天津大学

设计模式（6）实验报告

学　　院　智算学部

专　　业　软件工程

学　　号　3019213043

姓　　名　刘京宗

设计模式实验（6）

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. 状态模式

在某网络管理软件中， TCP 连接（TCP Connection）具有建立（Established）、监听（Listening）、关闭（Closed）等多种状态，在不同的状态下 TCP 连接对象具有不同的行为，连接对象还可以从一个状态转换到另一个状态。当一个连接对象收到其他对象的请求时，它根据自身的当前状态做出不同的反应。现采用状态模式对 TCP 连接进行设计，绘制对应的类图并编程模拟实现。

2. 享元模式

某 OA 系统采用享元模式设计权限控制与管理模块，在该模块中，将与系统功能相对应的业务类设计为享元类并将相应的业务对象存储到享元池中（提示；可使用 Map 实现，key 为业务对象对应的权限编码， value 为业务对象）。用户身份验证成功后，系统通过存储在数据库中的该用户的权限编码集从享元池获取相应的业务对象并构建权限列表，在界面上显示用户所拥有的权限。根据以上描述，绘制对应的类图并编程模拟实现。

3. 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。

4. 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，每个插槽都有一个控制灯具的开关，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。

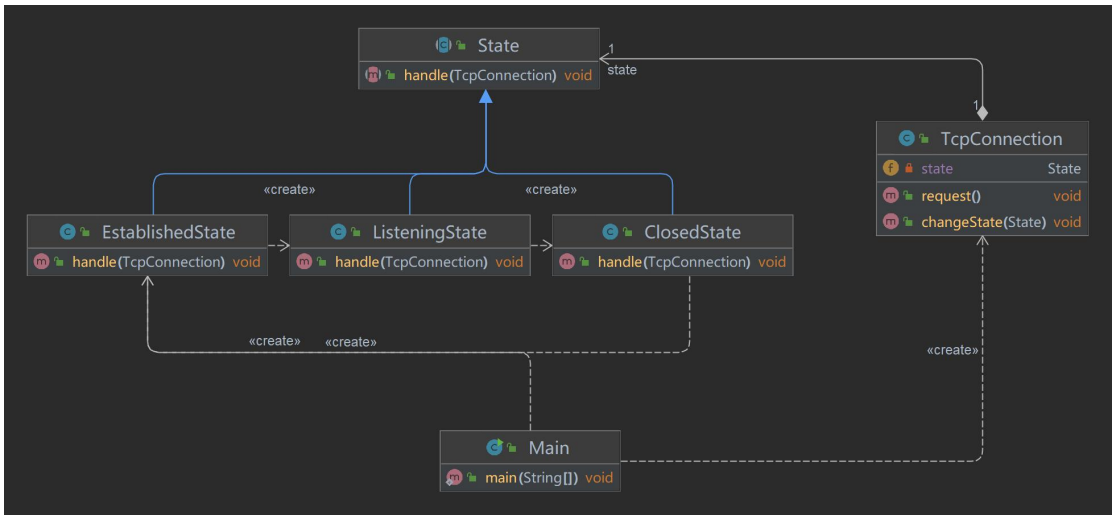
5. 解释器模式

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。

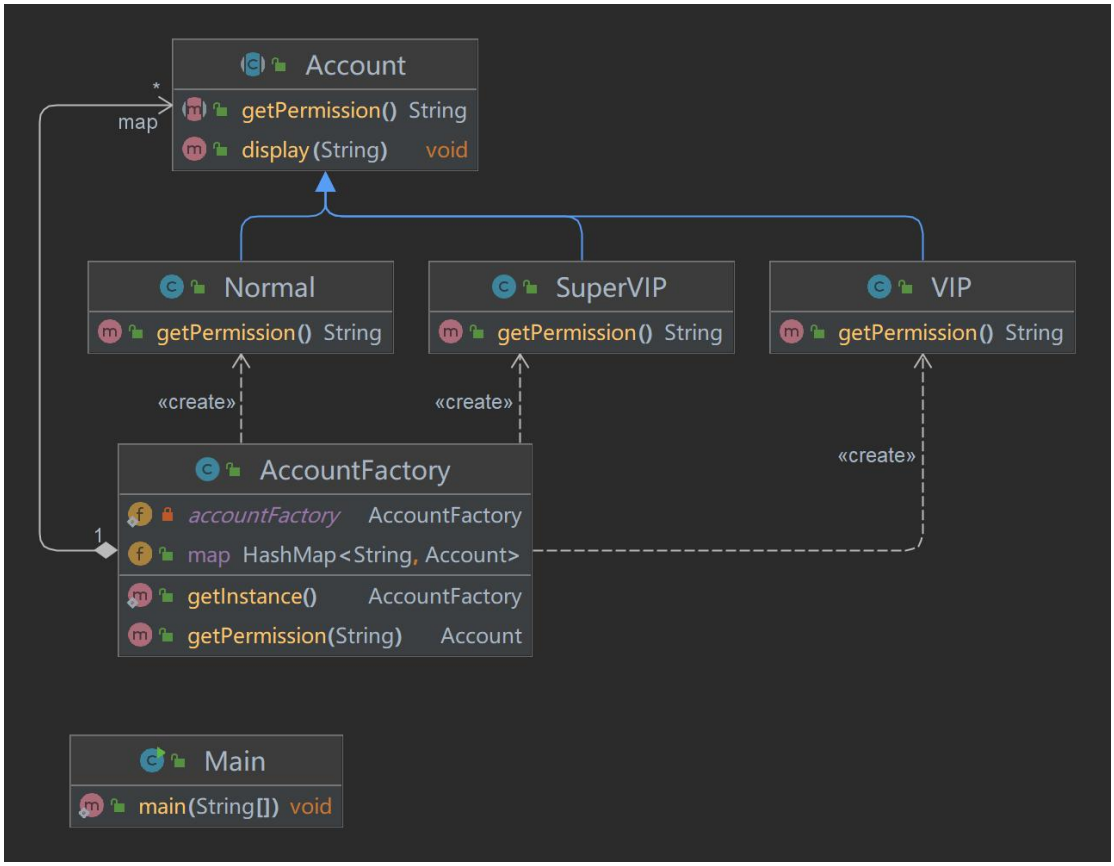
4.1 状态模式




```
public class TcpConnection {
    private State state;

    public TcpConnection(State state) {
        this.state = state;
    }

    public void request() {
        if (null != state) {
            this.state.handle(this);
        }
    }

    public void changeState(State state) {
        if (null != state) {
            this.state = state;
        }
    }
}
public   abstract class State {
    public abstract void handle(TcpConnection tcpConnection);
}
```

```
public class ClosedState extends State {
	public void handle(TcpConnection context) {
		System.out.println("handled by ClosedState");
		if (null != context) {
			context.changeState(new EstablishedState());
		}
	}
}
public class EstablishedState extends State {
	@Override
	public void handle(TcpConnection context) {
		System.out.println("handled by EstablishedState");
		if (null != context) {
			context.changeState(new ListeningState());
		}
	}
}
public class ListeningState extends State {
	@Override
	public void handle(TcpConnection context) {
		System.out.println("handled by ListeningState");
		if (null != context) {
			context.changeState(new ClosedState());
		}
	}
}
public class Main {
	public static void main(String[] args) {
		State state = new EstablishedState();
		TcpConnection context = new TcpConnection(state);
		context.request();
		context.request();
		context.request();
	}
}
```

