Teaching GUI Design and Event Handling Using Patterns

Hans Dulimarta and Jagadeesh Nandigam
School of Computing and Information Systems
Grand Valley State University
Allendale, Michigan 49401, USA
dulimarh@cis.gvsu.edu, nandigaj@gvsu.edu

Abstract - *Graphical User Interface (GUI) has become a common way to build and deploy desk-top software in introductory computer science courses. Students definitely enjoy building GUI programs as opposed to command-line programs. Writing a GUI program requires a paradigm shift from sequential processing to event-driven processing. Building a GUI program involves two main activities – design of user interface and implementing appropriate event handling. There are several approaches (code patterns or styles) to building GUI programs. Introductory computer science textbooks suitable for use in CS1 and CS2 courses tend to describe and use exclusively one approach to building GUI programs. This exclusive use of one approach to teaching GUI design is not effective as it can cause confusion when students move from one course to another and expect to see only the style of GUI program they have learned before. It is important that students are progressively exposed to different styles of GUI design independent of the approach suggested and used in the textbook adopted for the course. This paper presents various approaches to GUI design and event handling using Java Swing framework along with pros and cons of each approach.*

Keywords: GUI design, view patterns, event handling patterns.

1 Introduction

Since the 1980s Graphical User Interface (GUI) has been used as a standard means to deploy software. Certainly, some applications are designed to be "faceless" and they run without interacting with a human user. But, as future software developers, students need to be equipped with the necessary skills to build either type using proper techniques.

Writing a GUI program requires a paradigm shift from sequential processing to event-based processing. Callback functions must be provided to respond to various events generated by the GUI framework. There are several code patterns that can be employed when building GUI programs. Many textbooks [1, 2, 3] on Java programming used in CS1 and CS2 courses tend to describe one way of building a GUI program and all the GUI examples shown in the book are solved using this approach. When students move from one course to next (say CS1 to CS2), the book used in the next course may use a different style of GUI programming. This change in approach to GUI design from one book to another confuses students, especially those in the introductory computer science courses. Even the instructors teaching these courses must be aware of different approaches to building GUI programs when adopting a different textbook for a course. In these cases, the instructor must make a conscientious attempt to use the same approach as used in the textbook in an effort not to confuse students. It is important that instructors discuss and demonstrate with examples different approaches to building GUI programs and appropriateness of each of these approaches. Depending on the complexity of a GUI program to be developed, students will then be able to select the right approach to use for the problem at hand.

This paper describes various code patterns for teaching GUI design and event handling techniques in introductory CS courses. The rest of this paper is organized as follows. The Model-View-Controller (MVC) design pattern and its variants are discussed in Section 2. Sections 3 and 4 describe various patterns to view design and event handling. Section 5 discusses techniques for adding the model component to a GUI application. Section 6 describes alternative GUI design tools. Recommended approaches to teaching GUI design and event handling in introductory CS courses are outlined in Section 7. Section 8 ends the paper with concluding remarks.

2 Design Patterns

In object-oriented development, a large library of design patterns is used to assist programmers to discover inter-object relationships and structures commonly used in object-oriented programs. The book by the "Gang of Four" [4] is the *de facto* standard reference for design patterns, but the materials presented in the book are aimed for more experienced and professional programmers. Nevertheless, early CS students can benefit from understanding some fundamental design patterns and use them in writing complex multi-object programs [5].

One of the fundamental design patterns encountered by early CS students is the Model-View-Controller (MVC) design pattern, which also has inspired the implementation of a large number of Graphical User Interface (GUI) frameworks. This pattern gained popularity after its major use for building user interfaces in Smalltalk during the late 1970s [4]. Despite its popularity and wide-spread use in many real

software projects, Hansen and Fossum claim that the pattern receives inadequate attention and CS educators should emphasize the design decisions that guide the development of an application [6]. The MVC design pattern consists of three kinds of objects:

- The model manages the data used by the application.
- The view presents the model in a form that allows the user to interact with the data.
- The controller handles user actions on the view and initiates proper actions for updating the data.

There are also variants of the MVC design pattern. Two of these variants are called *model-delegate* and *model-view-presenter*. In most applications, the view and controller are coupled very tightly. The resulting design pattern is known as *model-delegate*. The GUI framework in Java Swing employs this pattern.

