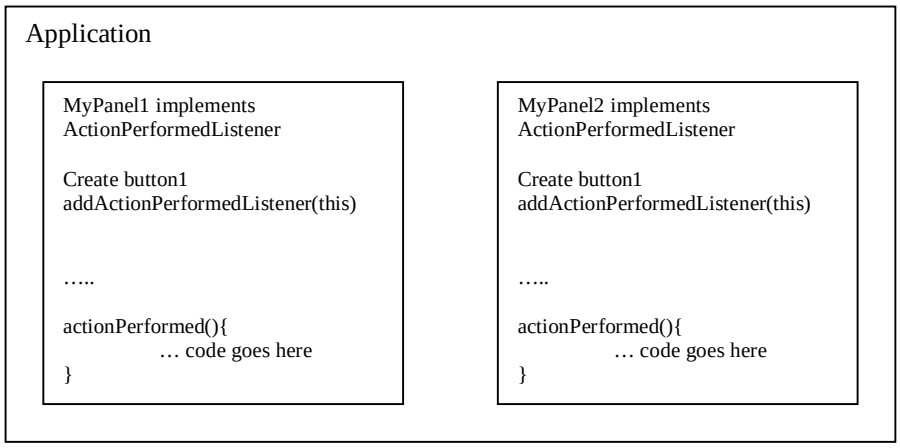
The OPController

1. What is it?

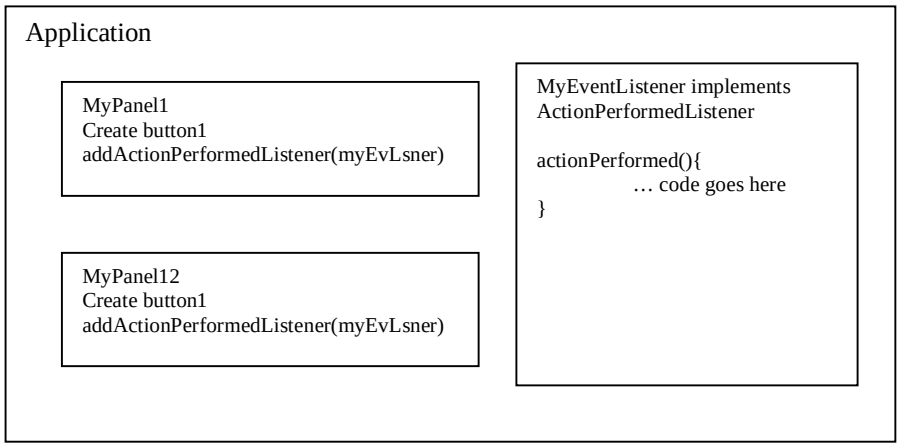
The OPController was conceived to attempt to give a *standard* way for developers in SL to organise their callbacks (listeners in Java terms). The aim was to effectively separate the visual part of the application (ie the panel with buttons, lists etc) from the code that would perform tasks in response to users' requests (ie a button press). In addition, the controller attempts to ease the burden of setting up potentially hundreds of listeners and gives the developer one method that will perform this task automatically. It is hoped that developers who choose the OPController method of building GUIs, will produce well structured, clearly defined, and *standard* applications.

2. What's wrong with the normal method of 'controlling' my application?

Generally, developers will develop classes that encapsulate both GUI components (eg buttons) and the corresponding event listeners.



This is normally the default for users of a GUI tool such as JBuilder (although the syntax differs slightly). The alternative, is to collect the event listeners in a separate class like this:



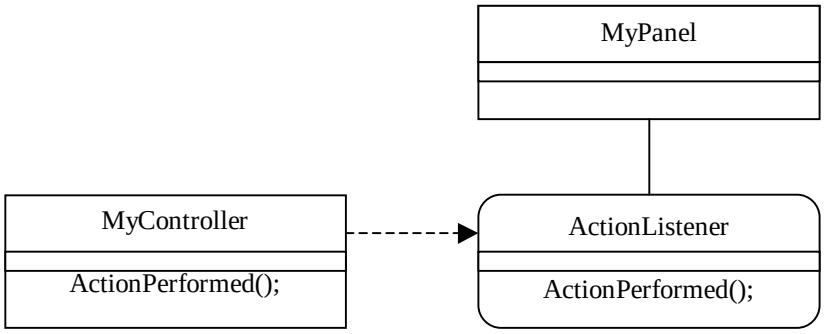
Use of this method, clearly separates the *actions* from the GUI, and makes the application's *action* code clearly defined and easily maintainable (ie to fix a bug, the developer doesn't need to wade through hundreds of lines of irrelevant code to find the problem!) It should be made clear at this point, that choosing to separate the actions from the GUI components does not follow the object oriented nature of

Java, and Java purists should stop reading here! The potential advantages of doing this however, may make you decide to choose this method.

With this arrangement, we satisfy the first *wish* made in section 1 – To separate the code from the GUI. We still however, need to attach many callbacks to our components. This is were the OPController framework can be used.