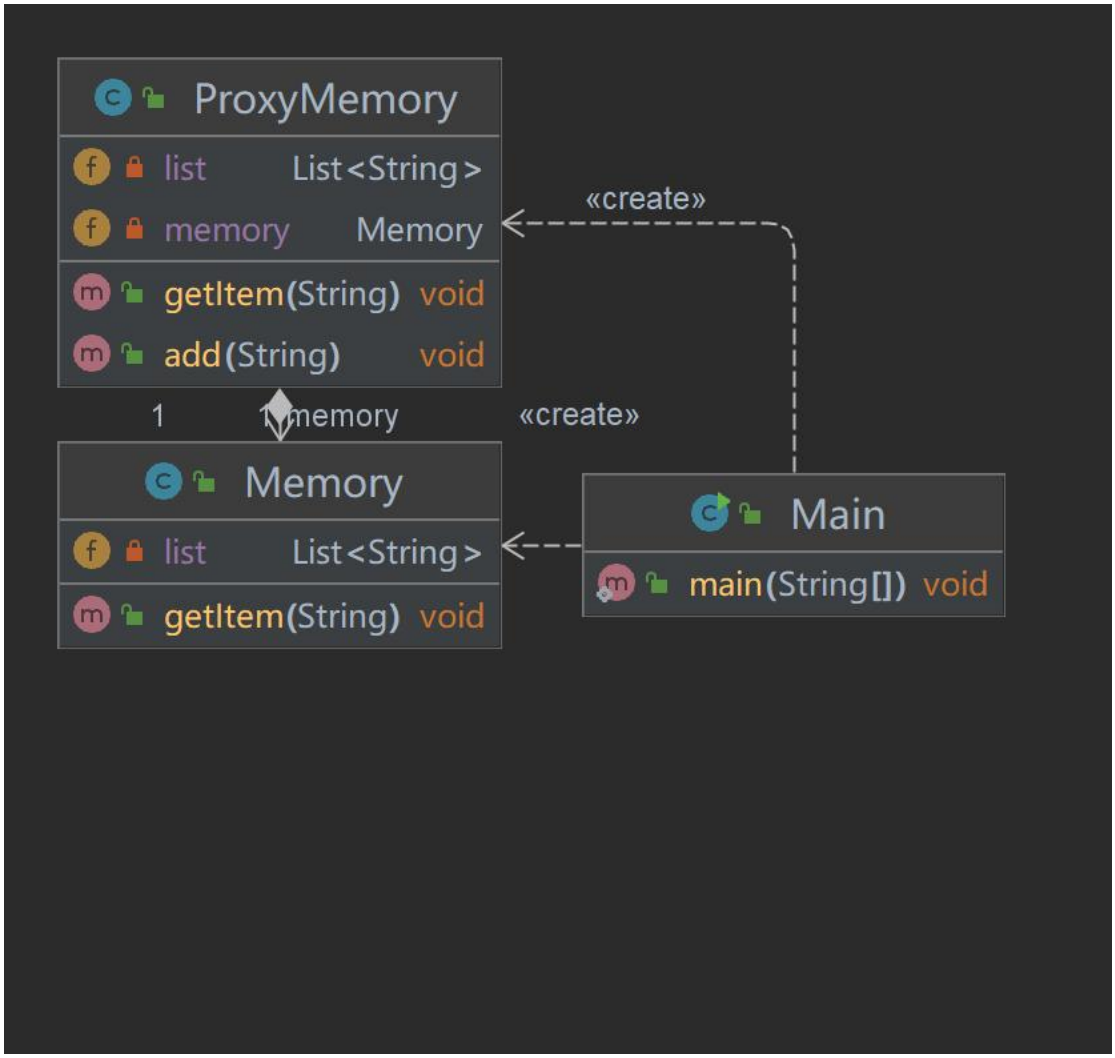
4.2 享元模式




```
public abstract class Account {
    public abstract String getPermission();

    public void display(String name) {
        System.out.println(name + "用户的权限是：" + getPermission());
    }
}
public class Normal extends Account {

    @Override
    public String getPermission() {
        return "身份：平民用户";
    }
}
public class SuperVIP extends Account {
    @Override
    public String getPermission() {
        return "身份：超级尊贵用户";
    }
}
public class VIP extends Account {
    @Override
```

```
    public String getPermission() {
        return "身份：尊贵用户";
    }

}
public class AccountFactory {
    public HashMap<String, Account> map;

    private AccountFactory() {
        map = new HashMap<String, Account>();
        map.put("01", new Normal());
        map.put("02", new VIP());
        map.put("03", new SuperVIP());
    }

    private static AccountFactory accountFactory = new AccountFactory();

    public static AccountFactory getInstance() {
        return accountFactory;
    }

    public Account getPermission(String code) {
        return map.get(code);
    }
}
public class Main {
    public static void main(String[] args) {
        AccountFactory factory = AccountFactory.getInstance();
        Account account1 = factory.getPermission("01");
        account1.display("张三");
        Account account2 = factory.getPermission("02");
        account2.display("李四");
        Account account3 = factory.getPermission("03");
        account3.display("王五");
    }
}
```

