Object-Oriented Graphical Interface Design in Java

Samuel Flynn
3/27/09

Executive Summary:

Modern graphical user interfaces (GUI’s) can consist of hundreds of components, such as buttons, text fields, and other controls. This multitude of devices must be able to communicate with each other in order to respond to changing conditions. Traditional systems execute a single block of code whenever an event, such as a mouse click or keystroke, happens and determine how to change the interface accordingly. This is a poor way of going about managing hundreds of objects, as this single event handler can become very long, which will cause long response times to user events.

A more organized way of handing graphical user interface design is to use an object-oriented approach. Instead of using a single event manager to change each component, the components simply send signals, in the form of function calls, to each other. By distributing tasks to the components, it becomes easier to edit one component without affecting all of the others.

The composite pattern is particularly useful in GUI design, as it sets up a tree-like structure of components. Under the composite pattern, container components can hold and manage any number of child components. These child components, in turn, may contain other components, themselves. Once this hierarchy has been set up, a large number of components can be manipulated by calling a function on a container. The container will automatically respond by calling that same function on all components it contains. This allows commonly used functions to be used repeatedly by containers recursively calling them on their children.

By using an object-oriented composite pattern, management of many components becomes a matter of programming the components to properly call each other’s functions when an event happens. In this application note, we will discuss in detail how to configure these components in the Java programming language and how the composite pattern works.
Introduction: The Composite Pattern:

The composite pattern is a computer science design pattern that uses a common component class, from which all other parts derive. The component class needs to have functions that all components must be able to respond to, such as an enable/disable function.

The abstract component class can be divided into two other kinds of classes: a container and a leaf class. The container, and other classes that further derive from it, manage other components that are placed under its control, including other containers. Therefore, the container class must implement the functions needed to manage its children, such as an add-child, remove-child, and get-child function.

The leaf classes, on the other hand, do not need to manage any other components, nor do they necessarily need to implement any new functions not declared by the top-level component class. The specific function of any given leaf class depends entirely on what the user needs them for.

Here is an example class structure where the composite pattern is used to represent a comic book:

![Composite Comic Book Class Structure](image)

Figure 1: Composite Comic Book Class Structure
In this example, the book, page, and chapter classes are containers while pictures and words are the leaves. The composite pattern can then be used to put together an entire comic book under a large tree structure by creating containers and adding other containers and leaves to them.

Figure 2: Composition of a Comic Book Object

This example shows how the abstract components, containers, and leaves, once given concrete implementations, can be used to organize a large number of components with relatively simple code. For instance, say we wanted to tear page two of this comic book to shreds. Without this pattern, we would have to call tearToShreds() on picture three, text 4, and finally on the page itself. With the pattern, however, we can simply call tearToShreds() on the page. The page object will then automatically take care of shredding all of its child classes before destroying itself and returning. The process of deciding which components to manipulate has been simplified under this tree structure and is, therefore, much easier to set up and debug.
Applying the Composite Pattern to Interface Design:
Java is an object-oriented language with many built-in tools and classes useful to designing a graphical user interface (GUI). The Swing toolkit, for example, uses the composite class to allow for easy assembling of window and panel-based interfaces. In order to begin to assemble a GUI using swing, start with the root frame.

Creating a Window with JFrame:
The JFrame class creates the window the operating system will run. This is a special container in that it can hold any number of components, but it can only display a single container class at a time. This is referred to as the frame’s “Content Panel” and can be accessed by using the function getContentPane().

In order to start writing a GUI, create a main function that creates a new JFrame and obtains its content panel. There are also a few housekeeping functions we need to call before the frame will do anything:
- frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); - This causes the program to exit when the main window is closed.
- frame.setLayout(null); - This command disables the layout manager and allows components to be placed freely.
- contentPane.setPreferredSize(new Dimension(300,300)); - This resizes the useable area in the program window. This sets aside 300 by 300 pixels that components can be placed on.
- frame.pack(); - Pack allows the window to assemble its components and get itself ready to be launched.
- frame.setVisible(true); - This command displays the frame. This should always be the last line in the program.

In addition, the following packages must be imported. Write these lines at the very beginning of the program:
- import javax.swing.*;
- import javax.swing.border.*;
- import java.awt.*;
- import java.awt.event.*;
- import java.net.URL;
Once these lines have been entered, the code should look something like this:

```java
import javax.swing.*;
import java.awt.*;

class SampleGUI {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        Container contentPane = frame.getContentPane();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);
        contentPane.setPreferredSize(new Dimension(300, 300));
        frame.pack();
        frame.setVisible(true);
    }
}
```

Figure 3: Code for an Empty Frame

Nothing has been added to the content panel yet, so the window will not contain anything when it is run.

