设计模式实验 21-23

一、实验目的

1.结合实例，熟练绘制设计模式结构图。
2.结合实例，熟练使用 Java 语言实现设计模式。
3.通过本实验，理解每一种设计模式的模式动机，掌握模式结构，学习如何使用代码实现这些设计模式。

二、实验要求

1.结合实例，绘制设计模式的结构图。
2.使用 Java 语言实现设计模式实例，代码运行正确。

三、实验内容

1. 代理模式：在某电子商务系统中，为了提高查询性能，需要将一些频繁查询的数据保存到内存的辅 助存储对象中（提示：可使用 Map 实现）。用户在执行查询操作时，先判断辅助存储对象 中是否存在待查询的数据，如果不存在，则通过数据操作对象查询并返回数据，然后将效据 保存到辅助存储对象中，否则直接返回存储在辅助存储对象中的数据。现采用代理模式中的 缓冲代理实现该功能，要求绘制对应的类图并编程模拟实现。

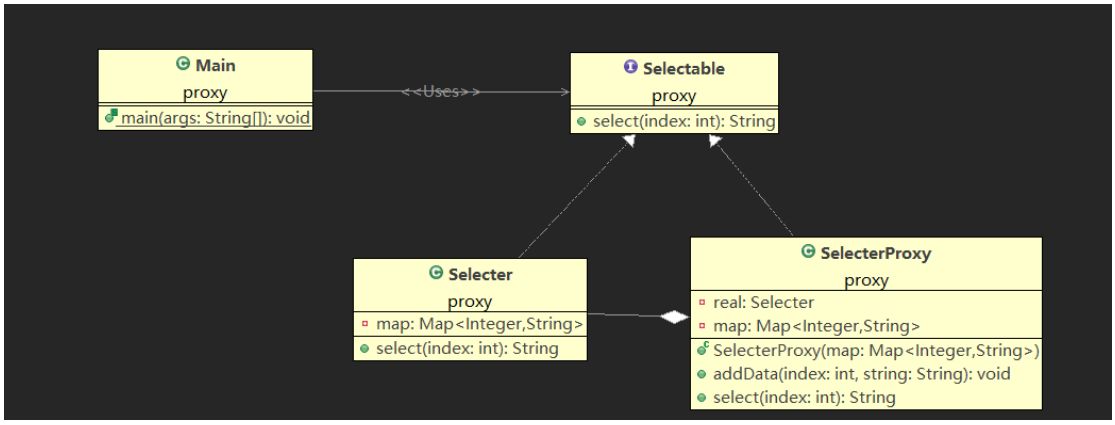
2. 命令模式：某灯具厂商要生产一个智能灯具遥控器，该遥控器具有 5 个可编程的插槽，这 5 个开关可以通过蓝牙技术控制 5 个不同房间灯光的打开和关闭， 用户可以自行设置每一个开关所对应的房间。现采用命令模式实现该智能遥控器的软件部分，绘制对应的类图并编程模拟实现。

3. 解释器模式：某软件公司要为数据库备份和同步开发一套简单的数据库同步指令，通过指令可以对数 据库中的数据和结构进行备份。例如，输入指令“ COPY VIEW FROM srcDB TO desDB ”，表 示将数据库 srcDB 中的所有视图（View）对象都拷贝至数据库 desDB ； 输人指令“ MOVETABLE Student FROM srcDB TO desDB ”，表示将数据库 srcDB 中的 Student 表移 动至数据库 desDB 。现使用解释器模式来设计并编程模拟实现该数据库同步指令系统。

