Using Stopmi Tool

INTRODUCTION

Welcome to Stopmi, a new way to create graphical user interfaces using the Java programming language. This guide describes several steps required to use the stopmi framework: how to install the stopmi library in your development tool and how to use those libraries.
Please read this documentation which contains a quick-start information to take off with Stopmi as well as answers to the most frequently asked questions.
For further information on the documentation available please consult the Stopmi web page at:

http://proj-stopmi.web.cern.ch/proj-stopmi/

INCLUDING STOPMI LIBRARIES IN JBUILDER.

STEP ONE: MAPPING ONE OF YOUR DISKS IN WINDOWS EXPLORER.

The stopmi libraries are installed on a Unix server and can be accessed from a MS WINDOWS environment. All you need is to map one of your disks on \\pcslux10\production (To map in Windows explorer: Tools/MapNetwork Drive,and then fill in the path specified choosing a disk to map).

STEP TWO: ADDING STOPMI LIBRARY TO JBUILDER.

To add a new library in Jbuilder you need to follow these steps:

1. Once you have Jbuilder opened, press on the "*Project*" menu item in the menu bar.

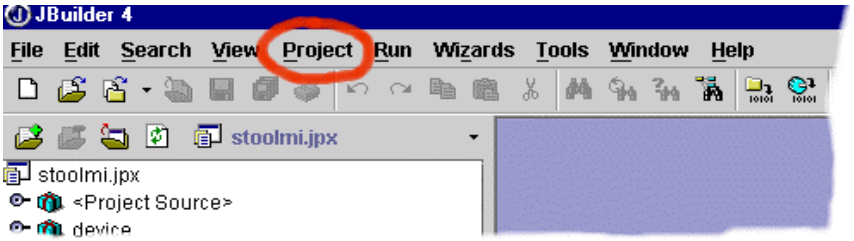


Fig. 1.1

2. Select "*Default Project Properties*" from the pull down menu.

3. A dialog box appears with different tabs on the top. Stay on the "*Paths*" tab that is selected by default, press on "*Required libraries*" which is another tab located in the middle of the dialog box.

4. Press the "*Add*" button.

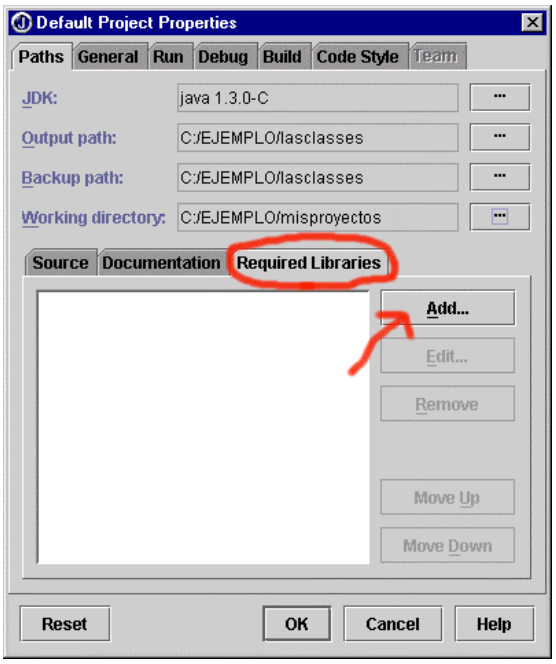


Fig.1.2

5. A dialog box appears where you can include those libraries you want the tool to read when compiling your files. First of all you need to create a library from stopmi files.

6. Press "*New*", a button you can find on the bottom of the dialog box.

7. The following window appears:

Fig.1.3

Fill in the first field specifying the name of the library, e.g. "Stopmi". Leave the location option as it is set by default ("*User Home*"). Then press "*Add*" to specify the path to the stopmi files.

8. You will see another window that lets you manage your files. On the disk you have mapped with the stopmi files, select the "*stopmiLIB.jar*" file which is under the *stopmi* directory and press OK.

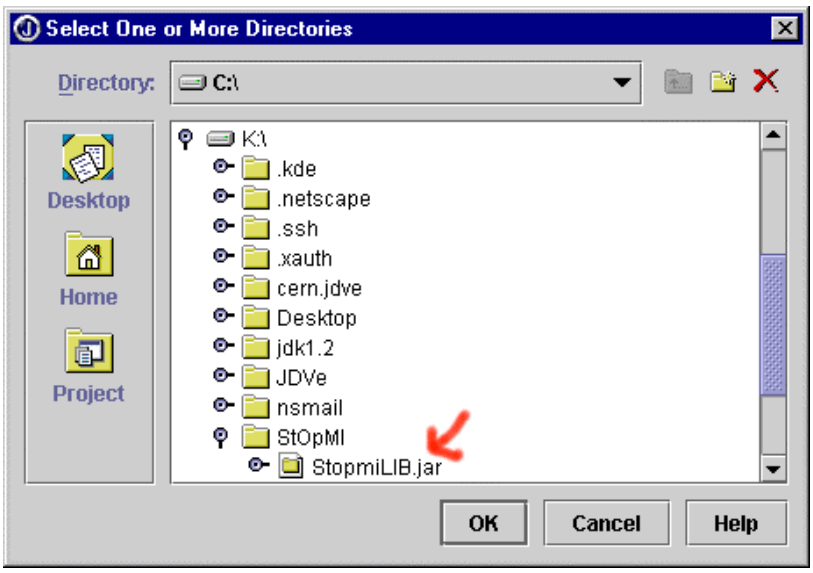


Fig 1.4

Press "*ADD*" again and select the StopmiExternalLIB.jar to include it in the *Stopmi* library.

