

Abstract Interface to Prototype Implementation with the WindowBuilder Plug-in

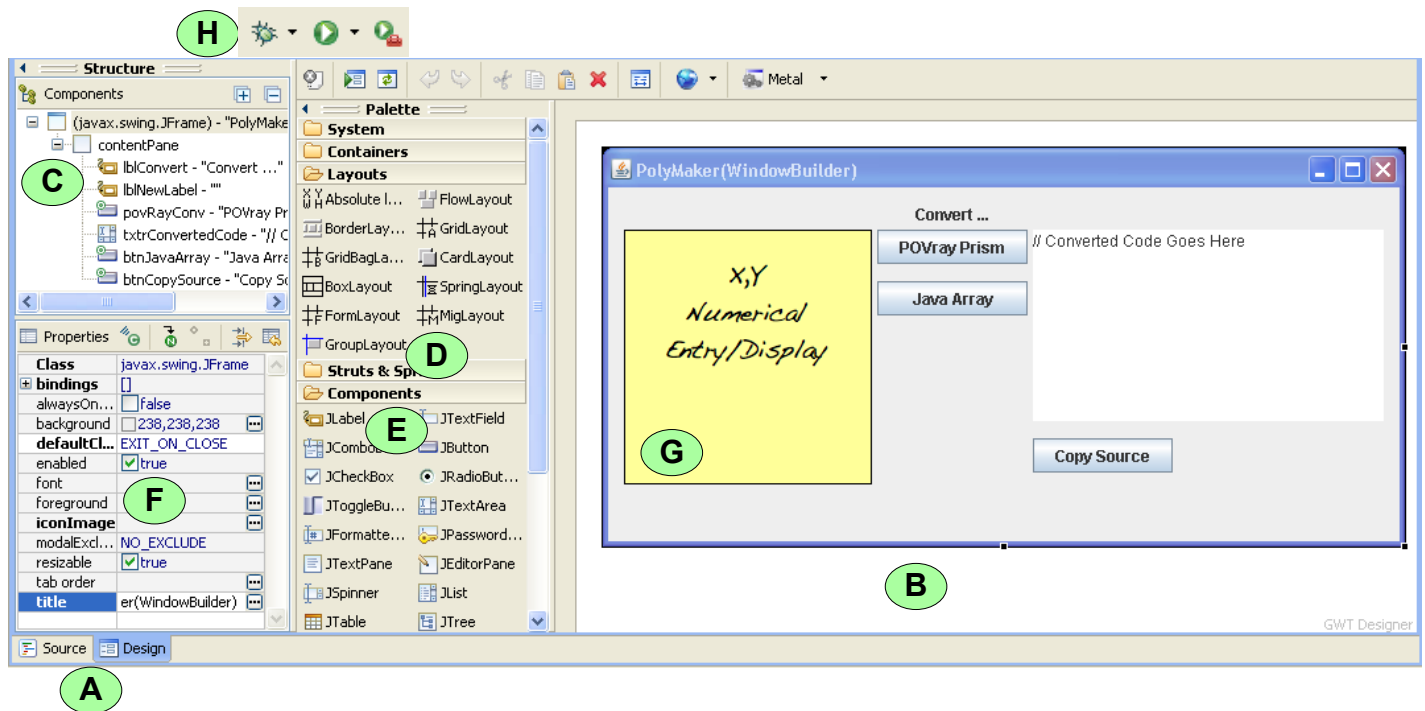


Figure 1: Screen shot of WindowBuilder Eclipse Plug-in constructing a version of PolygonMaker

Part 1: Introduction

An abstract interface is useful for roughing out the basic capabilities of what an interaction space should do. This can easily be done using paper and post-it notes (different colors might be used to represent different types of interaction). After some more refinement of your ideas, decisions must be made about how the abstract components will actually be implemented. Screen sketches and paper prototypes are often useful at this stage. Visual Interface development tools may also be useful as the system moves towards development.

