University of Ruhuna
Faculty of Technology
Department of Informtion and Communication Technology

Advanced Programming Practicum
(ICT3122)

Lab Sheet 4 29th Jan 2024

Implementing the Strategy Design Pattern for Payment Methods in an E-commerce App

Objective: Understand and implement the Strategy design pattern for implementing various payment methods in an e-commerce application.

Introduction: The Strategy pattern allows you to define a family of algorithms, encapsulate each one, and make them interchangeable. In this lab, we'll use the Strategy pattern to implement payment methods (PayPal and Credit Card) in an e-commerce application.

Step 1: Identify the Strategy Interface

Identify the common interface for all payment strategies. In this case, it is the **PayStrategy** interface. It defines methods like **pay** and **collectPaymentDetails**.

Step 2: Create Concrete Strategy Classes

Create concrete strategy classes (**PayByPayPal** and **PayByCreditCard**) that implement the **PayStrategy** interface. Each class should provide its own implementation for the payment and collecting payment details.

Step 3: Define Context Class

Define a context class named **Order**. This class contains a reference to the **PayStrategy** interface and uses it to process orders. The **Order** class should have methods like **processOrder**, **setTotalCost**, **getTotalCost**, **isClosed**, and **setClosed**.

Step 4: Update Client Code

Update the client code in the **Demo** class to demonstrate the use of the Strategy pattern. Clients can now select a payment method and process orders without knowing the details of the payment methods.

Step 5: Test the Implementation

Run the program and test different scenarios. Verify that the Strategy pattern allows you to easily switch between payment methods without modifying the client code.

Note to Students:

1. Understand the role of the **PayStrategy** interface and how it provides a common interface for all payment strategies.
2. Explore the concrete strategy classes (**PayByPayPal** and **PayByCreditCard**) and see how they implement payment and collecting payment details.
3. Observe how the context class (**Order**) uses the selected strategy to process orders without knowing the concrete details of the payment methods.
4. Test your implementation with various products and payment methods to ensure the flexibility and interchangeability of strategies.

Example Code Structure (Class and Method Names):

1. **PayStrategy** interface:
    - **boolean pay(int paymentAmount)**
    - **void collectPaymentDetails()**
2. **PayByPayPal** class:
    - Implementing the **PayStrategy** interface.
3. **PayByCreditCard** class:
    - Implementing the **PayStrategy** interface.
4. **CreditCard** class:
    - Dummy credit card class.
5. **Order** class:
    - **void processOrder(PayStrategy strategy)**
    - **void setTotalCost(int cost)**
    - **int getTotalCost()**
    - **boolean isClosed()**
    - **void setClosed()**
6. Updated **Demo** class:
    - Updated client code to demonstrate the use of the Strategy pattern.

Lab Sheet: Implementing the Observer Design Pattern in Java

Objective: Understand and implement the Observer design pattern for creating a simple topic and allowing observers to register and receive notifications when a new message is posted to the topic.

Introduction: The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. In this lab, we'll implement a simple Observer pattern for an e-commerce application where a topic sends notifications to registered observers when a new message is posted.

Step 1: Create the Subject Interface

Create the **Subject** interface with methods to register and unregister observers, notify observers of a change, and get updates from the subject.

```
public interface Subject {
    void register(Observer obj);
    void unregister(Observer obj);
    void notifyObservers();
    Object getUpdate(Observer obj);
}
```

Step 2: Create the Observer Interface

Create the **Observer** interface with methods to update the observer and attach the subject to observe.

```
public interface Observer {
    void update();
    void setSubject(Subject sub);
}
```

Step 3: Create the Concrete Subject (Topic) Class

Implement the **MyTopic** class that implements the **Subject** interface. This class maintains a list of observers, a message, and a boolean variable to track changes.

```
import java.util.ArrayList;
import java.util.List;

public class MyTopic implements Subject {
    private List<Observer> observers;
    private String message;
    private boolean changed;
    private final Object MUTEX = new Object();

    public MyTopic() {
        this.observers = new ArrayList<>();
    }

    @Override
    public void register(Observer obj) {
        if (obj == null) throw new NullPointerException("Null Observer");
        synchronized (MUTEX) {
            if (!observers.contains(obj)) observers.add(obj);
        }
    }

    @Override
    public void unregister(Observer obj) {
        synchronized (MUTEX) {
            observers.remove(obj);
        }
    }
    @Override
    public void notifyObservers() {
        List<Observer> observersLocal = null;
        synchronized (MUTEX) {
            if (!changed)
                return;
            observersLocal = new ArrayList<>(this.observers);
            this.changed = false;
        }
        for (Observer obj : observersLocal) {
            obj.update();
        }
    }

    @Override
    public Object getUpdate(Observer obj) {
        return this.message;
    }

    public void postMessage(String msg) {
        System.out.println("Message Posted to Topic:" + msg);
        this.message = msg;
        this.changed = true;
        notifyObservers();
    }
}
```

Step 4: Create the Concrete Observer Class

Implement the **MyTopicSubscriber** class that implements the **Observer** interface. This class consumes messages from the subject.

```
public class MyTopicSubscriber implements Observer {

    private String name;
    private Subject topic;

    public MyTopicSubscriber(String nm) {
        this.name = nm;
    }

    @Override
    public void update() {
        String msg = (String) topic.getUpdate(this);
        if (msg == null) {
            System.out.println(name + ":: No new message");
        } else
            System.out.println(name + ":: Consuming message::" + msg);
    }

    @Override
    public void setSubject(Subject sub) {
        this.topic = sub;
    }
}
```