9. Press OK again and you will see a window where the newly created library has been included ("*Stopmi*"). Select it, press ok and you will see that your library has been added to the "*Required libraries*" area.

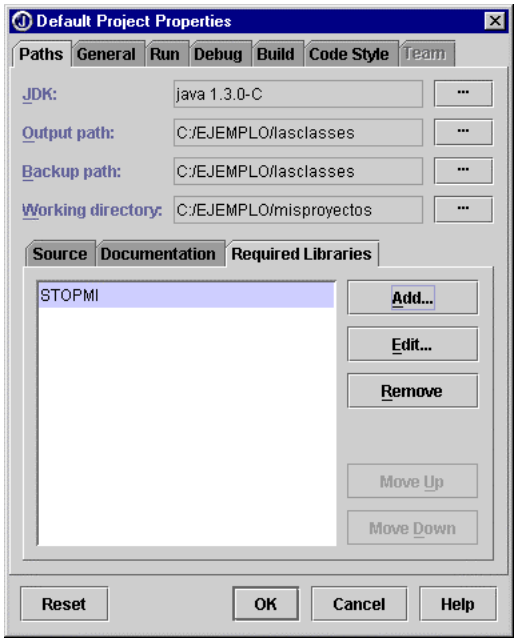


Fig. 1.5

ADDING STOPMI BEANS TO JBUILDER PALETTE.

Note: this action can be performed if you have included the Stopmi library in your development tool, as described in part1.

To include the stopmi beans on your palette, follow these steps:

1. Select the "*Tools*" option from the menu bar in Jbuilder and select "*Configure palette*" from the pulling submenu.

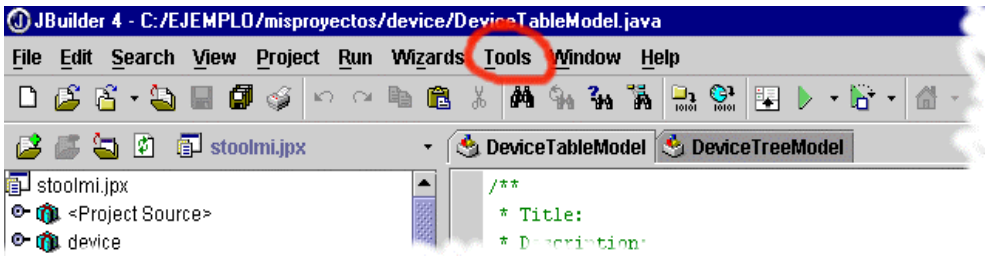


Fig. 2.1

2. A palette properties window appears with two tabs at the top: "*Pages*" and "*Add Components*". "*Pages*" refers to the way that beans are organized on the palette.

3. Create a new page to include your stopmi beans. With the "*Pages*" tab selected press "*Add*" button. Write the name of the new page you are creating.

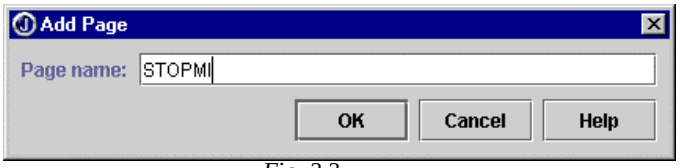


Fig. 2.2

4. Once you have created a new page, you will see that it appears on the list of existing pages. Select the page created and change the top tab to "*Add components*".

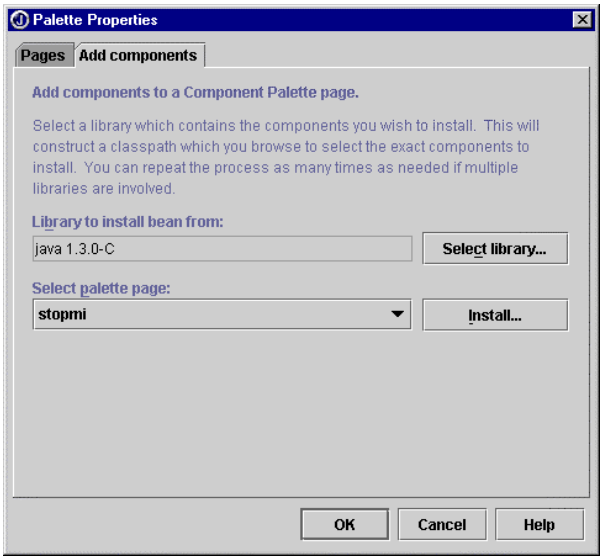


Fig. 2.3

5. Press the "*Select library*" button. Then select the *Stopmi library* you created with the first part of this guide.

6. Press OK to get back to the "*Palette properties"* window. Press the "*Install*" button and look for the *Cern* folder in the tree. Open the "*Stopmi* / *Beans*" folder and select all the OPBeans you find inside, except by the OPLabTypes and BeanInfo files (to select more than one bean at a time, hold the "*CTRL"* button when selecting the OPBeans).

7. Press ok and you will receive a message confirming that your beans have been added to your palette.

If you have followed the steps correctly, the OPBeans should now appear on your palette.

