

Министерство образования Республики Беларусь  
Учреждение образования  
«Брестский государственный технический университет»  
Кафедра ИИТ

Лабораторная работа №6  
По дисциплине: «СПП»

Выполнила:  
студентка 3 курса  
группы ПО-8  
Гордейчук М.В.  
Проверил:  
Крощенко А.А.

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

### **Вариант 7**

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.

**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.

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

### **Ход работы:**

#### **Задание 1:**

Паттерн проектирования: Singleton — порождающий паттерн, который гарантирует существование только одного объекта определённого класса, а также позволяет достигаться до этого объекта из любого места программы.

#### **Текст программы:**

```
abstract class Person {  
    String name;  
    String surname;  
    int age;  
}  
  
class Teacher extends Person {  
    private static Teacher teacher;  
    public static Teacher getInstance(String name, String surname, int age) { //singleton  
        if (teacher == null) {  
            teacher = new Teacher(name, surname, age);  
        }  
    }  
}
```

```

    }
    return teacher;
}
Teacher(String name, String surname, int age) {
    this.name = name;
    this.surname = surname;
    this.age = age;
}
void print() {
    System.out.println("Teacher data:\n" +
        "Name: " + name + "\nSurname: " + surname + "\nAge: " + age);
}
void checkLabWork() { //проверить лаб раб
    System.out.println("Mr. " + this.surname + ": Ready. I just checked your
laboratory work. It's OK.");
}
void makeConsultation() { //провести консультацию
    System.out.println("Mr. " + this.surname + ": Ok I'll make a consultation");
}
void takeExam() { //принять экзамен
    System.out.println("Mr. " + this.surname + ": OK. Ready.");
}
void setMark(Student student) { //выставить отметку
    int mark = (int) (Math.random() * 10);
    student.setMark(mark);
    student.print();
}
void giveLecture() { //провести лекцию
    System.out.println("Mr. " + this.surname + ": Ok I'll give you a lecture");
}
}
class Student extends Person {
    private int mark;
    Student(String name, String surname, int age) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }
}

```

```

    }
    void setMark(int mark) {
        this.mark = mark;
    }
    void teacherCheckLabWork() { //проверить лаб раб
        Teacher teacher = Teacher.getInstance("", "", 0);
        System.out.println("Student " + name + ": Mr. " + teacher.surname + " please
check my laboratory work.");
        teacher.checkLabWork();
    }
    void teacherMakeConsultation() { //провести консультацию
        Teacher teacher = Teacher.getInstance("", "", 0);
        System.out.println("Student " + name + ": Mr. " + teacher.surname + " please
make a consultation.");
        teacher.makeConsultation();
    }
    void teacherTakeExam() { //принять экзамен
        Teacher teacher = Teacher.getInstance("", "", 0);
        System.out.println("Student " + name + ": Mr. " + teacher.surname + " please
take an exam.");
        teacher.takeExam();
    }
    void teacherSetMark() { //выставить отметку
        Teacher teacher = Teacher.getInstance("", "", 0);
        System.out.println("Student " + name + ": Mr. " + teacher.surname + " please
set my mark.");
        teacher.setMark(this);
    }
    void teacherGiveLecture() { //провести лекцию
        Teacher teacher = Teacher.getInstance("", "", 0);
        System.out.println("Student " + name + ": Mr. " + teacher.surname + " please
give a lecture.");
        teacher.giveLecture();
    }
    void print() {
        Teacher teacher = Teacher.getInstance("", "", 0);
        System.out.println(teacher.surname + " " + teacher.name +

```

```

        " rated " + name + " " + surname + ". Mark - " + mark);
    }
}

public class Main {
    public static void main(String[] args) {
        Student student1 = new Student("Nicolay", "Qwerty", 21);
        Student student2 = new Student("Grigoriy", "Tree", 22);
        Student student3 = new Student("Evgeniy", "Lucky", 19);
        Student[] students = {student1, student2, student3};
        Teacher teacher = Teacher.getInstance("Ivan", "Ivanov", 39);
        teacher.print();
        student1.teacherGiveLecture();
        student2.teacherCheckLabWork();
        student3.teacherMakeConsultation();
        student3.teacherTakeExam();
        student3.teacherSetMark();
    }
}

```

### Результат:

```

Teacher data:
Name: Ivan
Surname: Ivanov
Age: 39
Student Nicolay: Mr. Ivanov please give a lecture.
Mr. Ivanov: Ok I'll give you a lecture
Student Grigoriy: Mr. Ivanov please check my laboratory work.
Mr. Ivanov: Ready. I just checked your laboratory work. It's OK.
Student Evgeniy: Mr. Ivanov please make a consultation.
Mr. Ivanov: Ok I'll make a consultation
Student Evgeniy: Mr. Ivanov please take an exam.
Mr. Ivanov: OK. Ready.
Student Evgeniy: Mr. Ivanov please set my mark.
Ivanov Ivan rated Evgeniy Lucky. Mark - 7

