天津大学

设计模式（5）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（5）

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. 外观模式

某软件公司为新开发的智能手机控制与管理软件提供了一键备份功能，通过该功能可以将原本存储在手机中的通讯录、短信、照片、歌曲等资料一次性全部拷贝到移动存储个质（例如 MMC 卡或 SD 卡）中。在实现过程中需要与多个已有的类进行交互，例如通讯录管理类、短信管理类等。为了降低系统的耦合度，试使用外观模式来设计并编程模拟实现该一键备份功能。

2. 中介者模式

为了大力发展旅游业，某城市构建了一个旅游综合信息系统，该系统包括旅行社子系统（ravel companies Subsystem）、宾馆子系统（HotelsSubsystem）、餐厅于系统(Restaurants Subsystem)、机场子系统（Airport Subsystem）、旅游景点子系统（Tourism Attractions Subsystem）等多个子系统，通过该旅游综合信息系统，各类企业之间可实现信息共享，一家企业可以将客户信息传递给其他合作伙伴。例如，当一家旅行社有一些客户后，该旅行社可以将客户信息传送到宾馆子系统、餐厅子系统、机场子系统和旅游景点子系统；宾馆也可以将顾客信息传送到旅行社子系统、餐子系统、机场子系统和旅游景点子系统；机场也可以将乘客信息传送到旅行社子系统、宾馆子系统、餐厅子系统和旅游景点子系统。由于这些子系统之间存在较为复杂的交互关系，现采用中介者模式为该旅游综合信息系统提供一个高层设计，绘制对应的类图并编程模拟实现。

3. 观察者模式

某文字编辑软件须提供如下功能：在文本编辑窗口中包含一个可编辑文本区和 3 个文本信息统计区，用户可以在可编辑文本区对文本进行编辑操作，第一个文本信息统计区用于显示可编辑文本区中出现的单词总数量和字符总数量，第二个文本信息统计区用于显示可编辑文本区中出现的单词（去重后按照字典序排序），第三个文本信息统计区用于按照出现频次降序显示可编辑文本区中出现的单词以及每个单词出现的次数（例如 hello :5）。现采用观察者模式设计该功能，绘制对应的类图并编程模拟实现。