CREATING A PANEL

1. Design by hand your interface.

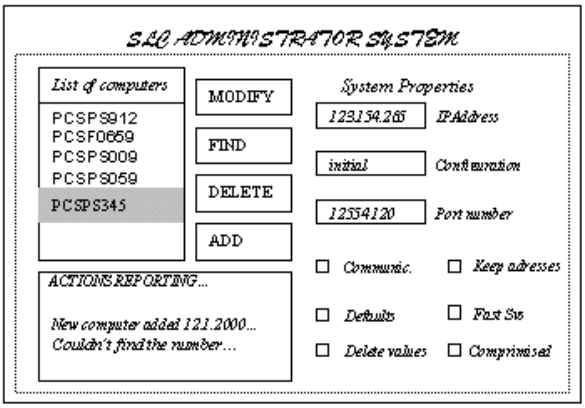


Fig. 3.1 Example of an interface

2. The interface you have created is a *Panel* in Java, and the elements you have placed in your interface like buttons, labels, textfields,..etc. are called "beans". In Appendix 3 you can find all the beans that Stopmi provides.

3. In Jbuilder open the file with the sufix "-mainPanel" that is been generated by the *wizard*. Your Panel appears now opened in Jbuilder as source code. Click the Design tab at the bottom to open the designer.

Fig. 3.2

This is what you see on the designer. The central square is your panel, you can enlarge it with your mouse clicking and dragging the point in the corner.

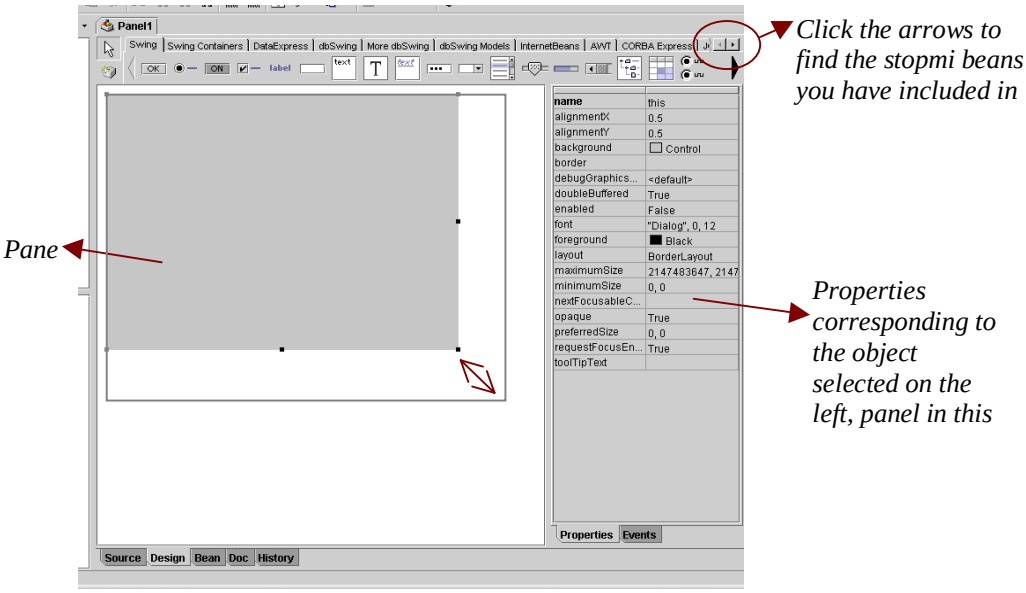
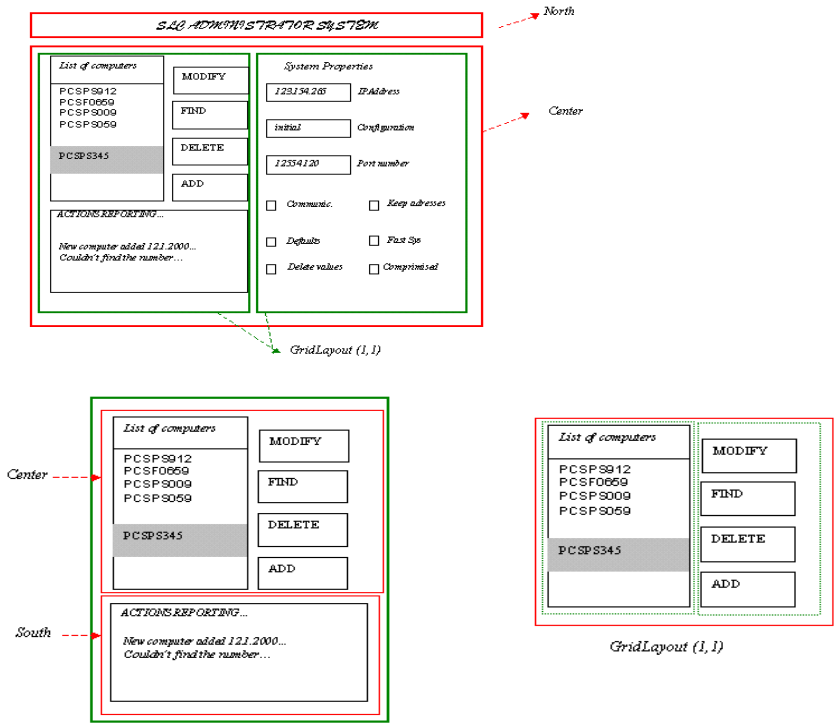


Fig. 3.3 Designer

3.1 The OPPanel you are working with has the ”layout” property set to “*XYLayout*”. This is a trick that will let you place the components wherever and with whatever size you like. **After** placing **all the components** you will change the layout again to a simpler one, so your panel description will not be so heavy and the components will adapt to a size change. For this layout conversion to be effective you have to place your beans in a way that the tool can convert their position to a defined layout. You can get this defining nested panels. (apendix 3 identifies the most common layouts you can use).