Here we show how the WindowBuilder plug-in interface builder for Eclipse can be used to construct a prototype implementation. The example application, PolygonMaker, is designed to allow users to enter point values which are then converted into source code for different applications that construct polygons. The interaction area are abstractly represented on paper with post-it notes but here using images placed on JLabels. As work progresses, we replace the abstractions shown with images, to specific interface widgets.

1. Create a new Eclipse Project

a. **New --> Java Project**; Name your project *PolyMaker*. Make sure your project will be using Java 6. You'll be able to see your Eclipse projects and their files in Eclipse's Package Explorer, usually on the left of your Eclipse window. Figure 2 shows the file structure your project should have after completing step b below.

b. Right menu on the *src* folder and make a package *polyMaker* for the java source code. (**New --> Package**). Make another package under the *src* folder and name it *resources*. (**New --> Package**). This folder is for images used to represent the abstract interface.

Copy these images from the web and place them in the *resources* folder.

<http://www.cs.siu.edu/~wainer/nbMakingGUI/abstract2SimpleImpl/xyEntryMockup.png>

<http://www.cs.siu.edu/~wainer/nbMakingGUI/abstract2SimpleImpl/convTrigMockup.png>

<http://www.cs.siu.edu/~wainer/nbMakingGUI/abstract2SimpleImpl/convDataMockup.png>

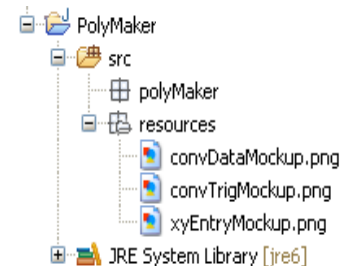


Figure 2: Starting Directory Structure

2. Begin the Prototype by Creating its JFrame and Readying the contentPane

- Right menu on the polyMaker package. **New** --> **Other ...**. A dialog box will come up allowing you to navigate among many items. Follow the path: *WindowBuilder*; *Swing Designer*; *JFrame*. Name this new class, *PolyMaker* and **Finish**. Once your choice has been completed and the dialog box dismissed, the class is created and some source code is generated. A window showing the source code should appear. You should see that this new class extends *JFrame*; it represents the application's top level window.
- Visualize and build the abstract prototype by switching to "Design Mode". Click the **"Design"** tab near the bottom left of the Eclipse window. (Fig. 1 – callout A). Using these tabs you can switch back and forth between working with the visual designer and the source code. You should see an empty *JFrame* in the design area (callout B in Fig. 1). Looking at the Components in the Structure area (callout C in Fig. 1), you see there is a *contentPane* which will hold all the items which appear in the *JFrame*.
- Visual containers like the *contentPane* can organize items within them in many ways. Swing manages this using objects called Layouts. Different Layouts can be used depending upon the needs and preferences of the designers. Layouts use algorithms to control how the components are sized and positioned as they and the container itself are modified. A Layout good for interactive visual design, is not necessarily good to have to maintain manually. To begin with, we'll choose a Layout that is easy to work with interactively in Design mode.
With the *contentPane* selected in the Components/Structure area (callout C in Fig. 1), look in the Properties pane (callout F in Fig. 1) for the **Layout** property. This shows the current Layout and to the right is a black triangle indicating a menu of alternatives. Use this menu to change the layout to **"GroupLayout"**. This layout manager is specifically designed to be used with visual designers. Layout managers are also indicated in the palette (callout D in Fig. 1).

