МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
КАФЕДРА ИИТ

Лабораторная работа №6

По дисциплине: «Современные платформы программирования»

Выполнил:
Студент 3 курса
группы ПО-8:
Макаревич Е.С.
Проверил:
Крощенко А.А.

Брест 2024

Цель работы: приобрести навыки применения паттернов проектирования при решении практических задач с использованием языка Java.

Вариант 17

- **Прочитать задания, взятые из каждой группы.**
- **Определить паттерн проектирования, который может использоваться при реализации задания. Пояснить свой выбор.**
- **Реализовать фрагмент программной системы, используя выбранный паттерн. Реализовать все необходимые дополнительные классы.**

Задание 1. Преподаватель. Класс должен обеспечивать одновременное взаимодействие с несколькими объектами класса Студент. Основные функции преподавателя – ПроверитьЛабораторнуюРаботу, ПровестиКонсультацию, ПринятьЭкзамен, ВыставитьОтметку, ПровестиЛекцию.

Для реализации задания был выбран паттерн Наблюдатель, поскольку он обеспечивает механизм подписки и оповещения об изменениях в объекте (в данном случае, в объекте преподавателя) для нескольких наблюдателей (студентов). Каждый студент подписывается на определенные события, такие как проверка лабораторной работы или проведение консультации, и получает уведомления об этих событиях, когда они происходят. Это позволяет обеспечить гибкость взаимодействия преподавателя с различными студентами и обеспечивает легкость добавления новых событий и наблюдателей без изменения основной логики.

Код программы:

task01.java

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

class TeacherEventDispatcher {
    Map<String, List<TeacherEventListener>> students = new HashMap<>();

    public TeacherEventDispatcher(String... events) {
        for (String event : events) {
            this.students.put(event, new ArrayList<>());
        }
    }

    public void subscribe (String event, TeacherEventListener student) {
        List<TeacherEventListener> subscribers = students.get(event);
        subscribers.add(student);
    }

    public void unsubscribe (String event, TeacherEventListener student) {
        List<TeacherEventListener> subscribers = students.get(event);
        subscribers.remove(student);
    }
```

```
    public void notify (String event, TeacherEventListener student) {
        List<TeacherEventListener> subscribers = students.get(event);
        for (TeacherEventListener subscriber : subscribers) {
            if(subscriber == student) {
                subscriber.update(event);
            }
        }
    }
}

class Teacher {
    public TeacherEventDispatcher events;

    public Teacher () {
        this.events = new TeacherEventDispatcher("Check lab work", "Conduct
consultation",
                "Take exam", "Grade exam", "Conduct lecture");
    }

    void checkLabwork (Student student) {
        events.notify("Check lab work", student);
        System.out.println("Teacher interacted with " + student.getName());
    }

    void conductConsultation (Student student) {
        events.notify("Conduct consultation", student);
        System.out.println("Teacher interacted with " + student.getName());
    }

    void takeExam (Student student) {
        events.notify("Take exam", student);
        System.out.println("Teacher interacted with " + student.getName());
    }

    void gradeExam (Student student) {
        events.notify("Grade exam", student);
        System.out.println("Teacher interacted with " + student.getName());
    }

    void conductLecture (Student student) {
        events.notify("Conduct lecture", student);
        System.out.println("Teacher interacted with " + student.getName());
    }
}

interface TeacherEventListener {
    void update(String eventType);
}

class Student implements TeacherEventListener {
    private String name;

    public Student(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public void update(String eventType) {
        System.out.println("Student " + name + " received notification: " +
```

```
eventType);
    }
}

public class task01 {
    public static void main(String[] args) {
        Teacher teacher = new Teacher();
        Student student1 = new Student("Ivan");
        Student student2 = new Student("Maria");

        teacher.events.subscribe("Check lab work", student1);
        teacher.events.subscribe("Check lab work", student2);
        teacher.events.subscribe("Conduct consultation", student1);
        teacher.events.subscribe("Take exam", student2);
        teacher.events.subscribe("Grade exam", student1);
        teacher.events.subscribe("Conduct lecture", student2);

        teacher.checkLabwork(student1);
        teacher.conductConsultation(student1);
        teacher.takeExam(student2);
        teacher.gradeExam(student1);
        teacher.conductLecture(student2);
    }
}
```

Результат программы:

```
Student Ivan received notification: Check lab work
Teacher interacted with Ivan
Student Ivan received notification: Conduct consultation
Teacher interacted with Ivan
Student Maria received notification: Take exam
Teacher interacted with Maria
Student Ivan received notification: Grade exam
Teacher interacted with Ivan
Student Maria received notification: Conduct lecture
Teacher interacted with Maria
```

Задание 2. ДУ автомобиля. Реализовать иерархию автомобилей для конкретных производителей и иерархию средств дистанционного управления. Автомобили должны иметь присущие им атрибуты и функции. ДУ имеет три основные функции – удаленная активация сигнализации, удаленное открытие/закрытие дверей и удаленный запуск двигателя. Эти функции должны отличаться по своим характеристикам для различных устройств ДУ.

Паттерн "Мост" был выбран для реализации данного задания, потому что он позволяет разделять абстракцию и реализацию, позволяя им изменяться независимо друг от друга. В данном случае абстракция представлена классом Car, который определяет основные функции автомобиля, такие как активация сигнализации, управление дверьми и запуск двигателя. Реализация представлена классами BMWFunctionality и AudiFunctionality, которые определяют конкретные реализации

этих функций для каждого конкретного производителя автомобиля. Использование паттерна "Мост" позволяет легко добавлять новые производители автомобилей или изменять функциональность существующих, не затрагивая саму абстракцию.

Код программы:

task02.java

```
interface CarFunctionality {
    void honk();
    void manipulateDoors();
    void startEngine();
}

class BMWFunctionality implements CarFunctionality {
    @Override
    public void honk() {
        System.out.println("BMW is sounding its horn!");
    }

    @Override
    public void manipulateDoors() {
        System.out.println("Your BMW's doors have been adjusted.");
    }

    @Override
    public void startEngine() {
        System.out.println("BMW engine is roaring!");
    }
}

class AudiFunctionality implements CarFunctionality {
    @Override
    public void honk() {
        System.out.println("Audi is beeping its horn!");
    }

    @Override
    public void manipulateDoors() {
        System.out.println("Your Audi's doors are now secured.");
    }

    @Override
    public void startEngine() {
        System.out.println("Audi's engine is purring!");
    }
}

abstract class Car {
    protected CarFunctionality functionality;

    public Car(CarFunctionality functionality) {
        this.functionality = functionality;
    }

    abstract void activateSignaling();
    abstract void manipulateDoors();
    abstract void startEngine();
}
```

```
class BMWCar extends Car {
    public BMWCar(CarFunctionality functionality) {
        super(functionality);
    }

    @Override
    void activateSignaling() {
        functionality.honk();
    }

    @Override
    void manipulateDoors() {
        functionality.manipulateDoors();
    }

    @Override
    void startEngine() {
        functionality.startEngine();
    }
}

class AudiCar extends Car {
    public AudiCar(CarFunctionality functionality) {
        super(functionality);
    }

    @Override
    void activateSignaling() {
        functionality.honk();
    }

    @Override
    void manipulateDoors() {
        functionality.manipulateDoors();
    }

    @Override
    void startEngine() {
        functionality.startEngine();
    }
}

public class task02 {
    public static void main(String[] args) {
        Car bmw = new BMWCar(new BMWFunctionality());
        System.out.println("BMW Client:");
        bmw.activateSignaling();
        bmw.manipulateDoors();
        bmw.startEngine();

        System.out.println("====================================");

        Car audi = new AudiCar(new AudiFunctionality());
        System.out.println("Audi Client:");
        audi.activateSignaling();
        audi.manipulateDoors();
        audi.startEngine();
    }
}
```

Результат программы:

```
BMW Client:
BMW is sounding its horn!
Your BMW's doors have been adjusted.
BMW engine is roaring!
==================================
Audi Client:
Audi is beeping its horn!
Your Audi's doors are now secured.
Audi's engine is purring!
```

Задание 3. Проект «Пиццерия». Реализовать формирование заказ(а)ов, их отмену, а также повторный заказ с теми же самыми позициями.

Паттерн "Команда" был выбран для реализации данного задания, потому что он позволяет инкапсулировать запрос как объект, что позволяет передавать запросы в качестве аргументов, сохранять их в истории, отменять и повторять операции. В данном случае каждая команда (CreateOrderCommand, CancelOrderCommand, RepeatOrderCommand) инкапсулирует определенное действие над заказом (создание, отмена, повторение), что позволяет легко расширять функциональность системы без изменения ее основной структуры.

Код программы:

task03.java

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

interface Command {
    void execute();
}

class CreateOrderCommand implements Command {
    private Order order;
    private List<MenuItem> itemsToAdd;

    public CreateOrderCommand(Order order, List<MenuItem> itemsToAdd) {
        this.order = order;
        this.itemsToAdd = itemsToAdd;
    }

    @Override
    public void execute() {
        order.addItems(itemsToAdd);
        System.out.println("New order has been created: " + order);
    }
}

class CancelOrderCommand implements Command {
    private Order order;
```

```
    public CancelOrderCommand(Order order) {
        this.order = order;
    }

    @Override
    public void execute() {
        order.cancel();
        System.out.println("Order has been cancelled: " + order);
    }
}

class RepeatOrderCommand implements Command {
    private OrderHistory orderHistory;
    private int orderIndex;

    public RepeatOrderCommand(OrderHistory orderHistory, int orderIndex) {
        this.orderHistory = orderHistory;
        this.orderIndex = orderIndex;
    }

    @Override
    public void execute() {
        Order orderToRepeat = orderHistory.getOrder(orderIndex);
        Order repeatedOrder = new Order();
        repeatedOrder.addItems(orderToRepeat.getItems());
        System.out.println("Repeated order has been created: " + repeatedOrder);
    }
}

class Order {
    private List<MenuItem> items;

    public Order() {
        this.items = new ArrayList<>();
    }

    public void addItems(List<MenuItem> itemsToAdd) {
        items.addAll(itemsToAdd);
    }

    public void cancel() {
        items.clear();
    }

    public List<MenuItem> getItems() {
        return items;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("Order: ");
        if (!items.isEmpty()) {
            for (MenuItem item : items) {
                sb.append(item.toString()).append(", ");
            }
            sb.setLength(sb.length() - 2);
        } else {
            sb.append("Empty");
        }
        return sb.toString();
    }
}
```

```java
class OrderHistory {
    private List<Order> orders;

    public OrderHistory() {
        this.orders = new ArrayList<>();
    }

    public void addOrder(Order order) {
        orders.add(order);
    }

    public Order getOrder(int index) {
        return orders.get(index);
    }
}

class MenuItem {
    private String name;
    private double price;

    public MenuItem(String name, double price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public String toString() {
        return name + " - $" + price;
    }
}

class Waiter {
    private List<Command> orders;

    public Waiter() {
        this.orders = new ArrayList<>();
    }

    public void takeOrder(Command command) {
        orders.add(command);
    }

    public void placeOrders() {
        for (Command command : orders) {
            command.execute();
        }
        orders.clear();
    }
}

public class task03 {
    public static void main(String[] args) {
        MenuItem pepperoni = new MenuItem("Pepperoni", 10.99);
        MenuItem margarita = new MenuItem("Margarita", 5.99);
        MenuItem hawaiian = new MenuItem("Hawaiian", 12.99);
        MenuItem veggie = new MenuItem("Veggie", 8.99);

        Order order = new Order();

        Waiter waiter = new Waiter();

        List<MenuItem> itemsToAdd = new ArrayList<>();
        itemsToAdd.add(pepperoni);
        itemsToAdd.add(margarita);
```

```
        itemsToAdd.add(hawaiian);
        itemsToAdd.add(veggie);
        Command createOrderCommand = new CreateOrderCommand(order, itemsToAdd);

        Command cancelOrderCommand = new CancelOrderCommand(order);

        OrderHistory orderHistory = new OrderHistory();
        orderHistory.addOrder(order);
        Command repeatOrderCommand = new RepeatOrderCommand(orderHistory, 0);

        waiter.takeOrder(createOrderCommand);
        waiter.takeOrder(repeatOrderCommand);
        waiter.takeOrder(cancelOrderCommand);

        waiter.placeOrders();
    }
}
```

Результат программы:

```
New order has been created: Order: Pepperoni - $10.99, Margarita - $5.99, Hawaiian - $12.99, Veggie - $8.99
Repeated order has been created: Order: Pepperoni - $10.99, Margarita - $5.99, Hawaiian - $12.99, Veggie - $8.99
Order has been cancelled: Order: Empty
```

Вывод: приобрели навыки применения паттернов проектирования при решении практических задач с использованием языка Java.