设计模式补充

分析题

1、画图小程序

在一个画图的小程序中，你已经实现了绘制点（Point）、直线（Line）、方块（Square）等图形的功能。而且为了让客户程序在使用的时候不用去关心它们的不同，还使用了一个抽象类图形（Shape）来规范这些图形的接口（Draw）。现在你要来实现圆的绘制，这时你发现在系统其他的地方已经有了绘制圆的实现类（XCircle），但同时你又发现已实现类XCircle的绘制方法（DrawIt）和你在抽象类Shape中规定的方法名称（Draw）不一样！这可怎么办?

方案一：修改已实现类XCircle的方法名DrawIt为Draw，是否合适？为什么?

方案二：修改抽象类Shape的方法名Draw为DrawIt，是否合适？为什么?

方案三：请你给出其它的解决方法。

解析:

方案一：修改已实现类XCircle的方法名DrawIt为Draw，是否合适？为什么?

不合适。可能会导致其它依赖于类XCircle的应用不能正常运行，因而要修改所有使用类XCircle的程序。

方案二：修改抽象类Shape的方法名Draw为DrawIt，是否合适？为什么?

不合适。如果修改抽象类Shape的的方法名，就要修改所有图形类的方法以及已有的引用。

方案三：请你给出其它的解决方法。

使用适配器模式。

```
class Triangle :public Shape
{
public:
  Triangle(XTriangle * ptr):xt(ptr)
  {  }
  virtual void Draw ( )
  {  xt-> DrawIt( );  }
private:
  XTriangle * xt;
};
```

2、员工类重构

在某公司财务系统的初始设计方案中存在如图1-2 所示的 Employee类，该类包含员工编号(ID)、姓名(name)、年龄(age)、性别(gender)、薪水(salary)、每月工作时数(workHoursPerMonth)、每月请假天数(leaveDaysPerMonth)等属性。该公司的员工包括全职和兼职两类，其中每月工作时数用于存储兼职员工每个月工作的小时数，每月请假天数用于存储全职员工每个月请假的天数。系统中两类员工计算工资的方法也不一样，全职员工按照工作日数计算工资，兼职员工按照工作时数计算工资，因此在 Employee类中提供了两个方法 calculateSalaryByDays()和 calculateSalaryByHours(),分别用于按照天数和时数计算工资，此外，还提供了方法 displaySalary()用于显示工资。

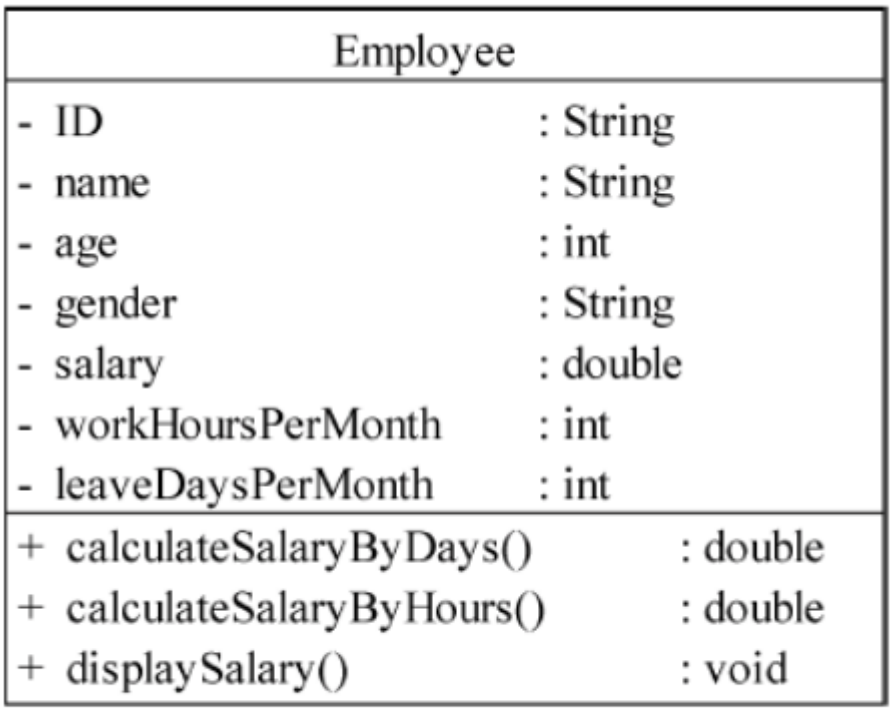
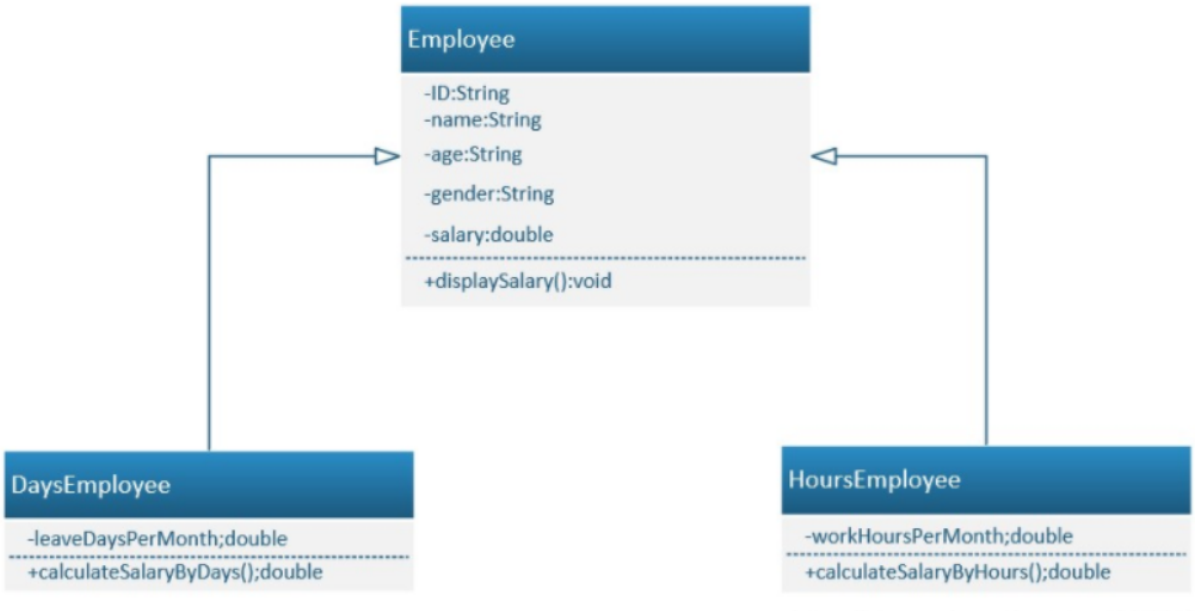


