Inf2-SEPP 2025-26

Tutorial 4 (Week 6)

Design Patterns

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

Introduction

The purpose of this tutorial is to help improve your understanding of the concept of *design patterns*. You will study an example of the realisation of the Observer pattern in Java. Then, you will get the chance to identify two design patterns that are suitable for given contexts. Throughout, we will make use of UML diagrams to talk about the patterns.

Study of design patterns ought to help improve your understanding of good design principles such as encapsulation, high cohesion and low coupling.

1 The Observer pattern

Revise the Observer Pattern from Lecture 11. 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. Then, see Figure 1 on page 4 for a listing of the code of the ActionListener interface and the important parts of the ButtonDemo class. Aim to understand how the JButton class and the ButtonDemo class, together with the ActionListener interface, realise the Observer pattern.

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 approach is taken 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 included 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?

For the following two tasks, make sure you revised both Lecture 11 and Lecture 13 Part 1 and their associated reading.

2 Pattern identification 1

As part of a larger system, objects of classes `A` and `B` need to be able to perform a wide range of mathematical calculations (addition, substraction, multiplication, division, sine, cosine, logarithm, power, exponent, etc.). They also need to be able to undo any number of previously performed calculations.

A colleague of yours started this design by placing operations corresponding to the different mathematical calculations, and their undo operations, within each of the classes `A` and `B`. Another option they were considering was having a separate `Calculator` class which contains all of these operations and can be used by objects of the `A` and `B` classes.

For this exercise, you are not allowed to use the Java `java.lang.Math` library.

Address the following questions and tasks:

1. What is your opinion about the solutions proposed by your colleague? What makes each of them problematic? Think both about the current system requirements, as well as making the system easily extensible (e.g. by supporting more and different mathematical calculations by each of the classes `A` and `B`), understandable and maintainable on the long term.

2. What design pattern might be appropriate to use in this context? What are its advantages over your colleague's proposed solutions?

3. Create a UML class diagram to show your use of the pattern. You may focus only on the addition, substraction and sine operations.

4. Write your implementation of the class diagram from the previous question using Java.

3 Pattern identification 2

You are part of the team designing a system for a bank. You are asked by your customers to include in the system the functionality of automatically logging withdrawals, deposits and transfers as soon as they occur, such that the bank could later request reports using the log. It is essential that each transaction is recorded with a timestamp. Withdrawals and deposits must be recorded with an account and an amount, while transfers must be recorded using the account-from, account-to, and amount. The bank is considering adding records of other operations to the log in the future, as it expands its business.

For this exercise, you should be thinking of designing your own logging functionality, without using the Java class `Logger` from `java.util.logging`. Although in real life logs would be saved to secure storage (e.g. a database), for this exercise you are only required to think of the object-oriented design (assume there is no database).

Address the following questions and tasks:

1. Without the use of design patterns, how would you ensure that all the classes involved in withdrawals, deposits and transfers, and other future operations in the bank system, can log such data? Why is this problematic?

2. What design pattern might be appropriate to use in this context? Why is it a particularly useful candidate?

3. Create a UML class diagram to show only the part of the system's design which is concerned with logging.

4. Write your implementation of the class diagram from the previous question using Java.

```java
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