3. Add Abstract Components to the JFrame's contentPane

- Find the *JLabel* Component in the palette (callout E in Fig. 1). Click it and then click again within the frame in the designer area (the cursor appears as a green plus to show it is over an appropriate target). This is the label we'll use to represent point entry (callout G in Fig. 1).
- With the label selected, look in its properties (callout F in Fig. 1) for **"icon"** and to the right of that **"..."**. Click the **"..."** to open the Image Chooser and select the **"Classpath"** option at the top. You should be able to expand the *src* and *resource* folders to see the 3 images that were copied over in step 1b. Select the yellow one, *"xyEntryMockup.png"*. The image will show on the right. Once you click OK, you will see the image on the label in the design view.
- Within the label properties, there is a "text" property which currently has a value like *"New label"*. Change it to the empty string and hit enter. The label in the design view should just show the image now.
- Drag the label around until you are happy with its placement.
- Repeat this process to create 2 more labels to hold the other 2 images. You should have a result similar to Figure 3. Select the *JFrame* (rather than *contentPane*) and resize it if you need a larger window.

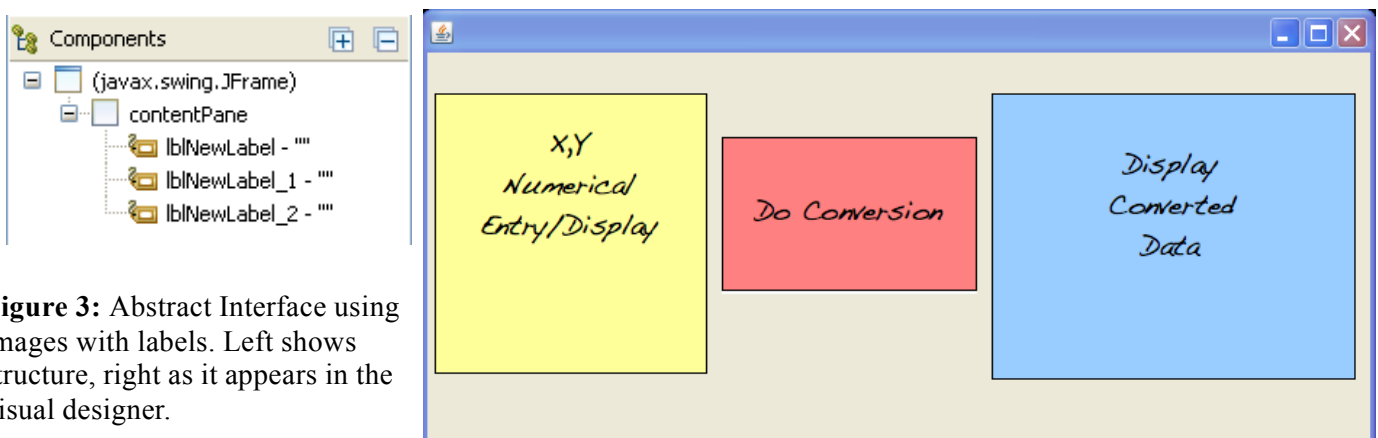


Figure 3: Abstract Interface using images with labels. Left shows structure, right as it appears in the visual designer.

4. Run your Code.

- With the *PolyMaker.java* class created in Step 2a selected, right menu and select **"Run As"**. Use the **"Java Application"** option to launch your code. Afterwards, there is a green circle with a triangle icon near the top of the Eclipse window which launches the last run target. (callout H in Fig. 1)

Part 2: Converting the Abstract components to Actual Widgets

In part 1 the WindowBuilder Interface builder plug-in was used to create an interface based upon an abstract prototype. Labels were used to display what might have been written on post-it notes to describe the abstract interaction components. As the design firms up, the abstract interface components will be replaced with actual widgets. These steps continue from where part 1 left off. It is assumed that your project is open in Eclipse. The JFrame class that is being edited (PolyMaker) should be open with WindowBuilder (use the “*Show with ...*” option if needed).