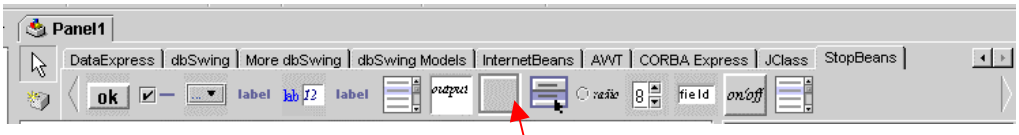
3.2 Identify the nested layouts your interface could be composed of.



3.6 Place the OPPanels: Select the Stopmi beans page (1), click on OPPanel bean (2) and then click to place it on the main area (3).

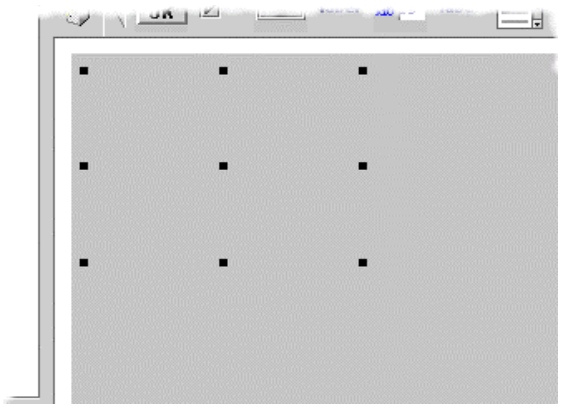
(1)

(2)



OPPanel

(3)



NOTE: On the properties panel on the right you can change the OPBorder property, so you can see the panel you have placed clearer.

Fig. 3.6

3.7 Adding successive OPPanels and resizing them you should get to this point:

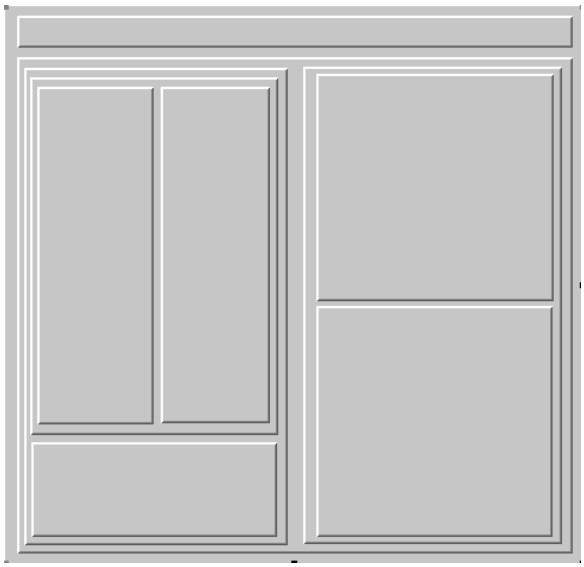
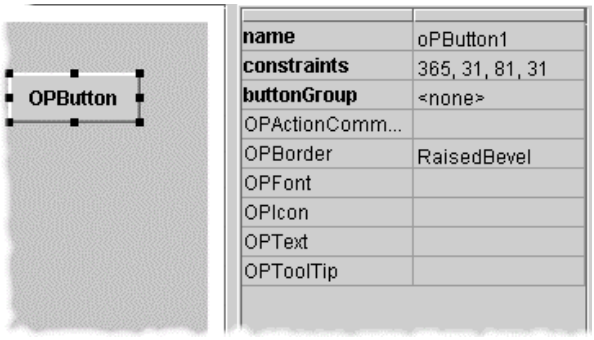


Fig 3.7. Nested Layouts.

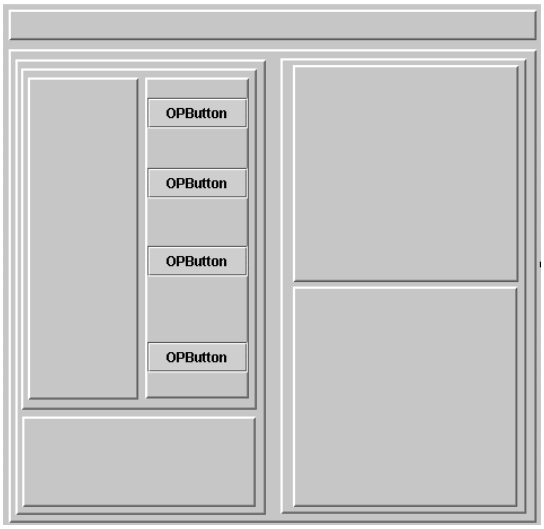
3.8 Start placing your OPBeans from the top layer to the bottom one. Follow the same 3 steps as before but selecting the corresponding beans.



Use the Properties Panel on the right to change labels, borders, fonts… Consult the appendix 4 for the meaning of the different properties.

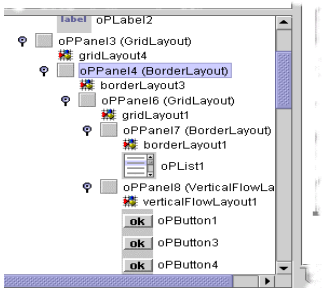
Fig. 3.8

3.9 Place the first four buttons and change its properties.



Save time!: By pressing the "ctrl" button while selecting components, you can change properties for all the components selected at the same time.

3.10 Continue placing components and remember that you can use the tree on your left to select components, copy and paste, delete, etc and see how your panels are organized.



