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

Part 3: Popups, Menus, Actions, More Refactoring & Simulating State Progression

Parts 1 & Part 2 built up an interface that put physical components on the screen based upon moving from an abstract design. In this part we add a pop-up message (using JOptionPane) and a menu bar with a simple File menu. An example of a Swing Action is given to show how multiple ways of doing the same user action can be coordinated in one place. For instance, a “Clear” command might be issued from a button as well as a menu item. Sometimes the command should be enabled or disabled. This also provides examples of refactoring as the code evolves. Finally, the “Point Entry/Display” mock-up label is morphed to a button so it can be used to fake progress through program state. This is a technique to let the coded prototype demonstrate some features before all of its dependencies are implemented.

Figure 1: As PolygonMaker evolves, more features are added.

1. Clear button
2. Pop-up warning
3. Menu w/accelerator
4. Label is now a button. (used to simulate interaction)
5. Clear action is dimmed if data is already cleared.

1: Add a Clear Feature with Pop-up Warning

Clear will be used to clear out a polygon's point data. Since all of its existing data will be destroyed, we want to warn the user that data will be lost. A JOptionPane is used to create a simple warning dialog that pops up and gives the user a chance to cancel the clear. Here are the basic steps.

a. Add a “Clear” button below the point entry/display area.
b. Change the Eclipse -> Preferences->WindowBuilder-> Swing-> Code Generation-> Event Handlers to “Create inner class”. Double-click your “Clear” button to generate the event handler. Inside the actionPerformed method add code similar to that given in Figure 2. Check against Figure 1 to correlate the parameters with what appears on the screen. Notice that the JOptionPane allows for a user response. Based upon that, the clear is completed or canceled.

Object[] options = { "No", "Yes" };
int userReply = JOptionPane.showOptionDialog(frame, // create pop and collect user's response
"Do you really want to clear all the polygon data?", "PolygonMaker: Warning", JOptionPane.DEFAULT_OPTION, 
JOptionPane.WARNING_MESSAGE, null, options, options[0]);

if (((String)(options[userReply])).equals("Yes")) {
    polygon.clear();
}

Figure 2: Handling a “Clear” request
2: Add a Simple File Menu

Most applications have menus. We'll create a simple one to get started with. WindowBuilder does have a visual way of letting you build up menus. An application's main menus are usually listed together across a JMenuBar. In design view there is a palette for Menus.

a. click and create a JMenuBar near the top of your frame.
b. add a JMenu to it. Name it “File”
c. add JMenuItem “Clear” and “Exit” to it. Put a JSeparator (from Components section) between them

Menu items also generate event handlers. Select the menu item, then right-menu, “Add Event Handler ...”, “action”. Add in the the code for the exit handler first. Figure 3. It sends a closing message as if from the application's main window. Once that window is closed, the application exists (see more about WindowListeners). Test run your code.

d. The code for handling “Clear” already exists since it was done previously for the button. Instead of copying and pasting all of that code, duplication can be reduced by doing an “Extract Method” refactoring first.
e. Examine the source code for the “Clear” event handler. Imagine you could replace the several lines of code inside of the actionPerformed method with a call to another method. Eclipse can do this for you automatically. Select the lines that you would like to replace, then right-menu, “Refactor”->”Extract Method...”. You'll be given a chance to name the method that will encompass all those lines of code. Choose something like: clearWithConfirmation. Eclipse will take care of defining the new method and moving the original code their while replacing the selection with a call to the new routine. Now the button and the menu item can call the same code and any changes can be done in one place.
f. Generate the event handler for the “Clear” menu item (if you haven't already). Update its actionPerformed method so it calls your newly extracted clearWithConfirmation method. Try out your code.