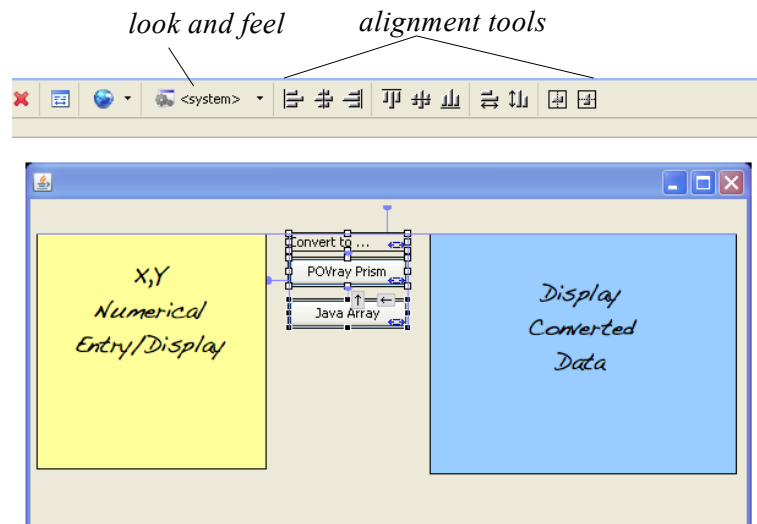
1: Replace the "Do Conversion" image label with buttons and a label

- Create a new label and place it above the “*Do Conversion*” image label. Change the text of this label so it reads “*Convert to ...*”. When a new label is created, its text is automatically selected so you can just type in the new value. Of course, you can always set the text in properties.
- Delete the “*Do Conversion*” label. Things may move around but can be re-adjusted by dragging. Don't worry too much about it now as things can better be adjusted after other items are added.
- Create a button and place it under the label. Like labels, just after creation, the text that will appear on the button is automatically selected and can be directly modified. Make it read “*POVray Prism*”.
- Create another button and place it under the “*POVray Prism*” button. Make it say “*POVray Prism*”.
- Straighten things up a bit. You might notice that the buttons may not be the same size or in some awkward alignment. Select all the 3 of the items created in steps a, c & d. Observe the additional alignment tools that become available as shown in Figure 4. Configure your interface to look like that in Figure 4.
- Run your code (refer to Part 1 step 4). If the controls don't quite look the same, you may be running under a different LaF (look and feel) than what you designed for. Try changing the look and feel (see Figure 4) of the Design view to match what you saw and fine tune if needed.

Figure 4: The “*Do Conversion*” image label has been removed and replaced with actual Swing components.

Selecting multiple items enables various alignment tools in the toolbar above the design area.

Java supports multiple looks and feels and their effects can also viewed as you construct an interface.



2: Replace the "Converted Data" image label with a text area

- Delete the “*Converted Data*” label.
- Add a JTextArea since the generated code will be several lines. This will get tricky since it may initially be sized very small. In its properties, select the “*Constraints ...*” option (near the top). A dialog box will pop-up allowing you to set a width and height. Try a width of 250 and a height of 150. Set its text to something like: “*// Converted source goes here*”.
- As components change size and move around, other components may also move unexpectedly as the layout manager tries to adjust. If you get stuck, there is an “undo” under the Edit menu, or you can just delete the textArea and try again. If you get a parsing error, try to reparse to return to where you were just before the error.
- Save and try running your design.

Part 3: Making the Interface Respond to the User

Visually we have replaced abstract components with real interface widgets which give the appearance of a working application, however pressing these buttons doesn't seem to make anything happen. A working application uses events generated by user interactions to carry out commands. We continue by adding event handling to the prototype.

1: Refactor to Rename Items and Hook up Buttons

There are some interface components which will need to be referenced by other parts of the program. At least those items should be renamed to make it easier to understand what they represent. Renaming is considered a form of refactoring – a reorganization of the code which doesn't change its function but makes it easier to understand and maintain. Refactoring is commonly used in agile development since it is a way to continuously maintain the design in the absence of a detailed upfront design.

a. Check the names of the 2 buttons and the text area to make sure they have meaningful variable names. WindowBuilder may have already named them based upon the text that you entered. You can rename using an object's properties or pop-up menus from other views of the object as shown in Figure 5. Eclipse also has a “Refactoring” menu that has a rename option. Subsequent steps will refer to the names as shown in Figure 5 when needed.