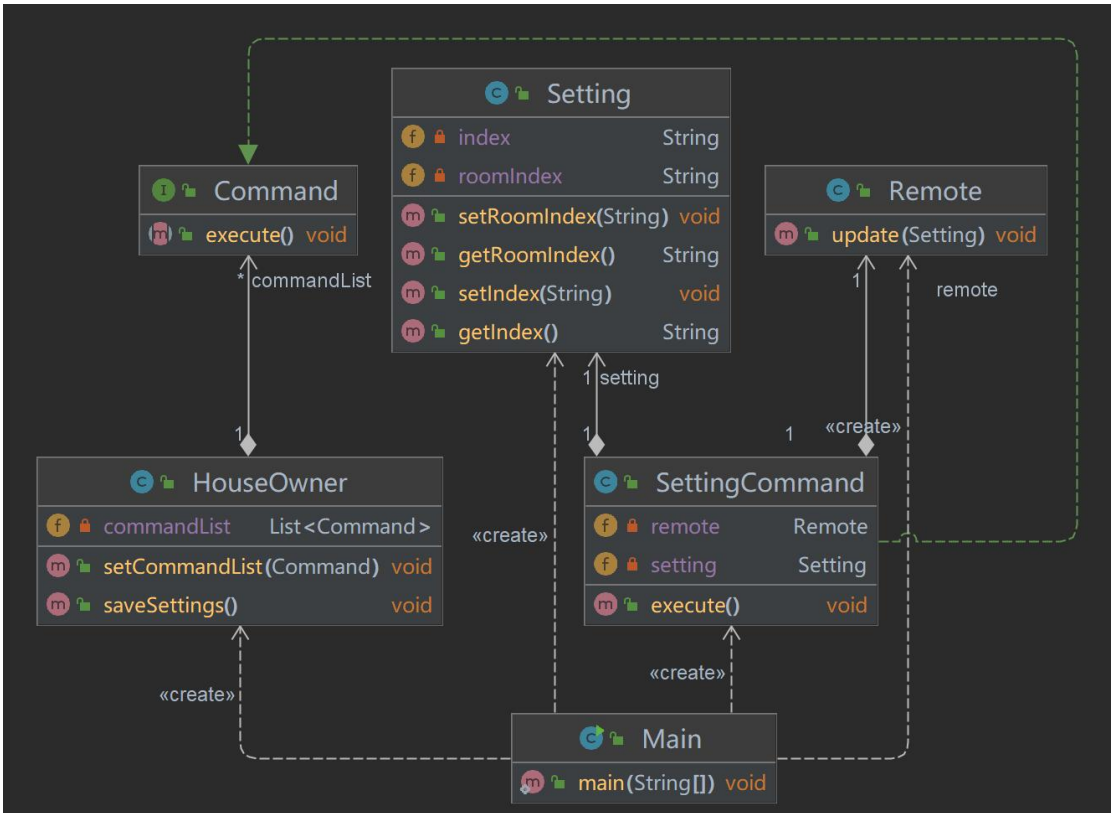
4.3 代理模式




```
public class Memory {
    public Memory(List list) {
        this.list = list;
    }

    private List<String> list = new ArrayList<>();


    public void getItem(String item) {
        if (list.contains(item)) {
            System.out.println("从内存中获取到" + item);
        }
    }
}
public class ProxyMemory {
    private Memory memory;
    private List<String> list = new ArrayList<>();
```

```
    public void add(String item) {
        list.add(item);
    }

    public ProxyMemory(Memory memory) {
        this.memory = memory;
    }

    public void getItem(String item) {
        if (list.contains(item)) {
            System.out.println("从缓冲代理内存中获取到" + item);
        } else {
            memory.getItem(item);
            list.add(item);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("元素 1");
        list.add("元素 2");
        list.add("元素 3");
        Memory memory = new Memory(list);
        ProxyMemory proxyMemoey = new ProxyMemory(memory);
        proxyMemoey.getItem("元素 1");
        proxyMemoey.getItem("元素 2");
        proxyMemoey.getItem("元素 1");
    }
}
```