图 1-2　Employee 类初始类图

试采用所学面向对象设计原则分析图1-2 中Employee类存在的问题并对其进行重构，绘制重构之后的类图。

解析：

违反了单一职责原则



3、迪米特法则重构

在某图形界面中存在如下代码片段，组件类之间有较为复杂的相互引用关系：

【代码部分省略】

如果在上述系统中增加一个新的组件类，则必须修改与之交互的其他组件类的源代码，将导致多个类的源代码需要修改。

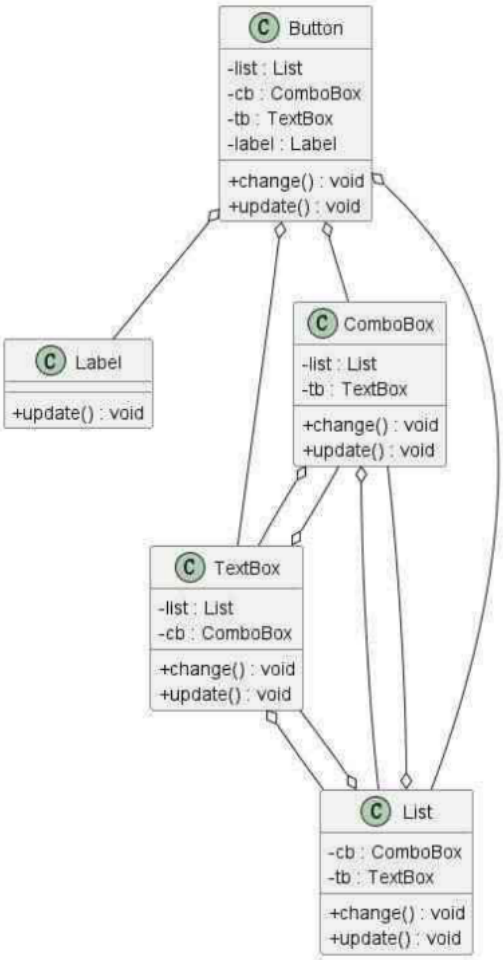
基于上述代码，请结合所学知识完成以下两道练习题：

(1)绘制上述代码对应的类图。

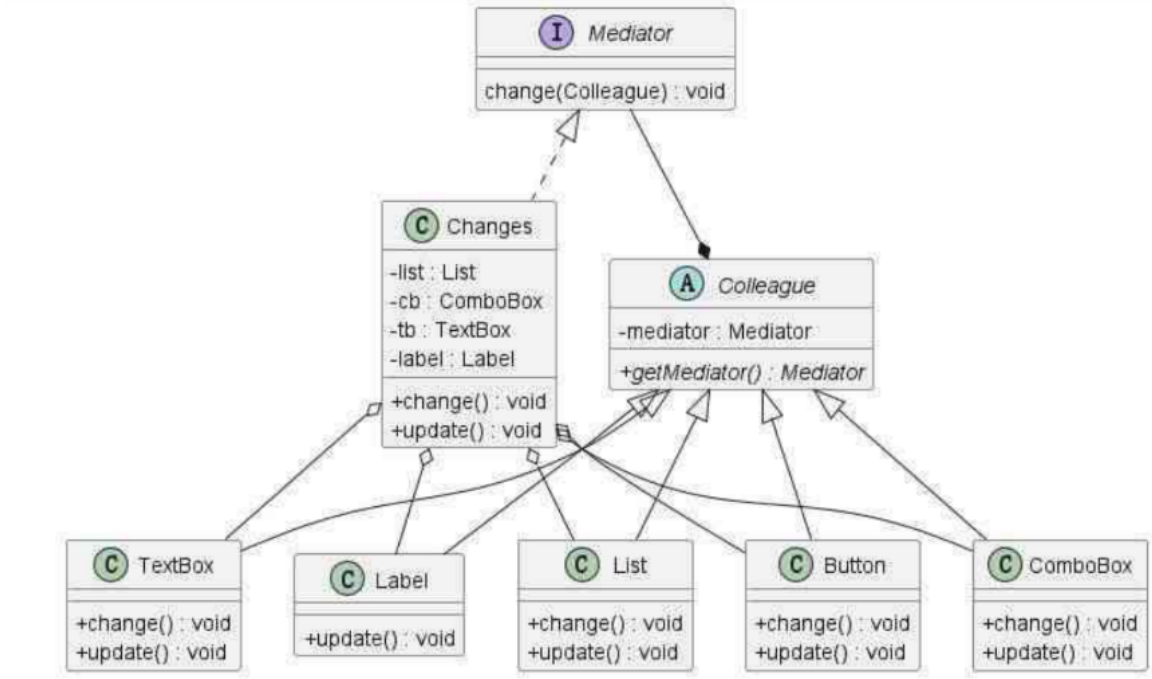
(2)根据迪米特法则对所绘制的类图进行重构，以降低组件之间的耦合度，绘制重构后的类图。

解析：

（1）


Button
-list : List
-cb : ComboBox
-tb : TextBox
-label : Label
+change() : void
+update() : void
Label
+update() : void
ComboBox
-list : List
-tb : TextBox
+change() : void
+update() : void
TextBox
-list : List
-cb : ComboBox
+change() : void
+update() : void
List
-cb : ComboBox
-tb : TextBox
+change() : void
+update() : void

(2)


Mediator