```

### Задание 2:

Паттерн проектирования: Bridge — структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

### Текст программы:

```

Remote device1 = new RCSecondType(car1);
    car1.printCharacteristic();
    device1.doorOpening();

```

```

        device1.engineStart();
        device1.doorClosing();
        device1.signalingActivation();
    }
}

interface Car {
    void open();
    void close();
    void signaling();
    void start();
    void printCharacteristic();
    String getType();
}

class BMWCar implements Car {
    private int year;
    private int cost;
    private String country;
    BMWCar(int year1, int cost1, String country1) {
        year = year1;
        cost = cost1;
        country = country1;
    }
    @Override
    public void open() {
        System.out.println("BMW car is opened");
    }
    @Override
    public void close() {
        System.out.println("BMW car is closed");
    }
    @Override
    public void signaling() {
        System.out.println("You activate signaling in BMW car");
    }
    @Override
    public void start() {
        System.out.println("BMW car starts to move");
    }
}

```

```

    }
    @Override
    public void printCharacteristic() {
        System.out.println("BMW characteristic:");
        System.out.println("Year: " + year);
        System.out.println("Cost: " + cost + "$");
        System.out.println("Country: " + country);
    }
    @Override
    public String getType() {
        return "BMW";
    }
}

class MercedesCar implements Car {
    private int year;
    private int cost;
    private String country;
    MercedesCar(int year1, int cost1, String country1) {
        year = year1;
        cost = cost1;
        country = country1;
    }
    @Override
    public void open() {
        System.out.println("Mercedes car is opened");
    }
    @Override
    public void close() {
        System.out.println("Mercedes car is closed");
    }
    @Override
    public void signaling() {
        System.out.println("You activate signaling in Mercedes car");
    }
    @Override
    public void start() {
        System.out.println("Mercedes car starts to move");
    }
}

```

```

    }
    @Override
    public void printCharacteristic() {
        System.out.println("Mercedes characteristic:");
        System.out.println("Year: " + year);
        System.out.println("Cost: " + cost + "$");
        System.out.println("Country: " + country);
    }
    @Override
    public String getType() {
        return "Mercedes";
    }
}

interface Remote {
    void signalingActivation();
    void doorOpening();
    void doorClosing();
    void engineStart();
}

class RCFirstType implements Remote {
    private Car car;
    RCFirstType(Car car) {
        this.car = car;
    }
    @Override
    public void signalingActivation() {
        System.out.println("Remote control device of first type is going to activate
        signaling in " + car.getType() + " car");
        car.signaling();
    }
    @Override
    public void doorOpening() {
        System.out.println("Remote control device of first type is going to open " + car.getType() + "
        car");
        car.open();
    }
    @Override

```



```

public void doorClosing() {
    System.out.println("Remote control device of first type is going to close " +
car.getType() + " car");
    car.close();
}
@Override
public void engineStart() {
    System.out.println("Remote control device of first type is going to start engine
in " + car.getType() + " car");
    car.start();
}
}
class RCSecondType implements Remote {
    private Car car;
    RCSecondType(Car car) {
        this.car = car;
    }
    @Override
    public void signalingActivation() {
        System.out.println("Remote control device of second type is going to activate
signaling in " + car.getType() + " car");
        car.signaling();
    }
    @Override
    public void doorOpening() {
        System.out.println("Remote control device of second type is going to open " +
car.getType() + " car");
        car.open();
    }
    @Override
    public void doorClosing() {
        System.out.println("Remote control device of second type is going to close " +
car.getType() + " car");
        car.close();
    }
    @Override
    public void engineStart() {

```

```

        System.out.println("Remote control device of second type is going to start engine in "
+ car.getType() + " car");
        car.start();
    }
}

```

### Результат:

```

Mercedes characteristic:
Year: 2018
Cost: 50$
Country: Germany
Remote control device of first type is going to open Mercedes car
Mercedes car is opened
Remote control device of first type is going to start engine in Mercedes car
Mercedes car starts to move
Remote control device of first type is going to close Mercedes car
Mercedes car is closed
Remote control device of first type is going to activate signaling in Mercedes car
You activate signaling in Mercedes car
BMW characteristic:
Year: 2016
Cost: 30$
Country: Switzerland
Remote control device of second type is going to open BMW car
BMW car is opened
Remote control device of second type is going to start engine in BMW car
BMW car starts to move
Remote control device of second type is going to close BMW car
BMW car is closed
Remote control device of second type is going to activate signaling in BMW car
You activate signaling in BMW car

```

### Задание 3:

Паттерн проектирования: Command — поведенческий паттерн, позволяющий заворачивать запросы или простые операции в отдельные объекты. Это позволяет откладывать выполнение команд, выстраивать их в очереди, а также хранить историю и делать отмену.