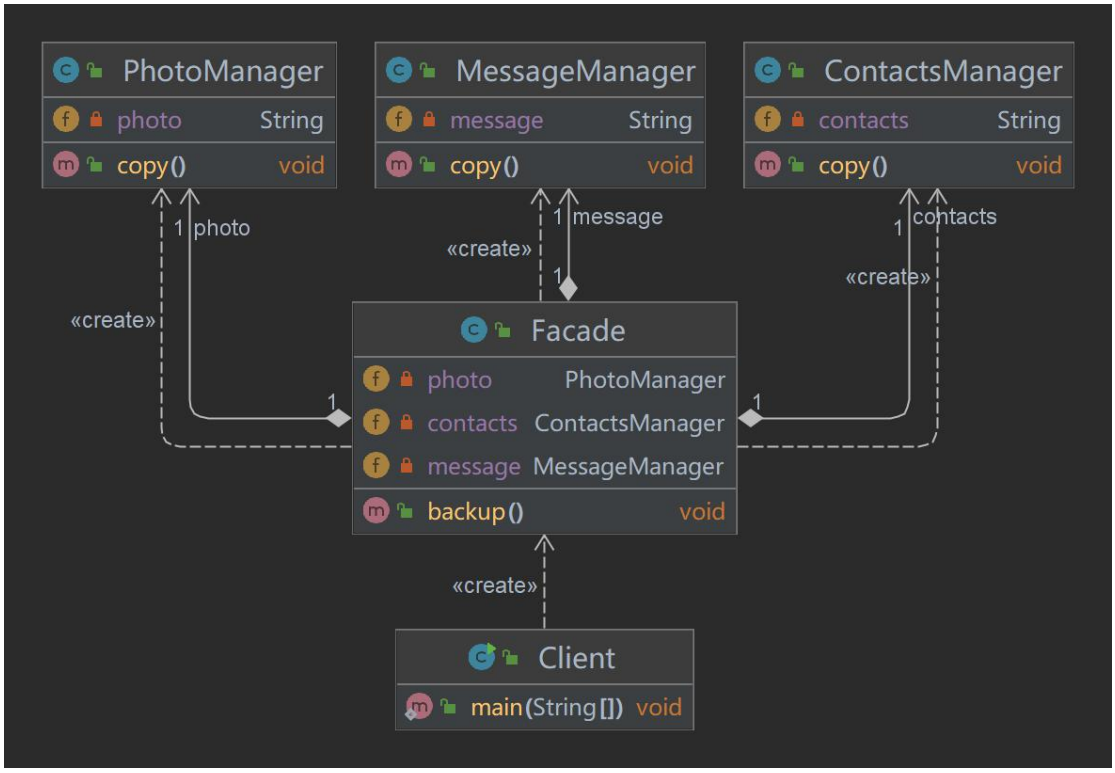
4. 备忘录模式

某文字编辑软件须提供撤销（Undo）和重做 / 恢复（Redo）功能，并且该软件可支持文档对象的多步撤销和重做。开发人员决定采用备忘录模式来实现该功能，在实现过程中引
人栈（Stack）作为数据结构。在实现时，可以将备忘录对象保存在两个栈中，一个栈包含用于实现撤销操作的状态对象，另一个栈包含用于实现重做操作的状态对象。在实现撤销操作时，会弹出撤销栈栈顶对象以获取前一个状态并将其设置给应用程序；同样，在实现重做操作时，会弹出重做栈栈顶对象以获取下一个状态并将其设置给应用程序。绘制对应的类图并编程模拟实现。

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。

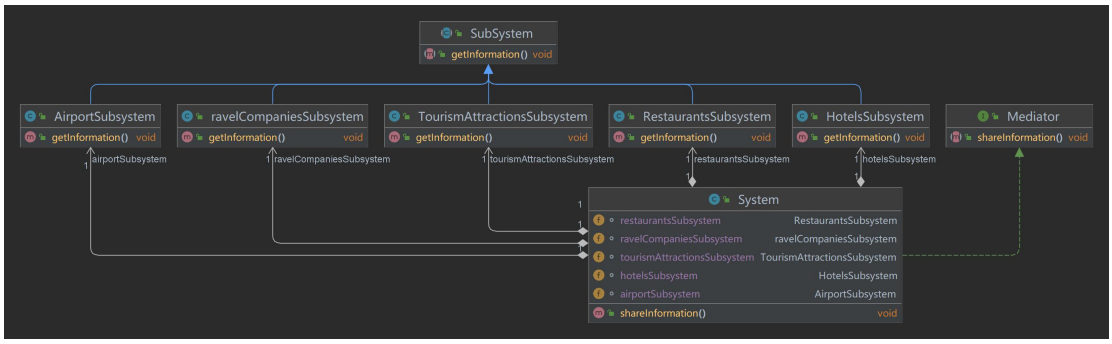
4.1 外观模式




```
public class ContactsManager {
    private String contacts;

    public ContactsManager(String contacts) {
        this.contacts = contacts;
    }

    public void copy() {
        System.out.println(contacts);
    }
}
public class MessageManager {
    private String message;
```

```
    public MessageManager(String message) {
        this.message = message;
    }

    public void copy() {
        System.out.println(message);
    }
}
public class PhotoManager {
    private String photo;

    public PhotoManager(String photo) {
        this.photo = photo;
    }

    public void copy() {
        System.out.println(photo);
    }
}
public class Facade {
    private ContactsManager contacts;
    private MessageManager message;
    private PhotoManager photo;

    public Facade() {
        this.contacts = new ContactsManager("通讯录");
        this.message = new MessageManager("短信");
        this.photo=new PhotoManager("照片");
    }

    public void backup(){
        contacts.copy();
        message.copy();
        photo.copy();
    }
}
public class Client {
    public static void main(String[] args) {
        Facade facade = new Facade();
        facade.backup();
    }
}
```