四、实验结果

需要提供设计模式实例的结构图（类图）和实现代码。

1. 代理模式

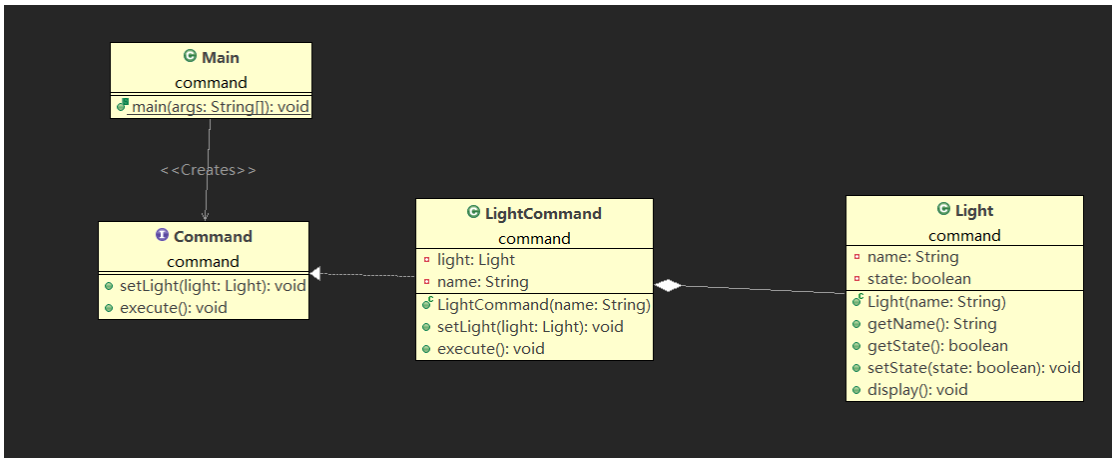


代码实现：

```
public interface Selectable {

    public String select(int index);

}

public class SelecterProxy implements Selectable{

    private Selecter real;
    private Map<Integer, String> map = new HashMap<Integer, String>();

    public SelecterProxy(Map<Integer, String> map) {
        this.map = map;
    }

    public void addData(int index, String string) {
        map.put(index, string);
    }

    @Override
    public synchronized String select(int index) {
        if (map.get(index) != null) {
            System.out.println(index + ", " + map.get(index));
            return map.get(index);
        }
        else {
            real = new Selecter();
            String string = real.select(index);
            addData(index, string);
            System.out.println(index + ", " + map.get(index));
            return string;
        }
    }
```

```
}

public class Selecter implements Selectable{

    private Map<Integer, String> map = new HashMap<Integer, String>();


    @Override
    public String select(int index) {
        System.out.println("Connection to DataBase and found it");
        return "StringData";
    }

}

public class Main {

    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<Integer, String>();
        map.put(1, "one");
        map.put(2, "two");
        Selectable selectable = new SelecterProxy(map);
        selectable.select(1);
        selectable.select(2);
        selectable.select(3);

    }

}
```