#### Текст программы:

```

import java.util.ArrayList;
import java.util.Stack;
public class Main {
    public static void main(String[] args) {
        Pizzeria pizzeria = new Pizzeria();
        AddCommand add = new AddCommand(pizzeria);
        RemoveCommand remove = new RemoveCommand(pizzeria);
        pizzeria.executeCommand(add, Dish.soup);
        pizzeria.executeCommand(add, Dish.pizza);
        pizzeria.executeCommand(add, Dish.paste);
        pizzeria.printOrder();
        pizzeria.executeCommand(remove, Dish.soup);
        pizzeria.printOrder();
        pizzeria.undo();
        pizzeria.printOrder();
    }
}

```

```

    }
}
class Pizzeria {
    private ArrayList<Dish> order;
    private CommandHistory history;
    Pizzeria() {
        order = new ArrayList<Dish>();
        history = new CommandHistory();
    }
    void add(Dish dish) {
        this.order.add(dish);
    }
    void remove(Dish dish) {
        int index = this.order.indexOf(dish);
        this.order.remove(index);
    }
    void undo() {
        if (history.isEmpty()) return;
        Command command = history.pop();
        if (command != null) {
            command.undo();
        }
    }
    void restore(ArrayList<Dish> order) {
        this.order = new ArrayList<Dish>(order);
    }
    void executeCommand(Command command, Dish dish) {
        command.execute(dish);
        history.push(command);
    }
    Snapshot createSnapshot() {
        Snapshot snapshot = new Snapshot(this, order);
        return snapshot;
    }
    void printOrder() {
        System.out.println("Your order: ");
        for (Dish dish: order) {

```

```

        switch (dish) {
            case soup:
                System.out.println("Soup");
                break;
            case paste:
                System.out.println("Paste");
                break;
            case pizza:
                System.out.println("Pizza");
                break;
            case pancakes:
                System.out.println("Pancakes");
                break;
            case sushi:
                System.out.println("Sushi");
                break;
        }
    }
}

enum Dish {
    soup,
    paste,
    pizza,
    pancakes,
    sushi,
}

class Snapshot {
    private ArrayList<Dish> order;
    private Pizzeria pizzeria;
    Snapshot(Pizzeria pizzeria, ArrayList<Dish> order) {
        this.order = new ArrayList<Dish>(order);
        this.pizzeria = pizzeria;
    }

    public void restore() {
        pizzeria.restore(order);
    }
}

```

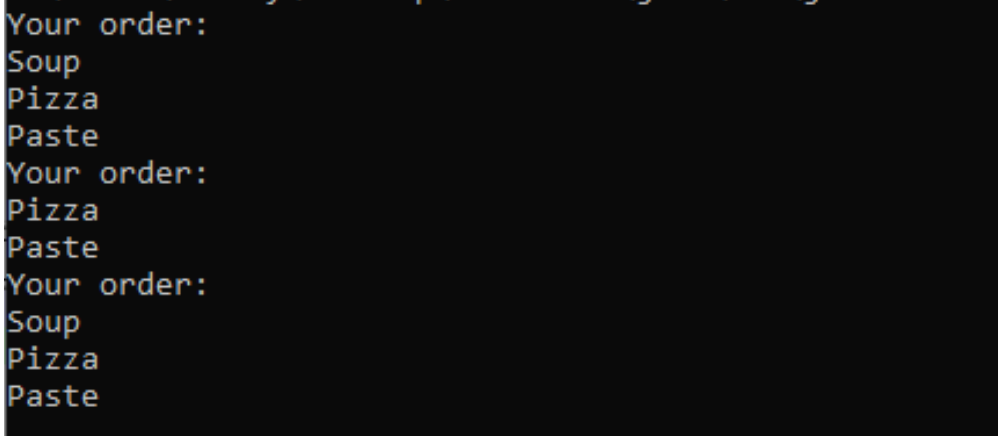
```

}
abstract class Command {
    private Snapshot backup;
    public Pizzeria pizzeria;
    Command(Pizzeria pizzeria) {
        this.pizzeria = pizzeria;
    }
    void makeBackup(){
        this.backup = pizzeria.createSnapshot();
    }
    void undo() {
        if (backup != null) {
            backup.restore();
        }
    }
    public abstract void execute(Dish dish);
}
class AddCommand extends Command {
    public AddCommand(Pizzeria pizzeria) {
        super(pizzeria);
    }
    @Override
    public void execute(Dish dish) {
        pizzeria.add(dish);
    }
}
class RemoveCommand extends Command {
    public RemoveCommand(Pizzeria pizzeria) {
        super(pizzeria);
    }
    @Override
    public void execute(Dish dish) {
        makeBackup();
        pizzeria.remove(dish);
    }
}
class CommandHistory {

```

```
        private Stack<Command> history = new Stack<Command>();
        public void push(Command c) {
            history.push(c);
        }
        public Command pop() {
            return history.pop();
        }
        public boolean isEmpty() {
            return history.isEmpty();
        }
    }
}
```

### Результат:



```
Your order:
Soup
Pizza
Paste
Your order:
Pizza
Paste
Your order:
Soup
Pizza
Paste
```

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