Figure 4: An Empty Frame
Adding Leaves to the Content Panel:

Unfortunately, this empty window is useless unless content is added to it. The simplest of content is the JLabel. The JLabel can display text and images to the user. Create a new JLabel with text “Hello, world!” This can be done either by passing “Hello, world!” into the constructor, or by calling the setText(“Hello, world!”) function on the label.

Once the JLabel has been created, it still needs to be added to the content panel. In order to properly add any component to a panel, the following steps need to be executed:

- `contentPane.add(label01);` - This adds the component to the content panel. Label01 is now a child component of contentPane.
- `label01.setBounds(10,15,100,20);` - SetBounds tells the program where this component will be displayed. The first two numbers say that the component’s upper-left hand corner will be 10 pixels from the left edge of the window, and 15 pixels below the top. The second two numbers specify that this label has a maximum size of 100 pixels wide, and 20 high.

The code should now look like this:

```java
import javax.swing.*;
import java.awt.*;

class SampleGUI
{
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        Container contentPane = frame.getContentPane();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);
        contentPane.setPreferredSize(new Dimension(300,300));

        JLabel label01 = new JLabel("Hello, world!");
        contentPane.add(label01);
        label01.setBounds(10,15,100,20);

        frame.pack();
        frame.setVisible(true);
    }
}
```

Figure 5: Hello World Code
Adding Buttons to Respond to User Input:

Unfortunately, a single text block is not a very useful GUI. In order to add some functionality to this interface, it needs some buttons. Adding buttons is very similar to adding JLabels. Start by creating and adding two JButton objects to the content panel. The text on the buttons can be set the same way as on the JLabels: either by passing it into the constructor, or by using the setText function. Remember that the buttons will not show up unless the add(button) function is called on the content panel and the setBounds function is called on the buttons themselves.

Now that the buttons have been created and added to the screen, it’s time to make them do work. Every JButton keeps track of a series of classes called ActionListeners. When a button is clicked, it calls the actionPerformed function on every ActionListeners it has added to its list. However, the actionPerformed function on ActionListener is unimplemented, so when a new ActionListener is created, its actionPerformed function must also be implemented. Add the following block of code to the main function:

```java
public void actionPerformed(ActionEvent e) {
    label01.setVisible(true);
}
```

Figure 7: How to Make an ActionListener
These two ActionListeners, when activated, hide and show the “Hello, world!” label created in the previous section. Unfortunately, as the code currently stands, this definition is illegal, as label01 can’t be referenced inside the in-line function definition. To solve this problem, move the initial declaration of label01 from inside the main function to outside the main function, but still inside the class. It is also important to add private and static modifiers to the declaration, as label01 is now a member variable of a class. Finally, call addActionListener(hider) on button01 and addActionListener(shower) on button02. This will allow the buttons to call the actionPerformed functions on their respective listeners. When finished, the code should look like this:

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class SampleGUI {
    private static JLabel label01;
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        Container contentPane = frame.getContentPane();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);
        contentPane.setPreferredSize(new Dimension(300, 300));

        label01 = new JLabel("Hello, world!");
        contentPane.add(label01);
        label01.setBounds(10, 15, 100, 30);

        JButton button01 = new JButton("Hide");
        JButton button02 = new JButton();
        button02.setText("Show");
        contentPane.add(button01);
        contentPane.add(button02);
        button01.setBounds(10, 45, 70, 30);
        button02.setBounds(10, 85, 70, 30);

        ActionListener hider = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                label01.setVisible(false);
            }
        };
        ActionListener shower = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                label01.setVisible(true);
            }
        };

        button01.addActionListener(hider);
        button02.addActionListener(shower);

