SOEN 6441
Advanced Program Practices

Lab 4: Refactoring, state & command pattern

Instructors: Dr. Joey Paquet joey.paquet@concordia.ca
Dr. Amin Ranj Bar amin.ranjbar@concordia.ca

TA: Hamed Jafarpour hamed.jafarpour@concordia.ca

Outline

- **Refactoring**
- **Example of Refactoring**
- **State Pattern**
- **Example of State Pattern**
- **Command Pattern**
- **Example of Command Pattern**

Refactoring

- Refactoring is the process of restructuring existing code without changing its external behavior.
- The goal of refactoring is to improve code quality, readability, and maintainability.
- It involves making small, incremental changes to the code to eliminate duplication, improve naming, simplify complex logic, and remove code smells.

Refactoring

- Some common refactoring techniques:
  - Extracting methods
  - Renaming variables,
  - Restructuring code to adhere to design principles like SOLID:
    - Single Responsibility Principle
    - Open/Closed Principle
    - Liskov Substitution Principle
    - Interface Segregation Principle
    - Dependency Inversion Principle

- Refactoring is an essential practice for maintaining and enhancing codebases over time, as it helps reduce technical debt and makes it easier to add new features or fix bugs without introducing regressions.

Refactoring (Example)

Original Code (Before Refactoring):

```java
public class Calculator {
    public static int calculateSum(int a, int b) {
        int result = 0;
        for (int i = 0; i < a; i++) {
            result++;
        }
        for (int j = 0; j < b; j++) {
            result++;
        }

        return result;
    }

    public static void main(String[] args) {
        int x = 5;
        int y = 3;
        int sum = calculateSum(x, y);
        System.out.println("The sum of " + x + " and " + y + " is: " + sum);
    }
}
```

Refactoring (Example)

Refactored Code (After Refactoring):

```java
public class Calculator {
    public static int calculateSum(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        int x = 5;
        int y = 3;
        int sum = calculateSum(x, y);
        System.out.println("The sum of " + x + " and " + y + " is: " + sum);
    }
}
```

Refactoring (Example)

- The Single Responsibility Principle (SRP) is one of the SOLID principles, and it states that a class should have only one reason to change.
- In other words, a class should have a single responsibility, which means it should encapsulate only one piece of functionality or behavior
- Original Code (Before Refactoring):

```
public class Order {
    private int orderId;
    private String orderDetails;

    public void createOrder() {
        // Code to create an order in the database
    }

    public void calculateTotalPrice() {
        // Code to calculate the total price of the order
    }

    public void sendConfirmationEmail() {
        // Code to send an email confirmation to the customer
    }
}
```

Refactoring (Example)

Refactored Code (After Refactoring):

```
public class Order {
    private int orderId;
    private String orderDetails;

    public void createOrder() {
        // Code to create an order in the database
    }
}

public class OrderCalculator {
    public double calculateTotalPrice(Order order) {
        // Code to calculate the total price of the order
    }
}

public class EmailService {
    public void sendConfirmationEmail(Order order) {
        // Code to send an email confirmation to the customer
    }
}
```

State Pattern

- The State pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes.
- It defines a family of classes representing various states and allows an object to transition from one state to another.
- This pattern is particularly useful in situations where an object's behavior needs to change based on its state but you want to avoid large conditional statements.

State Pattern

- In the State pattern, each state is typically represented by a separate class, and the context (the object that changes state) holds a reference to the current state object. When the state needs to change, the context updates the reference to the appropriate state object.
- This promotes cleaner and more maintainable code by encapsulating the behavior associated with each state.

State Pattern (Example)

- Imagine you have a simple document editor with two states: "Editing" and "Viewing." The editor can switch between these states, and each state behaves differently when you type text.

State Pattern (Example)

- Here's a straightforward example in Java:

```java
// Context class
public class DocumentEditor {
    private DocumentState currentState;

    public DocumentEditor() {
        // Initialize the editor in the "Viewing" state
        currentState = new ViewingState();
    }

    public void setCurrentState(DocumentState state) {
        currentState = state;
    }

    public void type() {
        currentState.type();
    }
}

// State interface
public interface DocumentState {
    void type();
}
```

```java
// Concrete states
public class EditingState implements DocumentState {
    @Override
    public void type() {
        System.out.println("You can edit the document.");
    }
}

public class ViewingState implements DocumentState {
    @Override
    public void type() {
        System.out.println("You can't edit. Viewing only.");
    }
}

// Client code
public class Main {
    public static void main(String[] args) {
        DocumentEditor editor = new DocumentEditor();

        editor.type();  // Output: "You can't edit. Viewing only."

        // Switch to editing mode
        editor.setCurrentState(new EditingState());
        editor.type();  // Output: "You can edit the document."

        // Switch back to viewing mode
        editor.setCurrentState(new ViewingState());
        editor.type();  // Output: "You can't edit. Viewing only."
    }
}
```

Command Pattern

The Command pattern is another behavioral design pattern that encapsulates a request as an object, thereby allowing you to parameterize clients with queues, requests, and operations.

It enables you to decouple the sender of a request from its receiver, providing more flexibility and extensibility in a system.

Command Pattern

- In the Command pattern, a command object represents an action to be performed.
- It typically includes an execute method that performs the action.
- Clients can create and configure command objects and place them in a queue, stack, or other data structures.
- This pattern is often used in systems where you need to support various actions, command history, or transactional behavior.

Command Pattern (Example)

- A simple example of the Command Pattern using a basic light switch scenario:

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

// Receiver class
class Light {
    void turnOn() {
        System.out.println("Light is on");
    }

    void turnOff() {
        System.out.println("Light is off");
    }
}

// Concrete command classes
class TurnOnCommand implements Command {
    private Light light;

    TurnOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOn();
    }
}
```

```
class TurnOffCommand implements Command {
    private Light light;

    TurnOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOff();
    }
}

// Invoker class
class RemoteControl {
    private Command command;

    void setCommand(Command command) {
        this.command = command;
    }

    void pressButton() {
        command.execute();
    }
}
```

```
// Client code
public class Main {
    public static void main(String[] args) {
        Light livingRoomLight = new Light();
        Command turnOnCommand = new TurnOnCommand(livingRoomLight);
        Command turnOffCommand = new TurnOffCommand(livingRoomLight);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(turnOnCommand);
        remote.pressButton();  // Output: "Light is on"

        remote.setCommand(turnOffCommand);
        remote.pressButton();  // Output: "Light is off"
    }
}
```

Thank You
For Your Attention

Reference

[1] https://www.visual-paradigm.com/tutorials/statedesignpattern.jsp
[2] https://www.visual-paradigm.com/tutorials/commanddesignpattern.jsp