The MVC pattern seems to imply that the model update is initiated by user actions on the view. In some applications, the model may be updated by an external source and the view must be updated accordingly. Consider a case when the user initiates a request to download remote data into the model. To keep the model and the view in sync with each other, the view must be updated only after the remote data is downloaded in full. For this use case, the view update must be initiated from the model. A common technique to accomplish this mechanism is to use the *Observer* pattern between the view and the model. This variant is known as *model-view-presenter* and was proposed by Fowler [7].

Beyond using well-known patterns, Bishop and Horspool [8] have identified common principles in GUI programming and use them in developing an XML-based GUI description language.

Using a simplified pattern than the standard MVC, Swing combines the view and controller roles into the *UI Delegate*. Different patterns to implement the view of a GUI application are presented in the next section. Section 4 describes several patterns to implement event handling in a GUI application.

3 View Patterns

In this section we will explore a number of techniques for implementing a GUI application using the Java Swing framework. We specifically exclude techniques used by modern IDEs that provide a visual GUI editor and generate the GUI code for Java developers. Instead, we will focus on techniques that can be used when the GUI code is hand-written by the users themselves.

Swing provides three top-level container classes: *JFrame*, *JApplet*, and *JDialog* as well as one general-purpose container class *JPanel*. Out of the three top-level container classes, we will focus our discussion on JFrame. With BorderLayout as its default layout manager, JFrame is a proper choice as a top-level container. The five sections of BorderLayout allow GUI components to be organized into coherent groups.

In contrast to JFrame, a container created from JPanel has the FlowLayout as its default layout. A FlowLayout does not provide a natural mechanism to organize its contents into coherent groups.

There are mainly three patterns to implementing the view in a GUI application. The table below summarizes these patterns. Rest of this section describes each pattern further with a sample code along with pros and cons of applying that pattern.

Patterns to Implement View in GUI App	
Pattern 1	Inheritance from JPanel
Pattern 2	Inheritance from JFrame
Pattern 3	JFrame as instance variable(s)

3.1 Pattern 1 – Inheritance from JPanel

We inherit a class from JPanel and place an instance of this panel into a JFrame's content pane. A typical Java code that uses this pattern is shown below:

```
public class MainView extends JPanel {
   public MainView() {
      setLayout(____);
      add(___);
      add(___);
   }
}

public class Main {
   public static void main(String[]
                                    args) {
      JFrame top = new JFrame();
      top.getContentPane().add(new
                              MainView());
      top.pack();
      top.setVisible(true);
   }
}
```

The above two classes are typically written as two separate files: MainView.java and Main.java. When more views are needed in the GUI, additional classes like "MainView" must be provided.

Pros: In general, this technique works well for a GUI program with a small number of widgets where they can be contained in a single panel.

Cons: When this technique is used, the event handling logic is usually written within the same class of the corresponding view panel. This approach may not facilitate easy updating to components in one view panel due to user actions in a different view panel. For instance, a text field in a panel in the north section of a JFrame is to be enabled/disabled based on the user action on a radio button in a panel in the east section of the JFrame. To implement such a feature, appropriate arguments must be passed from one view panel to another.

3.2 Pattern 2 – Inheritance from JFrame

For a more complex GUI design, a different technique should be used. To avoid interdependency across view panels as described earlier, we can move away from JPanel-based design and pull everything into a big JFrame. A typical Java code that uses this approach is given below:

```
public class GUI extends JFrame {

   public GUI() {
     add(viewOne(),BorderLayout.NORTH);
     add(viewTwo(), BorderLayout.CENTER);
     pack();
     setVisible(true);
   }

   private JPanel viewOne() {
      JPanel myView = new JPanel();
      myView.setLayout(____);
      /* setup components of this view */
      return myView;
   }

   private JPanel viewTwo() {
      JPanel myView = new JPanel();
      myView.setLayout(____);
      /* setup components of this view */
      return myView;
   }

   public static void main(String[]
                                 args) {
      GUI mainwin = new GUI();
   }
}
```

Pros: This "all-in-one" approach may seem natural to early CS1 students when they are not exposed to "functional decomposition" yet. All GUI components (panels, buttons, text fields, menu items, etc.) needed in the program are declared as instance variables in the class so they can be accessed from anywhere in the class. Visual updates that require information from several components can be easily written.

Cons: The amount of code for designing the GUI using this pattern can be overwhelmingly long and difficult to debug. In order to remedy this, the code can be organized into several private methods, each responsible for managing one coherent group. Essentially, this is similar to the JPanel-based technique described earlier except any cross views dependencies can be easily handled.