        frame.pack();
        frame.setVisible(true);
    }
}
```

Figure 8: Code with Buttons and ActionListeners
When the code is run, the two buttons should make the “Hello, world!” text appear and disappear.

Adding a New Container Class:

It is also possible to add additional containers to the contentPanel. This is useful when several components need to be controlled at once. To do this in Java, start by adding a new private static JPanel to the class’s member variables. In the main function, initialize the JPanel, add it to the content panel, and give it bounds of 5,155, 290, and 140. This will create a new JPanel that takes up the bottom half of the window. However, before components can be added to the new panel, the panel’s layout manager needs to be set to null, just like the Content Panel’s. Do this by calling setLayout(null) on panel01.

To add components to the new panel, follow the same procedure to add components to the old one. Start out by creating a new JLabel, two new URL objects, two new JRadioButton objects, and a new ButtonGroup object. They should all be declared as private static class member variables. The JLabel will display one of two images. The URL’s will provide a link to where the images are on the internet. The two JRadioButtons will allow the user to choose between two images that will be displayed. Finally, the ButtonGroup links the two JRadioButtons together, so only one can be active at a time.

Begin initializing these new components, placing them within the newly created panel01, not the contentPane used until now. When setting the components’ bounds, bear in mind that the X and Y coordinates are relative to panel01’s position. This means that placing an object at (0,0) on panel01 will place it at (5,155) relative to the whole window. Place the JLabel at (120,5,50,50), the and the JRadioButtons at (70,60,85,30) and (155,60,85,30). The ButtonGroup is not a component and, therefore, should not be added to a panel, nor given bounds.
To allow the JLabel to display a picture, the image must first be loaded into memory. Before initializing label02, add the following block of code:

```java
img01url = null;
try {
    img01url = new URL("http://img11.imageshack.us/img11/7997/profilespistolpiratebus.jpg");
} catch (MalformedURLException e1) {
    e1.printStackTrace();
}

img02url = null;
try {
    img02url = new URL("http://img125.imageshack.us/img125/8534/halloweenninjacostumema.jpg");
} catch (MalformedURLException e1) {
    e1.printStackTrace();
}
```

Figure 10: How to Make URL Objects

This code will create links the program can use to go obtain images from the internet locations [http://img11.imageshack.us/img11/7997/profilespistolpiratebus.jpg](http://img11.imageshack.us/img11/7997/profilespistolpiratebus.jpg) and [http://img125.imageshack.us/img125/8534/halloweenninjacostumema.jpg](http://img125.imageshack.us/img125/8534/halloweenninjacostumema.jpg). Next, take these two URL’s and pass them into an ImageIcon object to change the picture on the JLabel. Do this by calling the setIcon(new ImageIcon(img01url)) function on label02.

Next, set the two radio buttons’ text to read “Pirate” and “Ninja”. To configure them so only one may be selected at a time, initialize the button group and call add(rButton01) and add(rButton02) on it. These buttons are now managed by the button group and can now only be selected one at a time.

In order to add functionality to the radio buttons, add one ActionListener to each radio button. These ActionListeners will be similar to the hider and shower, except they will set a new icon for label02 instead of setting the visible condition of label01. For the pirate radio button, call label02.setIcon(new ImageIcon(img01url)), and call label02.setIcon(new ImageIcon(img02url)) for the ninja button.

Now that this smaller container has been set up, the program can manipulate it and all of its contained components at once. To show this, change the hider and shower ActionListeners to show and hide panel01. When setVisible(false) is called on panel01, not only will the panel hide itself, but it will also hide all of its child components.
This final version of the code should look like this:

```java
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.awt.image.*;
import java.awt.scroll.*;
import java.io.*;

class SampleGUI {
  private static JLabel label01;
  private static JPanel pane01;
  private static JLabel label02;
  private static JRadioButton button01;
  private static JRadioButton button02;
  private static ButtonGroup buttonGroup;
  private static URL image02;

  public static void main(String[] args) {
    JFrame frame = new JFrame();
    Container contentPane = frame.getContentPane();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLayout(null);
    contentPane.setPreferredSize(new Dimension(300, 300));

    label01 = new JLabel("Hello, world!");
    contentPane.add(label01);
    label01.setBounds(10, 15, 100, 20);

    JButton button01 = new JButton("Hide");
    JButton button02 = new JButton();
    button02.setText("Show");
    contentPane.add(button01);
    contentPane.add(button02);
    button01.setBounds(10, 45, 70, 30);
    button02.setBounds(10, 85, 70, 30);

    ActionListener hide = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        pane01.setVisible(false);
      }
    };
    ActionListener show = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        pane01.setVisible(true);
      }
    };

    button01.addActionListener(hide);
    button02.addActionListener(show);
  }
}
```

Figure 11a: Final GUI Code
Figure 11b: Final GUI Code Continued
This will display a panel that looks like this:

![Final GUI Window](image)

Figure 12: Final GUI Window
Conclusion:

The composite pattern is an easy way to organize GUI elements such that they can be managed with minimal code under the client program. In addition, these containers can be modularized such that they can used and re-used as needed in many applications. For example, another program could import the SampleGUI class and call SampleGUI.main(null). This would create a new copy of the window created above.

There are plenty of other Swing and AWT components not mentioned here that can be added to a GUI. To see documentation on these, and other available Java classes, visit the Java API at:

http://java.sun.com/javase/6/docs/api/