3. The OPController

The OPController, is basically a generic listener that can also hold other information pertaining to the actions of an application. In its basic form, the controller has no *listening* capabilities. Therefore, in order to create a useful controller, you will need to extend the base class, and implement the listeners you're interested in.



In Java terms this equates to the following:

```
MyController extends OPController implements ActionListener
```

Stating that you will implement the ActionListener interface, means that you must also implement the actionPerformed() method to collect the resulting events.

Now that your controller has ActionListener capabilities, you can call one of the methods in the controller (inherited from OPController) to register your panel with this controller. The following methods are currently available in the OPController.

- addListenersNeeded(Container)
  - Use this method to add listeners for the interfaces defined by your controller to all children of the given container. Normally the container is an OPPanel, but it could be a menu for example.
- addListenersNeeded(Container, Class[])
  - Use this method to add listeners for the specified interfaces to all children of the given container. Of course, your controller will need to implement the specified interfaces for this to work. An example here would be

```
myController.addListeners(myPanel,new Class[]{ActionListener.class});
```

- addListenersNeeded(Vector)
  - Use this method to add listeners for the interfaces defined by your controller to the given components (contained in the Vector).
- addListenersNeeded(Vector, Class[])
  - Use this method to add listeners for specified interfaces defined by your controller to the given components (contained in the Vector). Again, your controller must implement the given listeners.

Normally, you would use the first variant of the addListenersNeeded function. The other methods are supplied for special cases.

4. Other 'Objects'

As well as managing callbacks for GUI components, the controller also has the facility to store references to 'Objects'. Anybody who has written Java applications will know that passing references

to arbitrary objects between different components of an application can cause considerable problems. Theoretically, this *reference* problem can generally be solved by good, structured design of the OO hierarchy. In reality, programmers of new Java software in SL will probably opt for a method similar to the one supplied by the OPController, where they can expose references just like they do in 'C'. That is not to say, that objects in the controller are akin to 'C' global variables however!

The OPController solution to this problem is this. If you inform the OPController of the existance of an Object, by passing its reference, the controller code (ie your code) can access this object by querying its controller. For example, if I have an object of type 'CurrentUser.class', and I have two controllers (c1 & c2) that wish to examine this class, you would have something like the following:

```
CurrentUser user;
c1.addObjectReference(user);
c2.addObjectReference(user);
```

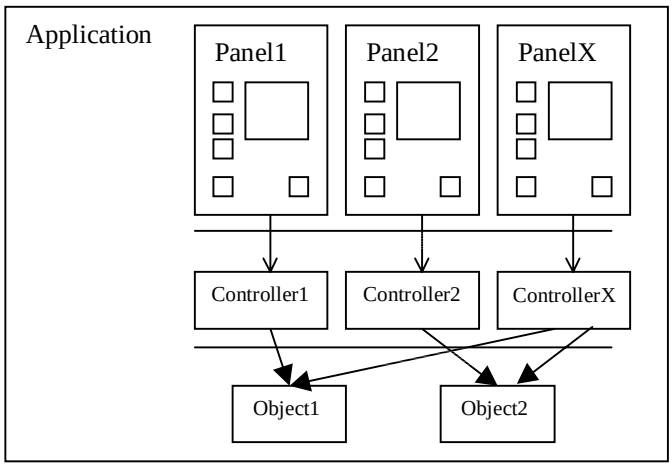
From anywhere within the controller, it is then possible to display the user information like this:

```
CurrentUser user = (CurrentUser)(this.getObject(CurrentUser.class));
System.out.println("CurrentUser::"+user);
```

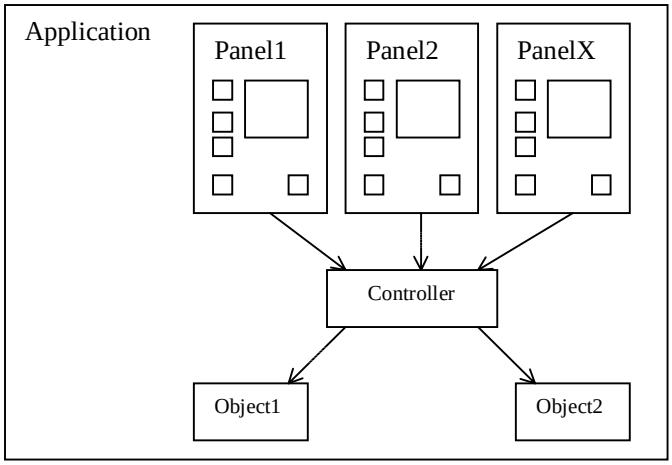
The more astute Java programmers, will be thinking "what if I have two objects of the same class?". The answer is that, at the moment, you can't! Therefore, if you want to pass an object reference to a controller, it needs to be a unique class. This in itself is not a bad thing, as it may encourage developers to encapsulate data and methods in class files, where they might otherwise have not done so. Nevertheless, this shortcoming still exists, and may be addressed in the future if demand dictates.

5. One for all, or all for one?

Given the design of the OPController, you have the freedom to organise your panels, objects and controllers in any way you wish. You could for example, have an application with 6 OPPanels contained in a tabbed pane, each with its own controller which will be given Object references to the objects it needs to act on.