3.3 Pattern 3 – JFrame as instance variable(s)

An alternative to the second technique, the GUI class can be designed without inheritance. Instead, a JFrame instance variable is defined in the GUI class.

```
public class GUI {
    private JFrame top;

    public GUI() {
       top = new JFrame();
       top.add(_____);
       top.add(_____);
       top.pack();
       top.setVisible(true);
    }

    /* private methods defined here */

    public static void main(String[]
                                 args) {
       GUI mainwin = new GUI();
    }
}
```

Pros: This technique is appropriate for early CS1 students when they are not yet exposed to inheritance. In fact this technique is the most flexible technique of all, since it allows provision of multiple top-level containers, in which case several JFrame (or JDialog) instance variables can be defined.

4 Event Handling Patterns

One of the central facilities in a GUI framework is handling of events via proper callback functions. Swing implements this facility through events and event listeners. An event-handling object can be instantiated either from either a Listener or an Adapter.

The most common technique for event handling is to attach the listener (or adapter) to the corresponding view. This is usually accomplished by one of the following techniques or patterns. This can be accomplished by using one of the patterns shown in the table below. Each of these patterns are discussed in the following sections using the MouseMotionListener (or MouseMotionAdapter) as an example.

Patterns to Implement Event Handling in GUI App	
Pattern 1	View implements Listener
Pattern 2	Inner Classes for Listener
Pattern 3	Anonymous Classes
Pattern 4	Adapter Classes
Pattern 5	External Classes

4.1 Pattern 1 – View Implements Listener

The view class implements one or more listeners. A typical java code that uses this technique is shown below:

```
public class GUIView implements
                  MouseMotionListener {
   private JPanel landscape;

   public GUIView() {
      landscape = new JPanel();
```

```
      landscape.addMouseMotionListener(
                                    this);
   }

   public void mouseMoved(MouseEvent
                               m_event) {
   }

   public void mouseDragged(MouseEvent
                               m_event) {
   }
}
```

Pros: This technique works very well for a simple GUI program with a small number of widgets.

Cons: 1) Menus (JMenu & JMenuItem), buttons (JButton), and input fields (JTextField) are among the most popular components used by early CS students. In Swing, these three groups of components trigger ActionEvents that must be handled by ActionListeners. Using the above technique, the corresponding actionPerformed method may be overwhelmingly too long to write. 2) For a more complicated GUI program, developers may prefer to organize the mouse-handling logic into several methods. For instance, dragging the mouse over a 2D graphics may be handled differently from dragging the mouse over landscape of 3D objects. The above technique forces the developer to write everything into one big mouseDragged method.

4.2 Pattern 2 – Inner Classes for Listener

The second technique calls for declaration of private inner classes. Each class typically implements a specific Swing listener.

```
public class GUIView {
   private JPanel landscape;
   Handler2D motionHandler;

   public GUIView() {
      landscape = new JPanel();
      motionHandler = new Handler2D();
      landscape.addMouseMotionListener(
                           motionHandler);
   }

   private class Handler2D implements
                    MouseMotionListener {
      public void mouseMoved(MouseEvent
                              m_event) {
      }

      public void mouseDragged(MouseEvent
                              m_event) {
      }
   }
}
```

Pros: Event handling logic can be written into independent cohesive classes. Organization of event handling logic into separate (inner) classes facilitates easier refactorization.

Cons: This technique requires more code writing and including inner classes in a Java class may confuse some students who cannot clearly distinguish classes from objects.

4.3 Pattern 3 – Anonymous Classes

Since the inner classes are usually private, their visibility is limited to the GUIView itself, and the declaration and instantiation of Handler2D in the above technique can be merged into one step, by instantiating an object from an anonymous class.

```
public class GUIView {
   private JPanel landscape;

   public GUIView() {
      landscape = new JPanel();
      landscape.addMouseMotionListener(
                           motionHandler);
   }

   private MouseMotionListener
                     motionHandler =
      new MouseMotionListener() {
        public void mouseMoved(MouseEvent
                               m_event) {
        }

       public void mouseDragged(MouseEvent
                               m_event) {
        }
      };
}
```

Pros: Slightly more concise than the previous technique of using private inner classes.

Cons: The syntax for writing anonymous classes may seem too complex to comprehend by early CS students, especially when the IDE used does not provide auto completion / code assist feature that supports this syntax.