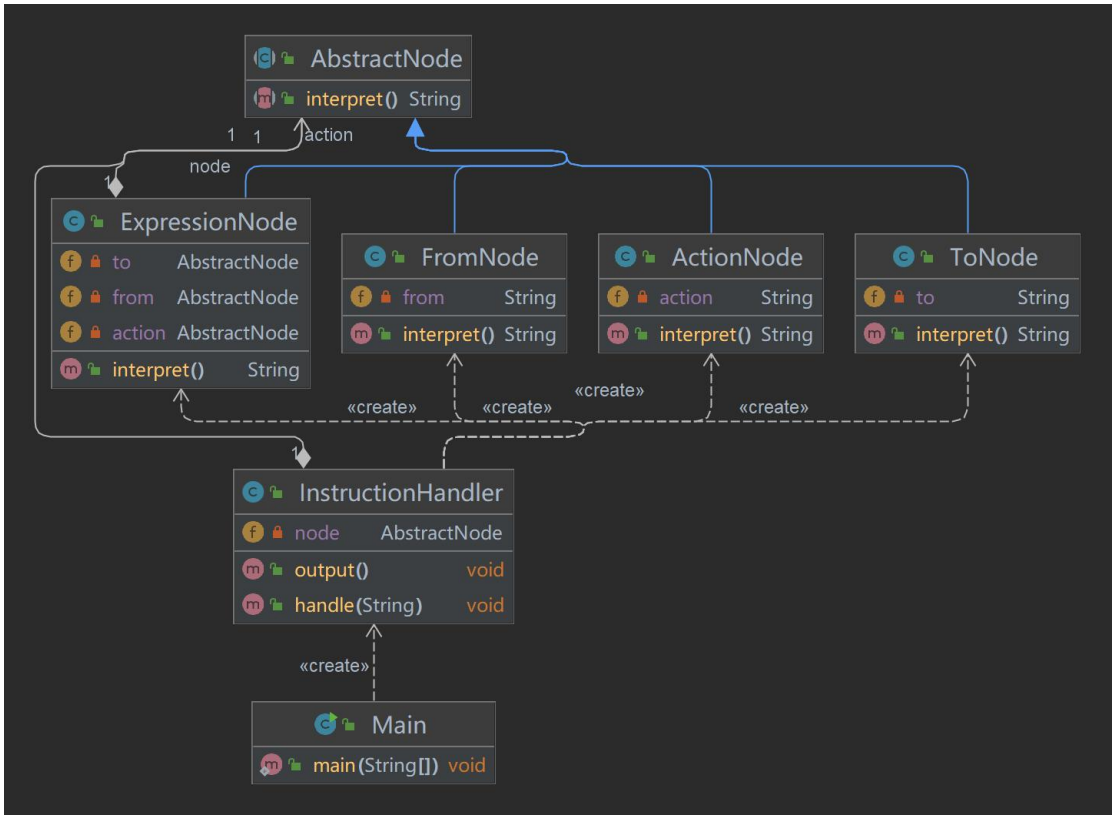
4.4 命令模式




```
public interface Command {
    void execute();
}
public class HouseOwner {
    private List<Command> commandList = new ArrayList<Command>();

    public void setCommandList(Command cmd) {
        commandList.add(cmd);
    }

    public void saveSettings() {
        System.out.println("分配好开关，保存配置");
        for (Command command : commandList) {
            if (command != null) {
                command.execute();
            }
        }
    }
}
public class Remote {
    public void update(Setting setting) {
        System.out.println("已保存配置：遥控器第" + setting.getIndex() + "个开关控制第" +
setting.getRoomIndex() + "个房间");
```

```
    }
}
public class Setting {
    private String index;
    private String roomIndex;

    public String getIndex() {
        return index;
    }

    public void setIndex(String index) {
        this.index = index;
    }

    public String getRoomIndex() {
        return roomIndex;
    }

    public void setRoomIndex(String roomIndex) {
        this.roomIndex = roomIndex;
    }
}
public class SettingCommand implements Command {
    private Remote remote;
    private Setting setting;

    public SettingCommand(Remote remote, Setting setting) {
        this.remote = remote;
        this.setting = setting;
    }

    @Override
    public void execute() {
        remote.update(setting);
    }
}
```

