Software Development (CS2500)

Lecture 37: **Event Handlers**

M.R.C. van Dongen

January 24, 2011

Contents

1 Outline

This lecture is about *event handlers* and GUIs. It starts with our first design pattern: the observer pattern. The observer pattern is heavilly used in GUI applications. It is known/used as "the" ⟨X⟩-event/⟨X⟩-event listener pattern. The rest of the lecture is an introduction to GUIs. Specifically, we shall study the components of a Java GUI, and buttons which change if you click on them.

2 The Observer Pattern

The *observer pattern* is a commonly used design pattern. It defines a one-to-many object dependency so that if one object's state changes, all its dependents are automatically notified [Gamma *et al.*, 2008]. The

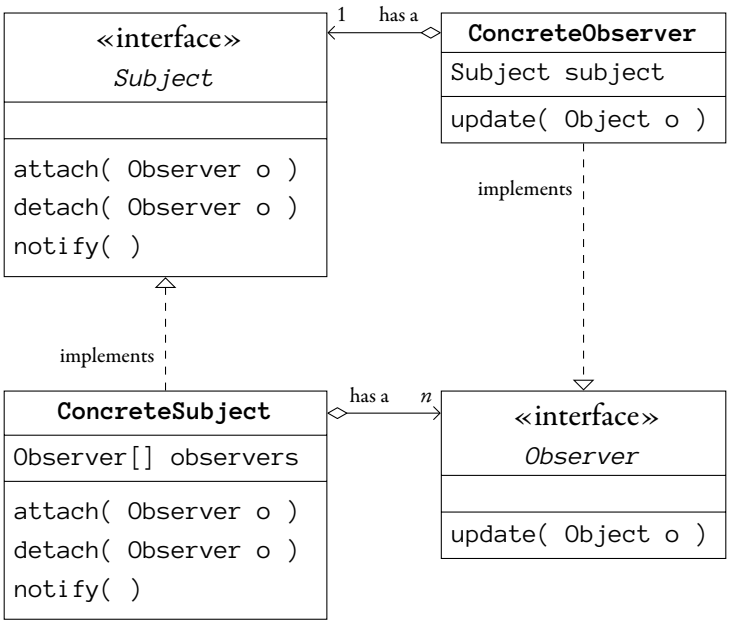

Figure 1: Observer Pattern in UML.

design pattern is also know as Dependents, Publish-Subscribe [Freeman and Freeman, 2005, Pages 44–78], and Event-Listener. The pattern works as follows:

- There is one `Subject`. You may regard it as a newspaper.
- There are zero or more `Observers`. You may regard them as potential newspaper readers.
- An `Observer` can be `attached` to the `Subject`. This is the equivalent of subscribing to the newspaper.
- An `Observer` can be `detached` from the `Subject`. This is the equivalent of unsubscribing to the newspaper.
- If the `Subject`'s state changes it notifies all its `Observers`. This is done by calling each `Subject`'s `update( )` method.

Figure 1 depicts a UML class diagram of the design pattern.

2.1 Case Study

Let's implement an example. We have a newspaper and readers of the newspaper. The readers can subscribe and unsubscribe. The newspaper informs the subscribers about new newsitems.

The `Subject` and `Observer` are best implemented with an interface. For the moment we write our `Subject` interface as follows.

```Java
public interface Subject {
    public void attach( Observer subscriber );
    public void detach( Observer subscriber );
    public void notify( String message );
}
```

For the purpose of an example, we give `notity` a `String` argument. The following is the `Observer` interface.

```Java
public interface Observer {
    public void update( String message );
}
```

Having defined the interfaces we proceed by implementing a concrete class for each of them. The following is the class `ConcreteObserver` which implements the readers of the newspaper.

```Java
public class ConcreteObserver implements Observer {
    String name;
    public ConcreteObserver( String name ) {
        this.name = name;
    }

    @Override
    public void update( String message ) {
        System.out.println( name + " updated: " + message );
    }
}
```