Step 5: Test the Implementation

Create a test program (**ObserverPatternTest**) to consume the implemented Observer pattern. Register observers, attach them to the subject, and post messages to observe the notification.

```
public class ObserverPatternTest {

    public static void main(String[] args) {
        //create subject
        MyTopic topic = new MyTopic();

        //create observers
        Observer obj1 = new MyTopicSubscriber("Obj1");
        Observer obj2 = new MyTopicSubscriber("Obj2");
        Observer obj3 = new MyTopicSubscriber("Obj3");

        //register observers to the subject
        topic.register(obj1);
        topic.register(obj2);
        topic.register(obj3);

        //attach observer to subject
        obj1.setSubject(topic);
        obj2.setSubject(topic);
        obj3.setSubject(topic);

        //check if any update is available
        obj1.update();

        //now send message to subject
        topic.postMessage("New Message");
    }
```

```
}
```

When you run the **ObserverPatternTest** program, you should see output similar to the one you provided.

Note to Students:

1. Understand the role of the **Subject** and **Observer** interfaces.
2. Explore the concrete implementation of the **MyTopic** class (Concrete Subject) and the **MyTopicSubscriber** class (Concrete Observer).
3. Observe how the observers are registered, attached to the subject, and notified upon a change.
4. Test your implementation with various scenarios to ensure that the Observer pattern is working as expected.

Implementing the Command Design Pattern in Java

Objective: Understand and implement the Command design pattern for a File System utility that supports multiple operating systems (Unix and Windows).

Introduction: The Command pattern encapsulates a request as an object, thereby allowing for parameterization of clients with different requests, queuing of requests, and logging of the requests. In this lab, we'll implement the Command pattern for a File System utility that provides methods to open, write, and close files, supporting multiple operating systems.

Step 1: Create the Receiver Interface

Create the **FileSystemReceiver** interface with methods to open, write, and close a file.

```
public interface FileSystemReceiver {
    void openFile();
    void writeFile();
    void closeFile();
}
```

Step 2: Create Concrete Receiver Classes

Implement the receiver classes for Unix and Windows operating systems that implement the **FileSystemReceiver** interface.

```
public class UnixFileSystemReceiver implements FileSystemReceiver {
    @Override
    public void openFile() {
        System.out.println("Opening file in Unix OS");
    }

    @Override
    public void writeFile() {
        System.out.println("Writing file in Unix OS");
    }

    @Override
    public void closeFile() {
        System.out.println("Closing file in Unix OS");
    }
}
```

Step 3: Create Command Interface

Create the **Command** interface with a method **execute()**.

```
public interface Command {
    void execute();
}
```

Step 4: Create Concrete Command Classes

Implement command classes for opening, writing, and closing a file, each associated with a specific receiver.

```
private FileSystemReceiver fileSystem;
    public OpenFileCommand(FileSystemReceiver fs) {
        this.fileSystem = fs;
    }

    @Override
    public void execute() {
        this.fileSystem.openFile();
    }
}


public class WriteFileCommand implements Command {
    private FileSystemReceiver fileSystem;
    public WriteFileCommand(FileSystemReceiver fs) {
        this.fileSystem = fs;
    }

    @Override
    public void execute() {
        this.fileSystem.writeFile();
    }
}


private FileSystemReceiver fileSystem;
    public CloseFileCommand(FileSystemReceiver fs) {
        this.fileSystem = fs;
    }
    @Override
    public void execute() {
        this.fileSystem.closeFile();
    }
}
```

Step 5: Create Invoker Class

Create an invoker class that will execute the commands.

```
public class FileInvoker {
    private Command command;
    public FileInvoker(Command c) {
        this.command = c;
    }
    public void execute() {
        this.command.execute();
    }
}
```

Step 6: Utility Class to Determine OS and Create Receiver

Create a utility class to determine the underlying operating system and create the appropriate **FileSystemReceiver** object.

```
public class FileSystemReceiverUtil {
    public static FileSystemReceiver getUnderlyingFileSystem() {
        String osName = System.getProperty("os.name");
        System.out.println("Underlying OS is: " + osName);
        if (osName.contains("Windows")) {
            return new WindowsFileSystemReceiver();
        } else {
            return new UnixFileSystemReceiver();
        }
    }
}
```

Step 7: Create Client Class

Write a client class that uses the Command pattern to perform actions on the file system utility.

```
    public static void main(String[] args) {
        // Creating the receiver object
        FileSystemReceiver fs = FileSystemReceiverUtil.getUnderlyingFileSystem();

        // Creating command and associating with receiver
        OpenFileCommand openFileCommand = new OpenFileCommand(fs);

        // Creating invoker and associating with Command
        FileInvoker file = new FileInvoker(openFileCommand);

        // Perform action on invoker object
        file.execute();

        WriteFileCommand writeFileCommand = new WriteFileCommand(fs);
        file = new FileInvoker(writeFileCommand);
        file.execute();

        CloseFileCommand closeFileCommand = new CloseFileCommand(fs);
        file = new FileInvoker(closeFileCommand);
        file.execute();
    }
}
```

Note to Students:

1. Understand the role of the **FileSystemReceiver** interface and its implementations (**UnixFileSystemReceiver** and **WindowsFileSystemReceiver**).
2. Explore the **Command** interface and its implementations (**OpenFileCommand**, **WriteFileCommand**, and **CloseFileCommand**).
3. Observe how the **FileInvoker** class is used to execute the commands on the file system.
4. Test your implementation with various operating systems to ensure that the Command pattern is working as expected.