4.4 Pattern 4 – Adapter Classes

For Listeners that define more than one method, Swing provides a corresponding adapter. Adapters avoid writing empty callback methods.

Cons: A minor misspelling in writing the method names will result in the program failure to override the intended callback method. Most students may easily overlook this mistake, especially when they are under crunch time. This can be avoided by encouraging students to first use Listeners and later refactor the design to use Adapter in place of Listeners.

4.5 Pattern 5 – External Classes

Technically it is possible to write the event handling class as an external class. Using the four techniques described earlier, the event-handling classes (and methods) have direct visibility of all the instance variables declared by the View

class. On the contrary, an external class does not have direct visibility, and hence any variables needed by the event-handler must be passed from the View class.

5 Adding the "Model" Component

So far, we excluded discussions on including the model into the picture. There seem to be only two options: to separate or not to separate. Some instructors may decide not to enforce model separation in the first few assignments. As students learn more about "functional decomposition," it makes sense to enforce model separation for later assignments. When model separation is enforced, the interaction between the model and the view components fits into one of the following two techniques:

5.1 Pull by View

Using this technique the View invokes two methods in the Model. The first method invocation triggers the model to update its internal state and the second method invocation to pull the new state and used for updating the view. This technique may seem easier to beginners and suitable when the updated state can be determined immediately. When updating the model's internal state takes too long to complete, the View may stay "frozen" and unresponsive for a noticeable duration.

5.2 Push by Model

Using this technique, the View makes only one invocation to trigger the model to update its internal state. The model will then push the new state to the view. Implemented improperly, this technique may entice students to write unnecessarily complex interaction between the two components. A better approach of properly pushing the model internal states to the View is to use the Observer and Observable pair.

6 Alternative GUI Design Tools

Many modern IDEs provide a feature for designing the user interface of a program using a drag-n-drop technique. This feature certainly saves developer time to create the GUI, but most of the time "hand-written" GUI code is much leaner than those generated by these IDEs. Using these tools, the event-handling code snippets can be injected into various sections of the program.

Some standalone GUI editors/designer tools may support multiple GUI platforms, and these tools usually allow the GUI design to be saved into a file (such as an XML file) for later retrieval. The source code for specific GUI platforms can be generated and compiled together with the rest of the program. A number of GUI frameworks like Gtk or the Android View system include facilities to "inflate" or "render" the visuals from an XML description.

7 Recommendations

The two sets of choices described above allow instructors to choose the best option, out of sixteen (or twenty) different combinations, that works well for a particular audience. However, since learning revolves as a progression, the baseline for selecting the "best" option keeps shifting as students learn more towards the end of the semester.

The available options as described above seem to share a common theme: choosing between inheritance and composition. We claim that there is no particular one option that fits best for all situations. After completion of a typical CS1 and CS2 sequence, it is important that students gain good understanding of the various patterns described here. More importantly, they are equipped with necessary knowledge to wisely choose the right pattern to apply in a project.

In our discussion above, we mentioned a number of pros and cons for each pattern or option. The drawbacks of one option can be carefully posed to be an opportunity for students to grow in their learning experience.

8 Conclusions

Building GUI programs require a paradigm shift from sequential processing to event-based processing. Two main activities in building a GUI program are design of user interface and implementation of event handling. This paper presented several approaches (as code patterns) to user interface design and event handling. It is important that students be exposed to different approaches to GUI design and event handling along with pros and cons of these approaches in CS1 and CS2 courses.

9 References

[1] David Barnes, Michael Kolling. *Objects First with Java*. 4th. Prentice Hall, 2009.

[2] Julie Anderson and Herve Franceschi, *Java Illuminated*. 3rd edition. Jones-Bartlett Learning, 2011.

[3] J Lewis and W Loftus, *Java Software Solutions*, 6th edition. Addison-Wesley, 2008.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns, Addison-Wesley, 1995.

[5] Prasun Dewan, "Teaching Inter-Object Design Patterns to Freshmen", *SIGCSE '05*, St. Louis, Missouri, 2005.

[6] Stuart Hansen and Timothy Fossum, "Refactoring Model-View-Controller", *CCSC Midwest*, Decatur, Illinois, 2005.

[7] Martin Fowler, GUI Architectures, http://www.martinfowler.com/eaaDev/uiArchs.html (accessed Mar 7, 2011).

[8] Judith Bishop and Nigel Horspool, "Developing Principles of GUI Programming Using Views", *SIGCSE '04*, Norfolk, Virginia, 2004.