The following is a concrete implementation of the `Subject`.

```Java
import java.util.ArrayList;

public class ConcreteSubject implements Subject {
    private final ArrayList<Observer> subscribers;

    public ConcreteSubject( ) {
        subscribers = new ArrayList<Observer>( );
    }

    @Override
    public void attach( Observer subscriber ) {
        subscribers.add( subscriber );
    }

    @Override
    public void detach( Observer subscriber ) {
        subscribers.remove( subscriber );
    }

    @Override
    public void notify( String news ) {
        for( Observer subscriber : subscribers ) {
            subscriber.update( news );
        }
    }
}
```

Having defined the concrete classes, we can use them in the following main class. Notice that it is *not required* that the variables have the types `Subject` and `Observer`. We could have also used any combination of `ConcreteSubject` and `Subject` for the newspapers and any combination of `ConcreteObserver` and `Observer` for the readers.

```java
public class Main {
    public static void main( String[] args ) {
        Subject eolas = new ConcreteSubject( );
        Subject sun   = new ConcreteSubject( );

        Observer john = new ConcreteObserver( "John" );
        Observer jane = new ConcreteObserver( "Jane" );
        Observer eoin = new ConcreteObserver( "Eoin" );

        sun.attach( john );
        sun.attach( eoin );
        sun.attach( jane );
        eolas.attach( jane );

        eolas.notify( "Assignment 5 handed back." );
        sun.notify( "Biffo quits and so do greens." );

        sun.detach( jane );

        sun.notify( "Jane unsubscribed." );
    }
}
```

When we run our program we get the following output:

```
$ java Main
Jane updated: Assignment 5 handed back.
John updated: Biffo quits and so do greens.
Eoin updated: Biffo quits and so do greens.
Jane updated: Biffo quits and so do greens.
John updated: Jane unsubscribed.
Eoin updated: Jane unsubscribed.
$
```

3 Windows

This section is an introduction to *windows*, JFrames really.

Without a window you could not write a GUI application. In Java a window is represented as a JFrame object. The JFrame is where you put your window's *widgets* in. Possible widgets are buttons, checkboxes, sliders, dialogue boxes, text fields, and so on. The appearance of a JFrame may differ from OS (operating system) to OS.

Creating a window is easy with JFrames. It gives some fresh air to your applications.

The following are the main operations that let you open a window with a JFrame.

- Create a JFrame.

  ```java
  JFrame frame = new JFrame( ⟨title string⟩ );
  ```

- Set the JFrame's closing operation.

  ```java
  frame.setDefaultClosingOperation( JFrame.EXIT_ON_CLOSE );
  ```

Figure 2: A `JButton` in a `JFrame`.

This method determines how the GUI behaves when you leave the window. Using `JFrame.EXIT_ON_CLOSE` exits the application using `System.exit( )` method.

- Make one or several widgets and add them to the `JFrame`.

```Java
JButton button = new JButton( "Click me" );
frame.getContentPane( ).add( button );
```

- Give the `JFrame` a size and make it visible. The size is provided as a pair of `int`s which represent the horizontal and vertical size in pixels.

```Java
frame.setSize( 300, 300 );
frame.setVisible( true );
```

The following is a whole program.

```Java
import javax.swing.*;

public class DummyButton {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "Our second Button" );
        JButton button = new JButton( "Click me" );
        frame.getContentPane( ).add( button );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setSize( 300, 300 );
        frame.setVisible( true );
    }
}
```

When you run the program you may see a similar button to the button which is depicted in Figure 2. That looks great and easy. But, horror of horrors, when we click the button: nothing happens.

4 Events

This section explains how to make GUI application interactive. The basic ingredients are *events*. The following starts by explaining events and by explaining events in terms of the observer pattern. This is continued by explaining how to translate the observer pattern in terms of `gui` operations. As we shall see, all we need to do is define an *event listener*. This section concludes by writing two interactive button applications.

It should not have come as a complete surprise that our button did nothing when we clicked it. After all, we didn't tell it what to do. Let alone, *how*, and *when*. The button already has a mechanism to tell us *when* it's clicked:

Event: When the button is clicked this generates an event. This event is called a button event.