4.5 解释器模式




```
public abstract class AbstractNode {
    public abstract String interpret();
}
public class ActionNode extends AbstractNode {
    private String action;

    public ActionNode(String action) {
        this.action = action;
    }

    @Override
    public String interpret() {
        String[] wordList = action.trim().split("\\s");
        int wordCount = wordList.length;
        if (wordCount == 2) {
            return action + " *";
        }
        return action;
    }
}
public class ExpressionNode extends AbstractNode {
    private AbstractNode action;
    private AbstractNode from;
```

```
    private AbstractNode to;

    public ExpressionNode(AbstractNode action, AbstractNode from, AbstractNode to) {
        this.action = action;
        this.from = from;
        this.to = to;
    }

    @Override
    public String interpret() {
        return this.action.interpret() + " "
                + this.from.interpret() + " "
                + this.to.interpret();
    }
}
public class FromNode extends AbstractNode{
    private String from;

    public FromNode(String from) {
        this.from = from;
    }

    @Override
    public String interpret() {
        return    "FROM " + from;
    }
}
public class InstructionHandler {
    private AbstractNode node;

    public void handle(String instruction) {
        AbstractNode actionNode, fromNode, toNode, expressionNode;
        int fromIndex = instruction.indexOf(" FROM ");
        int toIndex = instruction.indexOf(" TO ");
        String action = instruction.substring(0, fromIndex);
        String from = instruction.substring(fromIndex + 6, toIndex);
        String to = instruction.substring(toIndex + 4);
        actionNode = new ActionNode(action);
        fromNode = new FromNode(from);
        toNode = new ToNode(to);
        expressionNode = new ExpressionNode(actionNode, fromNode, toNode);
        this.node = expressionNode;
    }
```

```
    public void output() {
        String result = node.interpret();
        System.out.println(result);
    }
}
public class ToNode extends AbstractNode{
    private String to;

    public ToNode(String to) {
        this.to = to;
    }

    @Override
    public String interpret() {
        return "TO " + to;
    }
}
public class Main {

    public static void main(String[] args) {
        String instruction = "COPY VIEW FROM dbA TO dbB";
        InstructionHandler instructionHandler = new InstructionHandler();
        instructionHandler.handle(instruction);
        instructionHandler.output();

        instruction = "MOVE TABLE Student FROM dbA TO dbB";
        instructionHandler.handle(instruction);
        instructionHandler.output();
    }

}
```

五、实验小结

通过这次实验，我熟悉了状态模式、享元模式、代理模式、命令模式、解释器模式这些设计模式，收益匪浅。最后，对李老师在课程上悉心讲解与辛勤付出表示衷心的感谢！