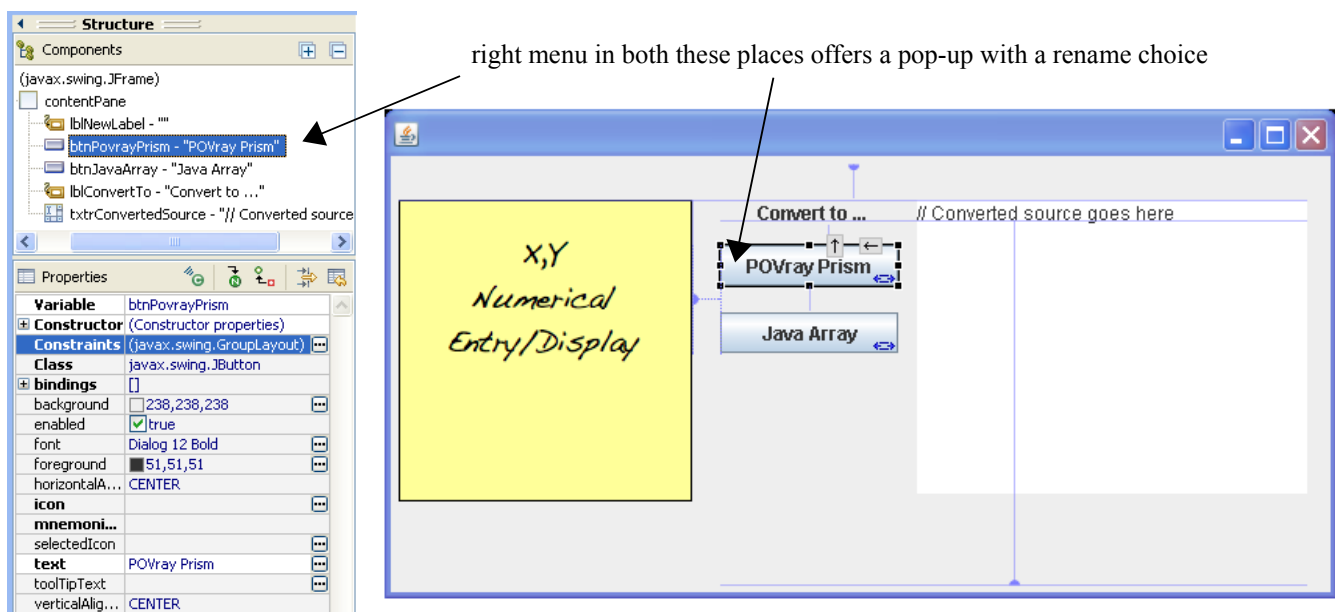


Figure 5: Assuring that button and text area objects are using meaningful variable names.

b. Select the text area, and using the right menu pop-up, select the “*Expose component ... option*”. Let it generate a method to obtain this component – it also will generate an object variable for it rather than keeping it local.

c. Double-clicking a button in the design view will switch to the source code for that button's *actionPerformed* method. Double-click the “*POVray Prism*” button. An anonymous innerclass is generated as that button's *ActionListener* and a stub *actionPerformed* method is created (if it doesn't already exist). To make the button do something, you fill in code for the *actionPerformed* method. (This is one style of event handling in Swing – WindowBuilder has preferences to adjust this.)

d. Add code to simulate generating POVray source code as shown in figure 6. This assumes that you have named the text area as in Figure 5.

e. Save and Run your application. Press the “*POVray Prism*” button and new text should be sent to the text area.

f. Repeat in a similar way for the “*Java Array*” button.

g. Add a “*Copy All*” button with code like “*ta.selectAll(); ta.copy();*” to copy text area contents to the clipboard.

```
 JButton btnPovrayPrism = new JButton("POVray Prism");
  btnPovrayPrism.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
      txrConvertedSource.setText(
        "// generated POVray source");
    }
  });
```

Figure 6: Customizing the actionPerformed method.