Inf2C Software Engineering 2017-18

Tutorial 3 (Week 8)

Design Patterns

Study this tutorial sheet and make notes of your answers BEFORE the tutorial.

1 Introduction

The purpose of this tutorial is to help improve your understanding of the concept of *design patterns*. For the first part you study a particular pattern – the Observer pattern – and an example realisation of it in Java. For the second part you consider more generally the role of design patterns in design. Study of design patterns ought to help improve your understanding of good design principles such as encapsulation, high cohesion and low coupling.

2 The Observer pattern

Take a look at popular online descriptions of this pattern. For example, visit one or both of the following:

- Wikipedia: `https://en.wikipedia.org/wiki/Observer_pattern`
- *Design Patterns Explained Simply* book:
  `https://sourcemaking.com/design_patterns/observer`

Your aim is to get the general idea of the pattern and how it reduces coupling.

Then take a look at how this pattern is central to the design of the Java Swing GUI library and how it handles input events. Go visit

`https://docs.oracle.com/javase/tutorial/uiswing/components/button.html`

and read the first part titled *How to Use the Common Button API* demonstrating the use of the JButton class with a class ButtonDemo. You might find it helps your understanding to download and run the code, though doing so is not required for this tutorial. Aim to understand how the JButton class and the ButtonDemo class, together with the ActionListener interface, realise the Observer pattern. When reading the code for the purposes of this tutorial, it is safe to ignore nearly all the Swing-related portions of the code. You don't

need to understand this Swing code to appreciate the pattern realisation. See Figure 1 on page 3 for a listing of the code of the ActionListener interface and the important parts of the ButtonDemo class.

To demonstrate your understanding of the code, create a UML class diagram showing the structure related to the Observer pattern, and create a UML sequence diagram showing what happens when one presses one of the buttons.

Keep the class diagram simple. Don't model the Java library class and interface generalisation hierarchies – just show the two classes JButton and ButtonDemo and the interface ActionListener.

Some questions to think about with the class diagram:

1. You don't want to dig into the implementation of JButton or one of its ancestor classes. Nevertheless, what association must you infer exists for the Observer pattern to be implemented and button events to be handled sensibly?

2. What approaches are taken in the two references on the Observer pattern and in the ButtonDemo example to the question of how an Observer object finds out about the nature of the events occurring at the Subject object?

3. Are there any associations in your class diagram that are contingent, that is, there in this particular example, but not required by the pattern?

4. Are there any other ways in which the implementation is structurally different from the abstract presentations of the pattern?

3 Design pattern in general

1. Design patterns suggest that you copy an existing solution and modify it correspondingly for your specific application. One thing novice programmers are taught is that *"Copy-and-paste"* programming is to be avoided. Worst still, you may apply the same design pattern more than once in the same project. Isn't this a good argument to avoid the use of design patterns?

2. If you agree that the main advantage to the use of design patterns is as an aid to communication with other developers. Is there any value in using design patterns for an individual project? If you believe that you will only ever write programs that no one else will read (which is a sad thought), is there any benefit to *learning* design patterns?

3. Some programming languages have existing features that subsume the need for particular design patterns. For example many dynamically typed languages (and some statically typed languages/variants) have a provision for *'MetaClasses'* and this largely obviates the need for the Factory pattern. Do you think that this means that the concept of a design pattern is not a useful one?

Paul Jackson. 1 Nov 2017.

```
public interface ActionListener extends EventListener {
    void actionPerformed(ActionEvent e);
}

public class ButtonDemo extends JPanel
                        implements ActionListener {
    protected JButton b1, b2, b3;

    public ButtonDemo() {
        ImageIcon leftButtonIcon = createImageIcon("images/right.gif");
        ImageIcon middleButtonIcon = createImageIcon("images/middle.gif");
        ImageIcon rightButtonIcon = createImageIcon("images/left.gif");

        b1 = new JButton("Disable middle button", leftButtonIcon);
        b1.setVerticalTextPosition(AbstractButton.CENTER);
        b1.setHorizontalTextPosition(AbstractButton.LEADING);
        b1.setMnemonic(KeyEvent.VK_D);
        b1.setActionCommand("disable");

        b2 = new JButton("Middle button", middleButtonIcon);
        b2.setVerticalTextPosition(AbstractButton.BOTTOM);
        b2.setHorizontalTextPosition(AbstractButton.CENTER);
        b2.setMnemonic(KeyEvent.VK_M);

        b3 = new JButton("Enable middle button", rightButtonIcon);
        //Use the default text position of CENTER, TRAILING (RIGHT).
        b3.setMnemonic(KeyEvent.VK_E);
        b3.setActionCommand("enable");
        b3.setEnabled(false);

        //Listen for actions on buttons 1 and 3.
        b1.addActionListener(this);
        b3.addActionListener(this);

        b1.setToolTipText("Click this button to disable the middle button.");
        b2.setToolTipText("This middle button does nothing when you click it.");
        b3.setToolTipText("Click this button to enable the middle button.");

        //Add Components to this container, using the default FlowLayout.
        add(b1);
        add(b2);
        add(b3);
    }

    public void actionPerformed(ActionEvent e) {
        if ("disable".equals(e.getActionCommand())) {
            b2.setEnabled(false);
            b1.setEnabled(false);
            b3.setEnabled(true);
        } else {
            b2.setEnabled(true);
            b1.setEnabled(true);
            b3.setEnabled(false);
        }
    }
     ....
}
```

Figure 1: JButton demonstration code