3: Refactoring “Clear” to use an Action

Some functions don't make sense all the time. If the data has just been cleared, it doesn't make sense do to it again until there is something that has been added. We now prepare to implement this with a logical constraint (disable the clear function unless there are points to clear). The tricky part is that both the button and menu item (including accelerator key) should be enabled and disabled simultaneously. Swing provides Actions as a high-level way of coordinating and working with events. This refactoring replaces the current event handling for “Clear” with an action.

a. Return to design view. Click on “New Action” in the Swing Actions palette. Drag it into your interface and place onto the “Clear” button. (Your button will change its name!)
b. Under the structure view, select the action and edit its properties. Give it a variable name like clearAction, a short description etc. as in Figure 4. Note that the accelerator key given is “N” since clear acts almost like creating a new object. If you can't get the properties view to display go into your code, near the end you should see a definition for a SwingAction class with something like: putValue(NAME, "SwingAction"); Change the name to “Clear”. Hopefully upon returning to view the properties dialog again, everything should display correctly.
c. Put the clearWithConfirmation method extracted in part 2 into the action's actionPerformed method.
d. Then edit the source code creating the “Clear” button to just:

    btnClear = new JButton(clearAction);

Figure 3: Handler to Exit the program.
The variable mainWin may be named differently in your code. Most likely frame if you haven't given your main window a title or otherwise renamed it.

Figure 4: New Actions can be created and their properties edited in design view.
The `clearAction` properties will take care of the button name, its actionListener, accelerator etc. Remove the button's old `actionListener` and try your code.

e. Do the same for the “Clear” menu item. (At creation: `JMenuItem mntmClear = new JMenuItem(clearAction);` then remove the old `actionListener` and try your code.) Try the accelerator key too.

**4: Enabling/Disabling – Simulating Progressing Through States**

It is very easy to enable or disable a control. Since our prototype starts with a test polygon already initialized, it makes sense to have “Clear” enabled. Once “Clear” has been executed, we know all the points have been removed, so “Clear” should be disabled. Since the command is always executed through the action, modifying its state is all we need to do.

a. In the method `clearWithConfirmation`, add the following to occur just after the polygon is cleared.

```java
clearAction.setEnabled(false);
```

Now run and test your code again. Of course, once the “Clear” command is disabled, there is no way to turn it back on again.

The basic idea of simulating progressing through states is to let the prototype advance to different states but instead of expecting all of the interface functionality, the prototype fills in some parts automatically. Since the details of point input have not been defined, we just simulate data entry. Moving to the next state should be triggered by some easy to do interaction. A suggestion is to change the abstract interface for data entry/display now a label into a JButton. The program could start with an empty polygon (change initialize so polygon starts as empty) but everytime the “simulate data entry/display” button is clicked it increments a count representing a (fake) state transition. Based upon the state count other actions might be taken (for instance, adding a point from a predefined array to the polygon model.) If done correctly, this would re-enable the “Clear” function.

b. Select the “Data Entry/Display” label. Use the right menu and notice that there is a morph to option. This allows one type of component to be changed into another. Change the label to a JButton.

c. To make the “Data Entry/Display” button do something, it will need to be hooked up to an appropriate listener. As mentioned earlier, the `actionPerformed` method might increment a state variable, add a point to the polygon model etc. Of course, once a point has been added, the “Clear” action can be enabled again: `clearAction.setEnabled(true);`

Additional feedback/realism can be achieved by changing the button's appearance by using an ImageIcon.

Just as it was very easy to display an image on a label, it is also quite easy for a JButton.

**5: Some Other Things to Think About:**

At this point there are probably many areas that could use some refactoring (for instance, the buildGUI method is probably very long and could be shortened with “extract method”).

Save your work often and practice using Eclipse's history mechanism.

GUI design – upon start up are your controls consistent with the application state?
(For instance, if you started with an empty polygon (which seems logical), is the “Clear” action disabled?)

What should “Clear” do regarding the converted source displayed to the user?
The program is in a certain state whether you think about it or not – what is your design decision?
Is “Clear” even the right name? How do we know?

**How to Make Dialogs (a Java Tutorial)** - [http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html](http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html)