Abstract Interface to Prototype Implementation
with Eclipse & the WindowBuilder Plug-in

Part 2: Using Widgets and Model Object(s)

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 interface 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 that is being edited is from the PolygonMakerApp class which should be open with WindowBuilder (use the “Show with ...” option if needed).

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

a. Create a new label and place it above the existing “Do Conversion” label representing the abstraction of the conversion interaction space. Change the text of the new 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.
b. Delete the old “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.
c. 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”.
d. Create another button and place it below the “POVray Prism” button. Make it say “Java Array”.
e. 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 1. Configure your interface to look like that in Figure 1.
f. 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 1) of the Design view to match what you saw and fine tune if needed.

![Figure 1: The “Do Conversion” image label has been removed and replaced with actual Swing components.](image)

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

a. Delete the “Converted Data” label. (The one showing the source code image).
b. 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

Java supports multiple looks and feels and their effects can also be viewed as the interface is constructed.

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

Part 2: Using Widgets & Model Object(s)
width and height. Try a width of 250 and a height of 150. Set it's text to something like: “// Converted source goes here”.

c. 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 JTextArea and try again. If you get a parsing error, try to reparse to return to where you were just before the error.

d. Save and try running your design.

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.

3: Refactor to Rename Items and then 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. The process of refactoring can actually begin with the screen sketches as the team works out which objects need naming and what are the most appropriate names.

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 2. Eclipse also has a “Refactoring” menu that has a rename option. Subsequent steps will refer to the names as shown in Figure 2 when needed.

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 3:Left. This assumes that you have named the text area as in Figure 2.

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. (Make it's text clearly show it is for a Java Array.)

g. Add a “Copy All” button with code like that shown in Figure 3:Right to copy the text area contents to the clipboard.

Try running your code. Do a conversion followed by a “Copy All”. Try pasting to ensure it all works.

Part 2: Using Widgets & Model
4: Refactor to Prepare to Incorporate the Model

Keep in mind that the code we are building here is expected to evolve. Look at the code that has been generated for PolygonMakerApp. Sometimes may seem awkward, if so that code should be considered for possible refactoring to improve its design and readability. Generated code can easily create long methods with hard to follow names and logic. Aside from that, the code as it is now, still has no connection to the model objects. To get ready to incorporate the model, we'll do another renaming refactoring to clarify that the current “initialize” method is actually just building the GUI. We'll need to add a custom initialize to set up the model.

a. Refactor the method “initialize” to rename it to buildGUI. Select “initialize”, then right-menu. In the “Refactor” submenu select “Rename” and type the new name “buildGUI”. This is a simple example of refactoring: no features were changed by this; it still looks the same when run now as it did before. It just gets the code better arranged for when we do make real changes.

Another reorganization can be applied to how events are handled. We already know that the conversion operations are basically the same except that a different PolygonConverter will take care of each different type. Since they are basically the same, it will probably work out well to put them in one place. Instead of having each converter button generate its own ActionListener, we'll let them all refer to the parent object as the Listener. WindowBuilder can generate code in this style by changing its preferences. Open the Eclipse menu Preferences -> WindowBuilder -> Swing -> Code Generation -> Event Handlers. Select the option “Implement Listener Interface in Parent Class”. New event handlers will be generated with this style but existing code won't change.

b. In Design view, select the POVray button. With the right menu, select the “Exposé component...” option. This allow access to the button after it has been created. Now double-click the button and a new event handler will be created. Add the code from Figure 4 to the empty actionPerformed method. Since multiple events are handled here, the if is finding out which button sent the event. Run your code to confirm it still works.

c. Follow the same process for the Java Array button. (Expose, delete handler, create new handler). The handler will be the same actionPerformed method so you'll need to add an else if branch to match the event source against that button.

5: Including the Model Objects

a. At least to start, our model will use one PolygonModel object and one PolygonConverter type. So near the beginning of the file, in the declarations area, add the code to the right.
b. Create the new initialize method to set up the model objects as given in Figure 5 Left. Make sure to call initialize from the constructor (put it just before buildGUI()). By supplying data for a test polygon we can demo parts of our application while the data entry details are still being worked out. (This also suggests a new polygon constructor.)

c. To enable the conversion function to work, go back to the actionPerformed method and simply plugin the model objects in the if branch for the POVRay conversion as follows in Figure 5 Right. Try your code out. You should get a real conversion on the polygon data. Your “Copy All” button should still work as well.

```java
public void initialize() {
    polygon = new PolygonModel();
    double[][] polygonData = { // test data
        {0,0}, {100,0}, {100,100};
    for (double[] xy : polygonData)
        polygon.addPoint(xy[0], xy[1]);
    povRayConv = new POVRayConverter();
}
```

---

6: Cleaning up the Interface

a. The converted code could easily overflow the source display. Let's add scroll bars. Find where the JTextArea is created and immediately after it create a JScrollPane with your text area in the constructor.

```java
taxtrConvertedSource = new JTextArea();
taxtrConvertedSource.setText("// Converted source goes here");
JScrollPane scrollableTextArea = new JScrollPane(txtrConvertedSource);
```

Add the scrollableTextArea object to the window instead of the text area. Look for addComponent method(s) which are adding the text area item and replace with the scroll pane (there may be more than one depending upon your layout). Since the scroll pane already has the text area inside of it, the text will still be in the window. Try running your code (resize if needed to make the scroll bars appear). Afterwards switch back to design mode and you will see the scroll pane in the structure view.

b. When resizing you may have found things moving around and changing sizes in unwanted ways. Experiment with glue and struts to see if you can improve that behavior ([samples](#)). You might also try to use panes to group widgets and incorporate other layout managers. Sometimes a “Surround with ... “ item appears in the context menu; it can help when trying to wrap items with a JPanel.

---

Figure 5: Left: Initialize the model objects. Note the polygon is supplied with test data. In the actual application the user would need to specify this data. Right: Body of actionPerformed if for POVRay converter button.

Figure 6: The same interface reworked to use nested panels and a variety of layout managers. This helps to tune resize behavior (along with struts and glue).