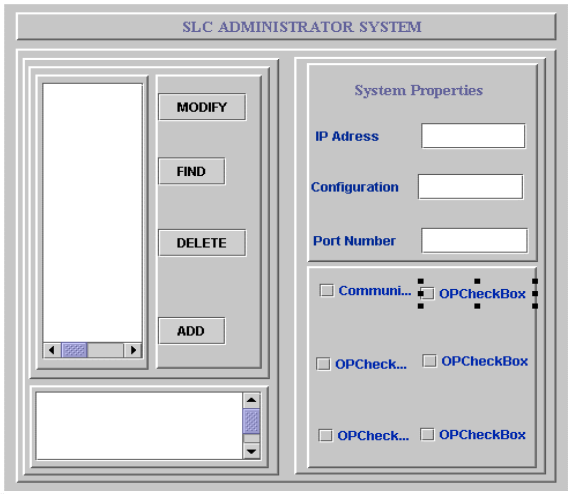
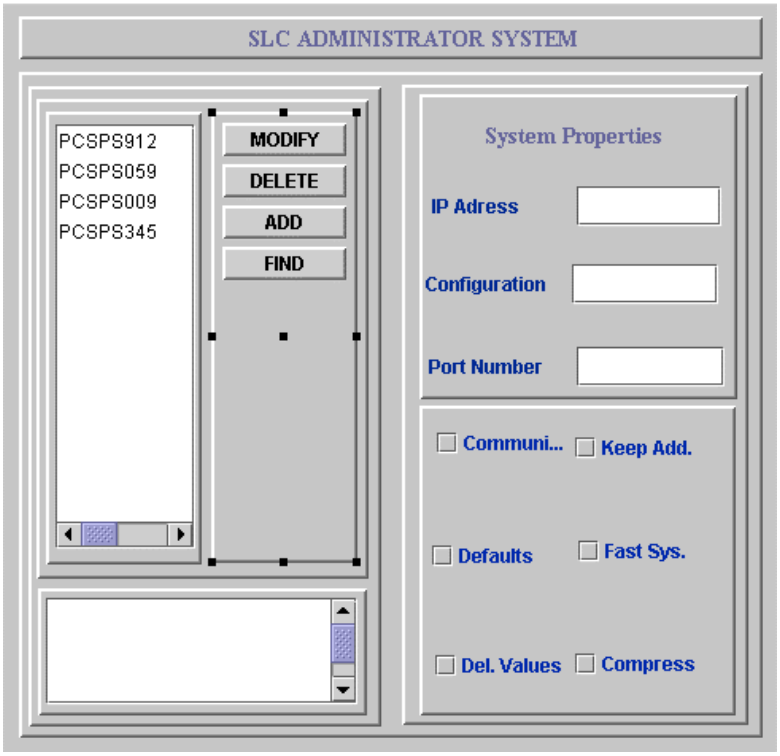