2. 命令模式



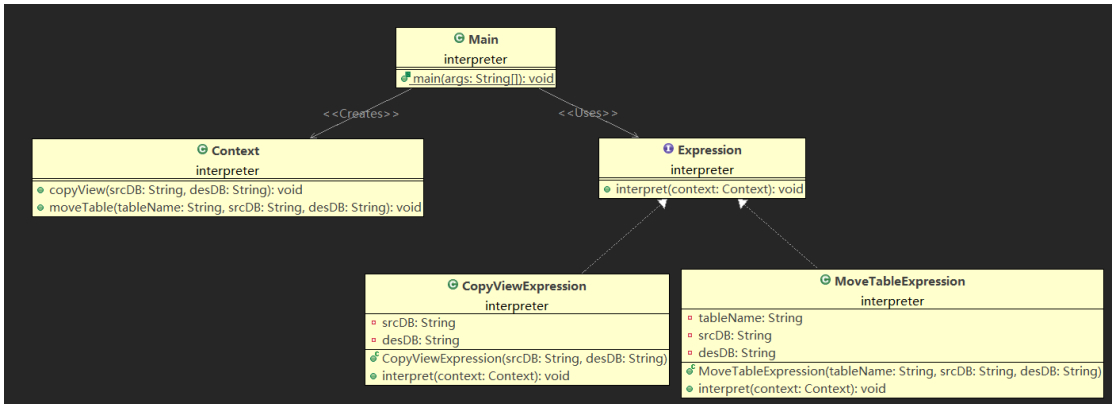
代码实现：

```
public interface Command {

    public void setLight(Light light);
    public void execute();

}
```

```
public class LightCommand implements Command{

    private Light light;
    private String name;

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

    @Override
    public void setLight(Light light) {
        this.light = light;
        System.out.println("Command " + name + " controls Light " +
light.getName());
    }

    @Override
    public void execute() {
        boolean s = light.getState();
        s = !s;
        light.setState(s);

    }

}

public class Light {

    private String name;
    private boolean state = false;

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

    public String getName() {
        return name;
    }

    public boolean getState() {
        return state;
    }

    public void setState(boolean state) {
        this.state = state;
    }

    public void display() {
        if(state) {
            System.out.println("Light " + name + "is ON");
        }
        else {
            System.out.println("Light " + name + "is OFF");
        }
    }

}
```

```
public class Main {

    public static void main(String[] args) {
        Command c1 = new LightCommand("c1");
        Command c2 = new LightCommand("c2");
        Command c3 = new LightCommand("c3");

        Light l1 = new Light("l1");
        Light l2 = new Light("l2");
        Light l3 = new Light("l3");

        c1.setLight(l1);
        c2.setLight(l2);
        c3.setLight(l3);

        c1.execute();

        l1.display();
        l2.display();
        l3.display();
    }

}
```

3. 解释器模式



代码实现：

```
public interface Expression {

    public void interpret(Context context);

}

class CopyViewExpression implements Expression {
    private String srcDB;
    private String desDB;

    public CopyViewExpression(String srcDB, String desDB) {
        this.srcDB = srcDB;
        this.desDB = desDB;
    }
```

```
    @Override
    public void interpret(Context context) {
        context.copyView(srcDB, desDB);
    }
}

class MoveTableExpression implements Expression {
    private String tableName;
    private String srcDB;
    private String desDB;

    public MoveTableExpression(String tableName, String srcDB, String desDB) {
        this.tableName = tableName;
        this.srcDB = srcDB;
        this.desDB = desDB;
    }

    @Override
    public void interpret(Context context) {
        context.moveTable(tableName, srcDB, desDB);
    }
}

public class Context {

    public void copyView(String srcDB, String desDB) {
        System.out.println("Doing the Copying views from " + srcDB + " to " +
desDB);
    }

    public void moveTable(String tableName, String srcDB, String desDB) {
        System.out.println("Doing the Moving table " + tableName + " from " + srcDB
+ " to " + desDB);
    }


}

public class Main {

    public static void main(String[] args) {
        Context context = new Context();
        String instruction = "COPY VIEW FROM srcDB TO desDB";

        String[] parts = instruction.split(" ");
        Expression expression = null;

        if (parts.length == 6) {
            if (parts[0].equals("COPY") && parts[1].equals("VIEW") &&
parts[2].equals("FROM") && parts[4].equals("TO")) {
                expression = new CopyViewExpression(parts[3], parts[5]);
            }
        } else if (parts.length == 7) {
            if (parts[0].equals("MOVETABLE") && parts[2].equals("FROM") &&
parts[4].equals("TO")) {
                expression = new MoveTableExpression(parts[1], parts[3], parts[6]);
            }
        }
```

```
        if (expression != null) {
            expression.interpret(context);
        } else {
            System.out.println("Invalid instruction");
        }
    }
}
```

五、实验小结

请总结本次实验的体会，包括学会了什么、遇到哪些问题、如何解决这些问题

以及存在哪些有待改进的地方。

通过实际编写代码，我加强了对代理者模式、命令模式、解释器模式三种设计模式的认识，了解了这三种设计模式在实际运用中的作用和意义。

命令模式中，遇到的问题：如何将设置电灯与遥控器的关系。
解决办法：将电灯的实例组合进入遥控器的实例中，通过遥控器的命令修改电灯的状态。