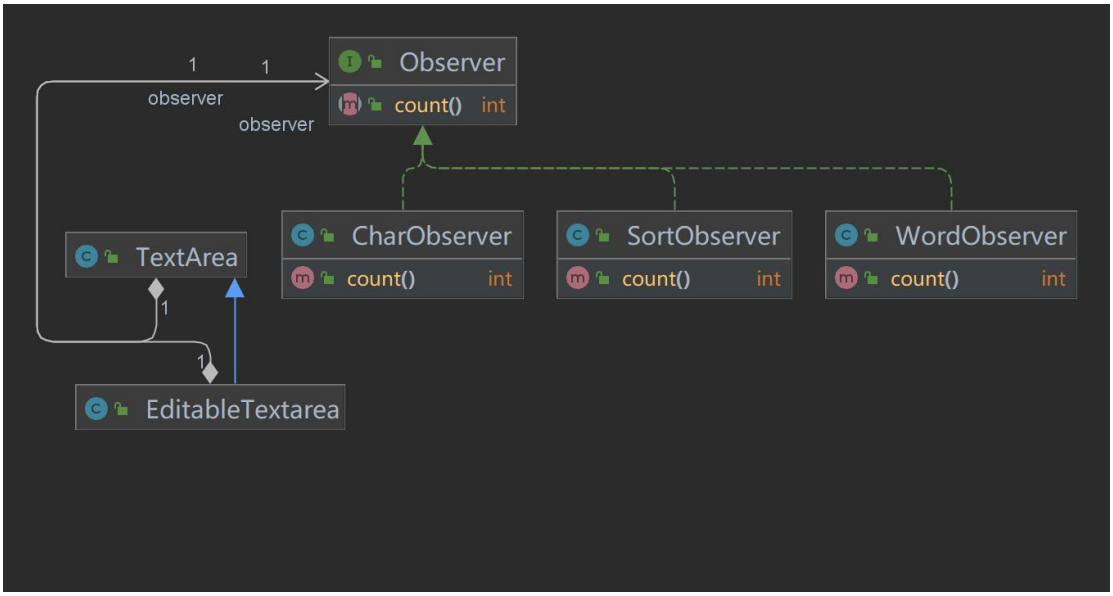
4.2 中介者模式




```
public class AirportSubsystem extends SubSystem{
    @Override
    public void getInformation() {

    }
}
public class HotelsSubsystem extends SubSystem{
    @Override
    public void getInformation() {

    }
}
public interface Mediator {
    public void shareInformation();
}
public class ravelCompaniesSubsystem extends SubSystem{
    @Override
    public void getInformation() {

    }
}
public class RestaurantsSubsystem extends SubSystem{
    @Override
    public void getInformation() {

    }
}
public abstract class SubSystem {
    public abstract void getInformation();
}
public class System implements Mediator {
    AirportSubsystem airportSubsystem;
    HotelsSubsystem hotelsSubsystem;
    ravelCompaniesSubsystem ravelCompaniesSubsystem;
```

```
    RestaurantsSubsystem restaurantsSubsystem;
    TourismAttractionsSubsystem tourismAttractionsSubsystem;

    @Override
    public void shareInformation() {

    }
}
public class TourismAttractionsSubsystem extends SubSystem{
    @Override
    public void getInformation() {

    }
}
```

4.3 观察者模式




```
public interface Observer {
    public int count();
}
public class TextArea {
    Observer observer;
}
public class EditableTextarea extends TextArea {
    Observer observer;
}
public class CharObserver implements Observer{
    @Override
    public int count() {
        return 0;
```

```
    }
}
public class SortObserver implements Observer {
    @Override
    public int count() {
        return 0;
    }
}
public class WordObserver implements Observer {
    @Override
    public int count() {
        return 0;
    }
}
```