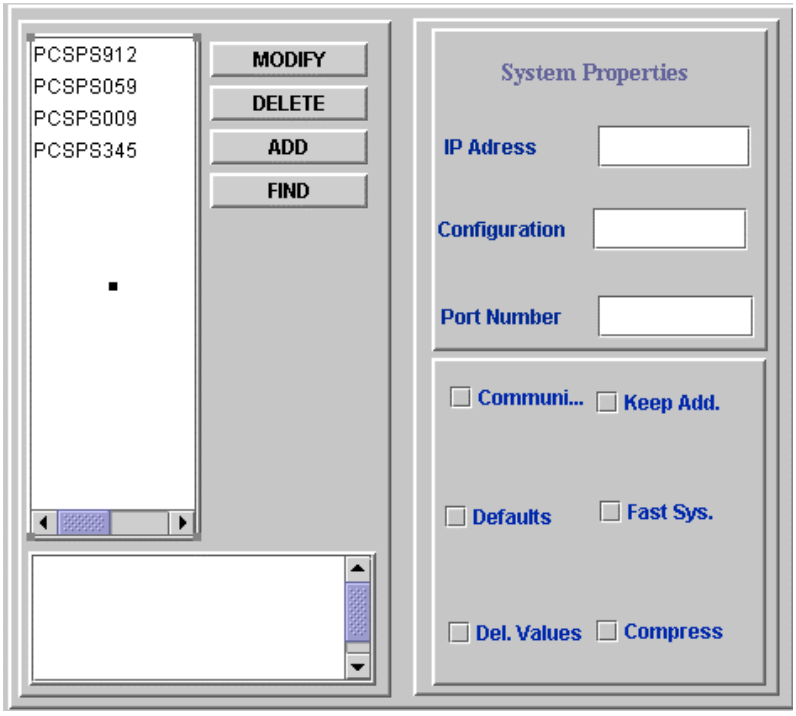
Fig. 3.10

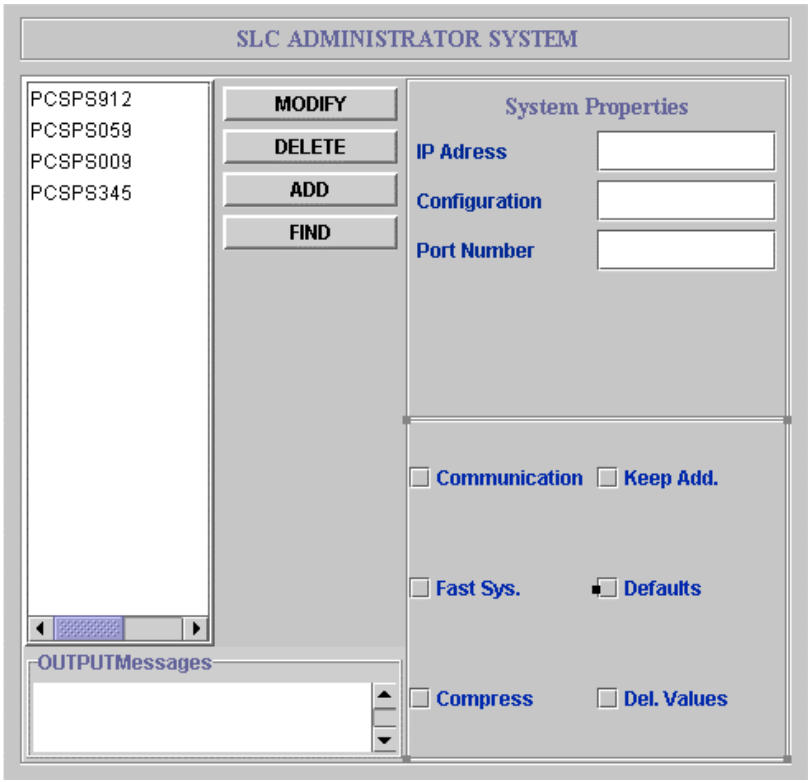
3.11 Once you have placed all the components it is time to change the panel layouts. Remember that you have worked with XYLayout and if you do not change this layout your panel definition will be long, heavy and will not resize when you run your application.

3.12 Select one of the top panels, e.g. the one containing the four buttons and change the Layout property to VerticalFlowLayout.



3.13 Continue converting the layouts that you have defined at the beginning of your design. If you have only one component over a panel, change to BorderLayout (you use this panel below the component because this is your first design, in the future you will not need it). When changing to GridLayout, you can specify number of columns and rows clicking on “Gridlayout” on the tree on your left. Figure 3.12 shows more layout changes and in figure 3.13 the borders of OPPanels have been modified too.





3.14 Performing the last change:

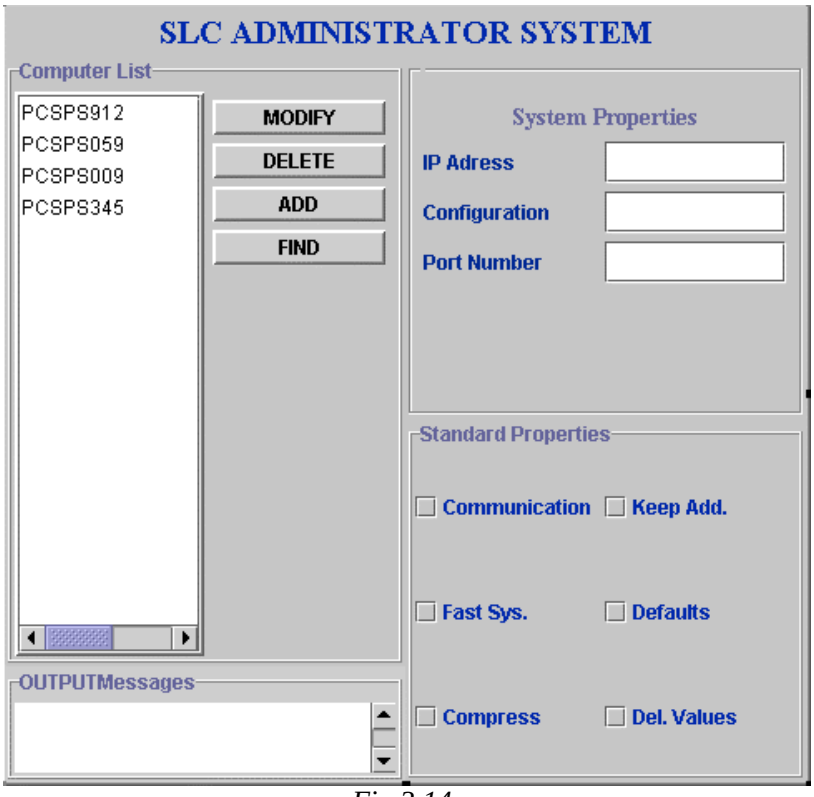


Fig 3.14.