To make the button do something when it's clicked we need two things:

Handler: A button event handler that makes the button behave *how* it is supposed to behave.

Listener: A button event listener that detects button events. Each time the button event listener detects the button event, it calls the button event handler.

The following is the mechanism.

1. The button event is activated when the button is clicked.
2. The button event triggers the button event listener.
3. The button event listener calls the button event handler.

4.1 Back to Observers

The mechanism which we studied in the previous section is just an instance of the observer pattern. The `JButton` is the `Subject`. Clicking the `JButton` is a *user action*. The `JButton` turns it into a button event object, `event`. It may be thought of as a call to `notify( event )`. The `Observers` are the button event listeners. Each `Observer` implements its button event handler. Each event handler is a dedicated `update( )` method. The call `update( event )` allows the `Subject` to send the button event to the `Observer` object.

4.2 Creating an Event Listener

This section explains how to create an event listener. As we shall see this is easy.

An event listener class implements an event listener interface.

- Button event listeners implement the button listener interface,
- mouse event listeners implement the mouse listener interface,
- and so on.

Some interfaces have more than one `notify( )` method. For buttons you usually are only interested if it's clicked. However, it is possible to distinguish between events pressing and releasing a button. The "click events" for `JButtons` are `ActionEvent` objects. So to implement our application we implement the `ActionListener` interface. The method `actionPerformed( ActionEvent event )` in the interface is equivalent to the `Observer`'s `update( )` method.

4.3 An Interactive Button

The following presents a simple example of an interactive button. All it does is print `You clicked me` when you click it. For simplicity we implement the button listener class as part of the `main` class. The code for creating the `JFrame`-specific code has also been omitted for simplicity.

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

public class SimpleGUI implements ActionListener {
    private final JButton button;

    public static void main( String[] args ) {
        JFrame frame = ⟨Create JFrame⟩
        SimpleGUI gui = new SimpleGUI( );
        ⟨Remaining JFrame-related statements.⟩
    }

    public SimpleGUI( ) {
        button = new JButton( "Click me" );
        button.addActionListener( this );
    }

    @Override
    public void actionPerformed( ActionEvent event ) {
        button.setText( "You clicked me" );
    }
}
```

Since the button event listener is in the class `SimpleGUI`, and since `SimpleGUI` implements the `ActionListener` interface, each `SimpleGUI` object IS-An `ActionListener` object. In the constructor `SimpleGUI( )` we attach the observer (action even listener) `this` to the `JButton button` with the call `button.addActionListener( this )`.

4.4 A Button with a Counter

Now that we know the basics we can easily add things to it. In this section we make the button count and display the number of times it has been clicked.

If you think about it all that's needed is add a counter attribute to the action listener object, change the constructor of the action listener, and change the implementation of the method `actionPerformed( )`. The following are the details. Of course the assignment `count = 0` could have been omitted. (Why?)

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

public class CountingButton implements ActionListener {
    private int clicks;
    private final JButton button;

    public static void main( String[] args ) {
        JFrame frame = ⟨Create JFrame⟩
        SimpleGUI gui = new SimpleGUI( );
        ⟨Remaining JFrame-related statements.⟩
    }

    public CountingButton( ) {
        clicks = 0;
        button = new JButton( "Click me" );
        button.addActionListener( this );
    }

    @Override
    public void actionPerformed( ActionEvent event ) {
        String text = "# clicks = " + ++ clicks + ". Try again.";
        button.setText( text );
    }
}
```

5 For Wednesday

Study the lecture notes, and study Pages 353–368 of Chapter 11.

6 Bibliography

References

[Freeman and Freeman, 2005] Eric Freeman and Elisabeth Freeman. Head First *Design Patterns*. O'Reilly, 2005.

[Gamma *et al.*, 2008] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns* Elements of Reusable Object-Oriented Software. Addison–Wesley, 2008. 36th Printing.