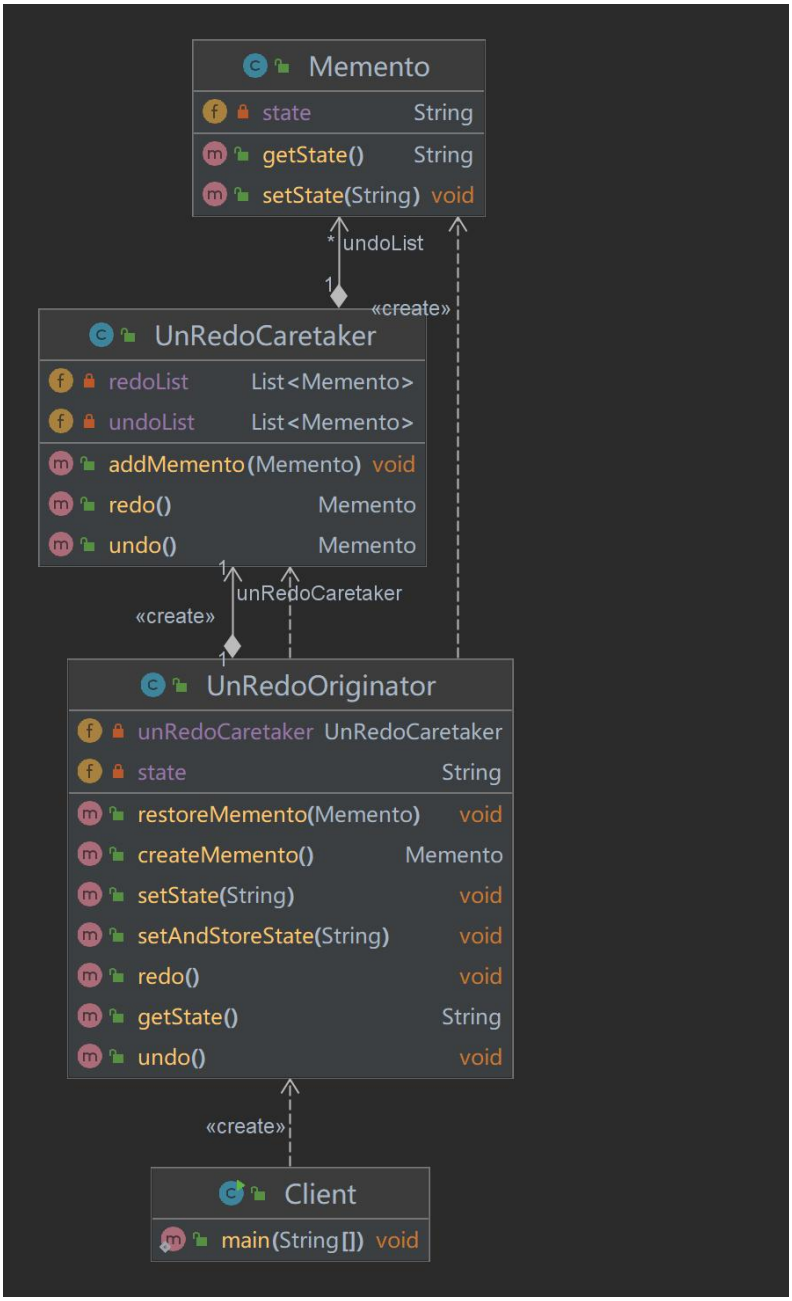
4.4 备忘录模式

```
public class Memento {
    private String state;

    public Memento(UnRedoOriginator unRedoOriginator) {
        this.state = unRedoOriginator.getState();
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }
}
public class UnRedoCaretaker {
    //      undo 操作使用的备忘录列表
    private List<Memento> undoList;
    //      redo 操作使用的备忘录列表
    private List<Memento> redoList;

    public UnRedoCaretaker() {
        undoList = new ArrayList<>();
        redoList = new ArrayList<>();
    }

    public void addMemento(Memento memento) {
//      undoList 添加新的备忘录之前，删除 redo 操作备忘录列表中的备忘录
        for (int i = redoList.size() - 1; i >= 0; i--) {
```

```
                redoList.remove(i);
            }
            undoList.add(memento);
        }

        /*undoList 中的最后一个 memento 代表当前状态*/
        public Memento undo() {
            Memento result = null;
            if (undoList.size() > 1) {
                /*获取 undoList 中的倒数第二个 memento，才是需要返回的状态*/
                result = undoList.get(undoList.size() - 2);
                /*将 undoList 中的表示当前状态的 memento，存储至 redoList 中 */
                redoList.add(undoList.get(undoList.size() - 1));
                /*将 undoList 中的表示当前状态的 memento 移除*/
                undoList.remove(undoList.size() - 1);
            } else {
                System.out.println("fail to undo operation!");
            }
            return result;
        }

        public Memento redo() {
            Memento result = null;
            if (redoList.size() > 0) {
                result = redoList.get(redoList.size() - 1);
                undoList.add(result);
                redoList.remove(redoList.size() - 1);
            } else {
                System.out.println("fail to redo operation!");
            }
            return result;
        }
}
public class UnRedoOriginator {
        private String state;
        private UnRedoCaretaker unRedoCaretaker;

        public UnRedoOriginator() {
            unRedoCaretaker=new UnRedoCaretaker();
        }

        public Memento createMemento(){
            return new Memento(this);
        }
```

```
    public void restoreMemento(Memento memento){
        this.state=memento.getState();
    }
    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }
    public void setAndStoreState(String state){
        this.setState(state);
        unRedoCaretaker.addMemento(this.createMemento());
    }
    public void undo(){
        Memento memento=unRedoCaretaker.undo();
        if (memento!=null){
            System.out.println(memento.getState());
        }
    }
    public void redo(){
        Memento memento=unRedoCaretaker.redo();
        if (memento!=null) {
            System.out.println(memento.getState());
        }
    }
}
public class Client {
    public static void main(String[] args) {
        UnRedoOriginator originator = new UnRedoOriginator();
        System.out.println("执行的操作顺序:add sub mul div");
        originator.setAndStoreState("add");
        originator.setAndStoreState("sub");
        originator.setAndStoreState("mul");
        originator.setAndStoreState("div");

        System.out.println("第一次 undo 操作：");
        originator.undo();
        System.out.println("第二次 undo 操作：");
        originator.undo();
        System.out.println("第一次 redo 操作：");
        originator.redo();
        System.out.println("第三次 undo 操作：");
```

```
        originator.undo();
        System.out.println("第四次 undo 操作：");
        originator.undo();
        System.out.println("第二次 redo 操作：");
        originator.redo();
    }
}
```



五、实验小结

通过这次实验，我熟悉了外观模式、中介者模式、观察者模式、备忘录模式这些设计模式，收益匪浅。