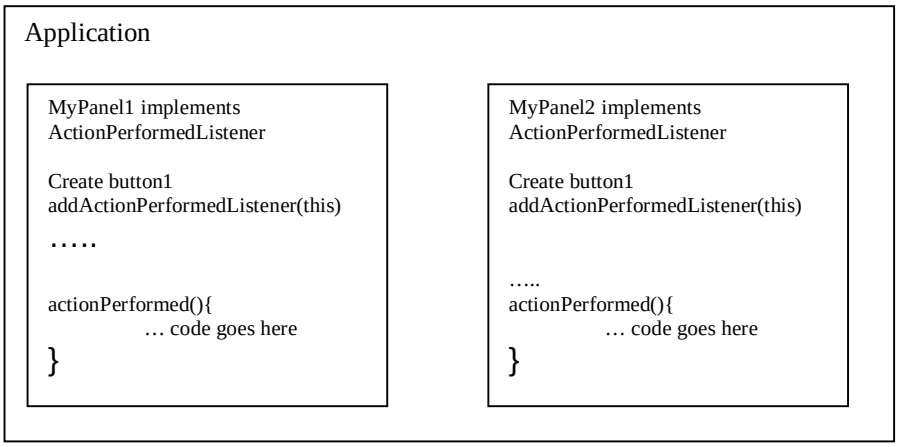
The OPController

1. What is it?

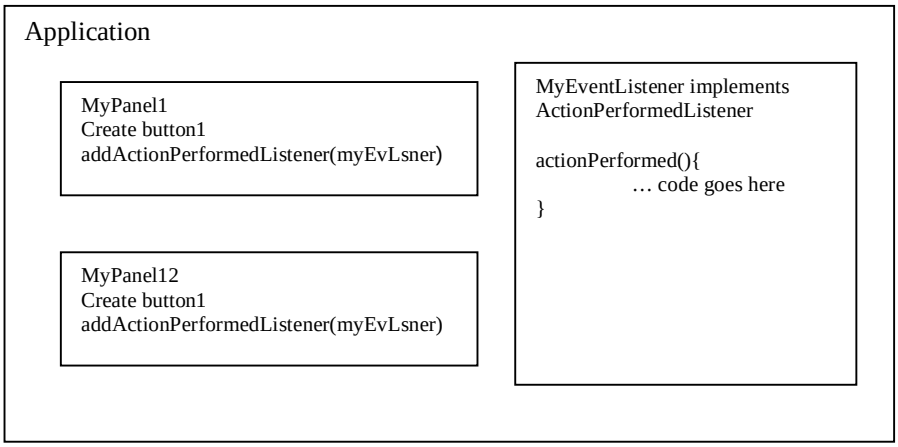
The OPController was conceived to attempt to give a *standard* way for developers in SL to organise their callbacks (listeners in Java terms). The aim was to effectively separate the visual part of the application (ie the panel with buttons, lists etc) from the code that would perform tasks in response 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:



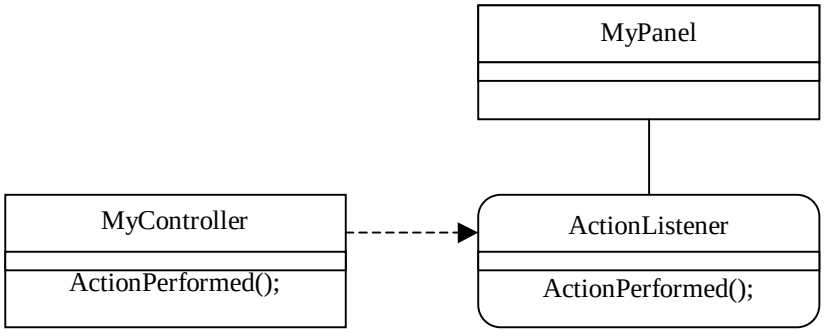
Use of this method, clearly separates the *actions* from the GUI, and makes the application's *action* code clearly defined and easily maintainable (ie to fix a bug, the developer doesn't need to wade through hundreds of lines of irrelevant code to find the problem!) It should be made clear at this point,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

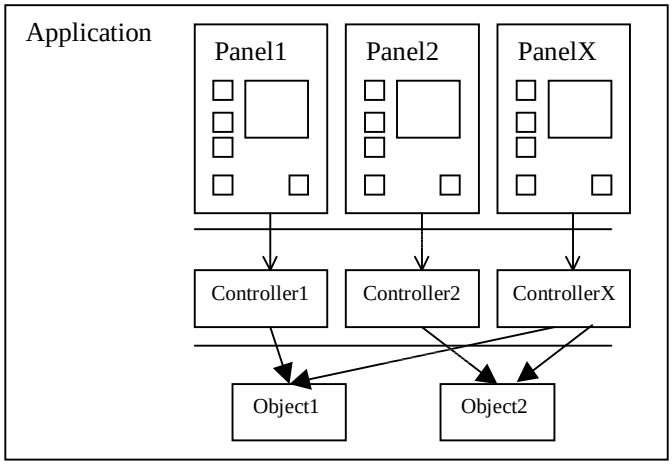
From anywhere within the controller, it is then possible to display the user information like this:

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

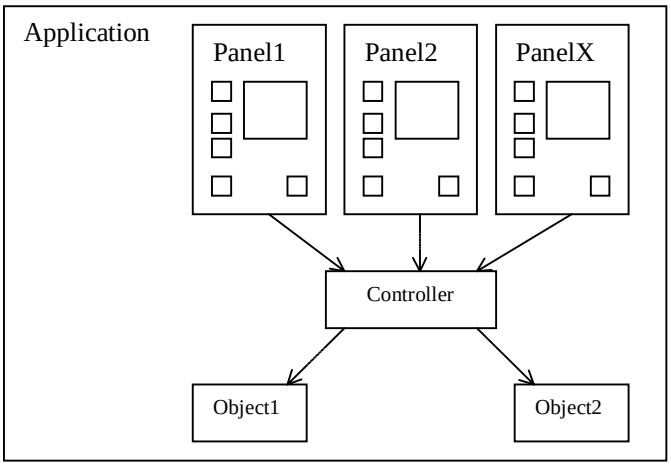
The more astute Java programmers, will be thinking "what if I have two objects of the same class?". The answer is that, at the moment, you can't! Therefore, if you want to pass an object reference to a controller, it needs to be a unique class. This in itself is not a bad thing, as it may encourage developers 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:

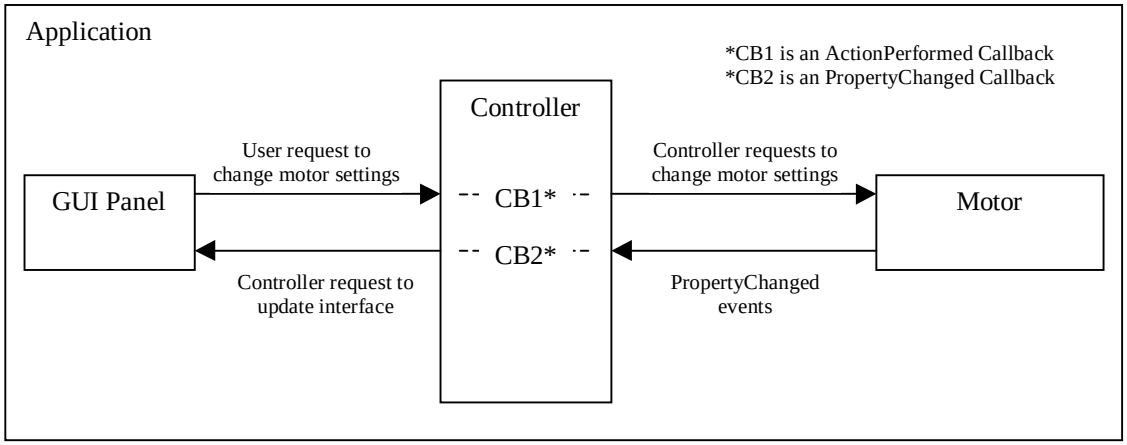
```
Public void actionPerformed(ActionEvent e){
   MainPanel myMainPanel = (MainPanel)getOPPanel(MainPanel.class);

   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

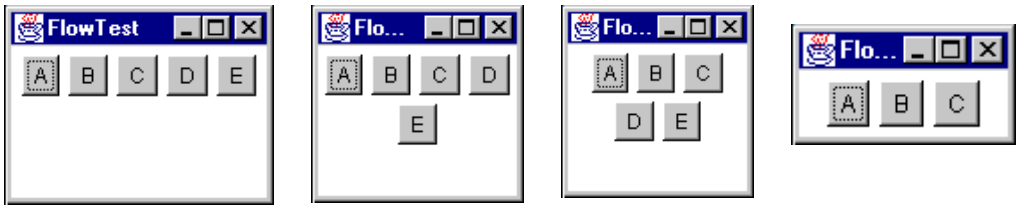
Examples: http://proj-stopmi.web.cern.ch/proj-stopmi/MotorExample.html
StopMI Wizard: \\pcslux10\production
StopMI Docs: http://proj-stopmi.web.cern.ch/proj-stopmi/sdocmi/Packages.html

Appendix 1. LAYOUTS

FlowLayout

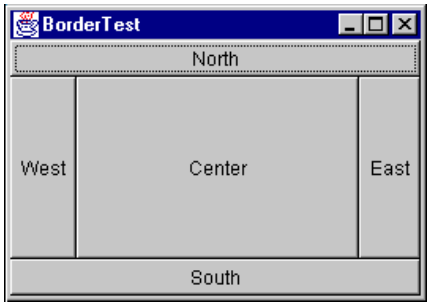
This is the simplest of the AWT layout managers. Its layout strategy is:

- lay out as many components as will fit horizontally within a container
- start a new row of components if more components exist
- if all components can not fit, they are not shown.



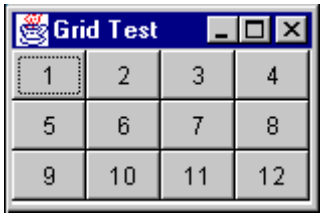
BorderLayout

BorderLayout is probably the most useful of the standard layout managers. It defines a layout scheme that maps its container into five logical sections:
The first thing going through your mind should be "but I will never have a GUI that looks like that!" Moreover, you are probably correct. However, the secret is in mastering its nesting capabilities, and using two or three of the logical sections. (It's very rare that you'll actually use more than three of the positions in a container at once).



GridLayout

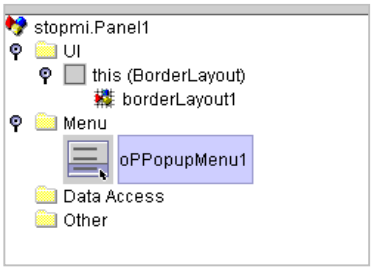
GridLayout lays out its components in a grid. Each component is given the same size and is positioned left-to-right, top-to-bottom. When specifying a GridLayout, there are two main parameters: rows and columns.



Appendix 2. How to create popupmenus.

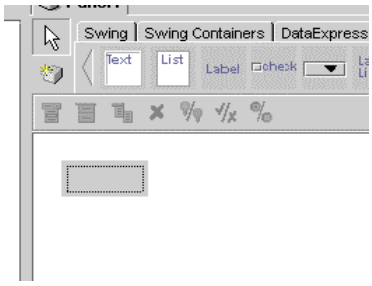
PopUpMenus are small windows that appear when you click the right button of your mouse on a component. You can create these menus when designing your panel with Jbuilder:

1. Click on OPPopUpMenu on the palette and paste it anywhere on your panel.
2. You will see that in the *tree* facility of Jbuilder the "*Menu*" folder is open and your *Popupmenu* has been added:

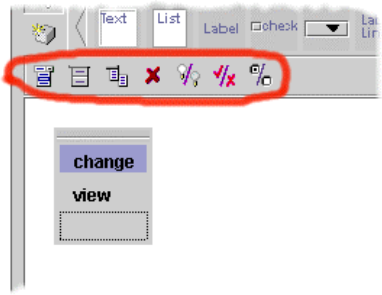


3. Click twice on the new *oPPopupMenu1* placed on the above mentioned tree.

4. Your Panel is not shown now and a new visual designer appears. This is the designer used by Jbuilder to create *PopupMenus*.



5. Click twice on the small square to start creating your menu. You have a small palette to add new items, separators, submenus…



6. Once you have finished creating your OPPopUpMenu, click on one of the OPBeans from the *tree* in Jbuilder tool to see your main panel again.
7. The popupmenu you have just created is part of your Panel, so you can have a reference to it as to the other variables on the Panel.