On the other hand, you could choose to have one controller for your entire application like this:



Whichever way you choose depends on the application and is up to you.

6. Writing the callbacks

The way in which you will write your callbacks is down to you. It's important to note however, that for efficiency reasons, you may want to determine *which* Panel an event was fired from (this avoids countless if/else combinations for large GUIs). In order to facilitate this, the controller provides some convenience functions to discover who is firing the events. These are as follows:

- `getOPPanel(Object obj)`
  - This function will return a registered OPPanel, which is the visual parent of the given object (which should be a GUI component reference obtained from the event source). In order for this to work, you should have added the component through the function

    `'myController.addListenersNeeded(myMainP)'.`

    Once you know the parent of the object, you can create your event handler as follows:

```
Public void actionPerformed(ActionEvent e){
   // get the annonymous parent
   OPPanel parent = getOPPanel(e.getSource());

   // find which panel it came from
   if(parent instanceof MyPanel){
         // we know it came from MyPanel
         MyPanel myP = (MyPanel)parent;
         // find the exact component
         if(e.getSource() == myP.button1){
               // do something
         }
   }else if(parent instanceof MyPanel2){
         // we know it came from MyPanel2
         MyPanel2 myP = (MyPanel2)parent;
         //find the exact component
         if(e.getSource() == myP.text1){
               // do something
         }
   }
   // etc
```

    You should see, that using this method, you can separate events from different panels and act on them accordingly.

- `getOPPanel(Class toSearchFor)`

- This function will return a registered OPPanel, which has the same *class* as the given argument. So, if you had a *distinct* class for you panel, called 'MainPanel.class', and you added it to the controller by performing 'myController.addListenersNeeded(myMainP)', you can retrieve a reference to this panel in you callback. With this reference, you can create, for example, an ActionListener callback like this:

```
Public void actionPerformed(ActionEvent e){
   MainPanel myMainPanel = (MainPanel)getOPPanel(MainPanel.class);

   if(e.getSource() == myMainPanel.button1){
         // do something
   }
   // etc
```

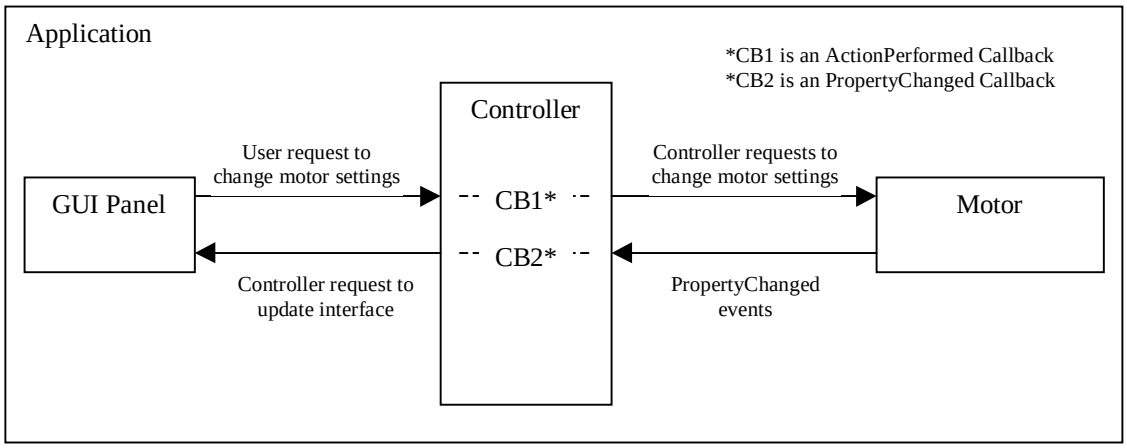
  Note that for this method to work, the class must *extend* from OPPanel.

  Use of this method to determine *who* fired an event is not recommended for controllers of many panels. The reason for this is that in order to determine which component fired the event, you may have to test against every component from every registered panel in the controller. Therefore, it's recommended that you stick with the first method.

7. Not just GUI components can fire events…

The OPController is primarily designed for handing events from a GUI. If you examine the controller however, you will see that it is basically a generic event listener. Therefore, you can ask the controller to listen events fired from *any* source.

For example, say you have a motor device, which fires a PopertyChangedEvent every 5 seconds. So long as the controller implements the PropertyChangedListener interface, there is no reason why it can't respond to the fired events. Furthermore, if you want to update a textfield in a GUI with the latest values from the motor, you need only register the OPPanel containing the textfield to be able to write a simple callback for the motor. To complete the application, you can control the motor settings through the controller, by responding to user button clicks for example.



The resulting application is clear, and can be documented very easily.

8. Useful examples and references

Source code example: http://proj-stopmi.web.cern.ch/proj-stopmi/MotorExample.html
StopMI Wizard: \\pcslux10\production
StopMI Docs: http://proj-stopmi.web.cern.ch/proj-stopmi/sdocmi/Packages.html