change(Colleague) : void
Changes
-list : List
-cb : ComboBox
-tb : TextBox
-label : Label
+change() : void
+update() : void
Colleague
-mediator : Mediator
+getMediator() : Mediator
TextBox
+change() : void
+update() : void
Label
+update() : void
List
+change() : void
+update() : void
Button
+change() : void
+update() : void
ComboBox
+change() : void
+update() : void

4、矢量图模板重构

在某图形库 API中提供了多种矢量图模板，用户可以基于这些矢量图创建不同的图形，图形库设计人员设计的初始类图如图1-3 所示。

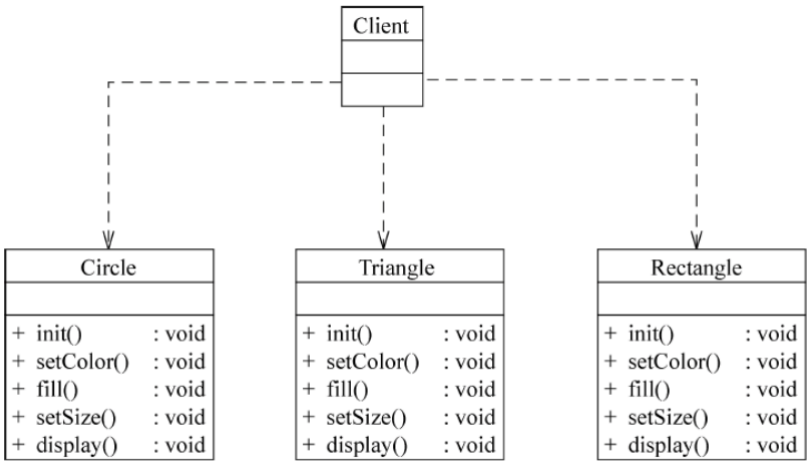


图 1-3　图形库初始类图

在该图形库中，每个图形类(例如 Circle、Triangle 等)的init()方法用于初始化所创建的图形，setColor()方法用于给图形设置边框颜色，fill()方法用于给图形设置填充颜色，setSize()方法用于设置图形的大小，display()方法用于显示图形。用户在客户类(Client)中使用该图形库时发现存在如下问题：

(1)由于在创建窗口时每次只需要使用图形库中的一种图形，因此在更换图形时需要修改客户类源代码。

(2)在图形库中增加并使用新的图形时，需要修改客户类源代码。

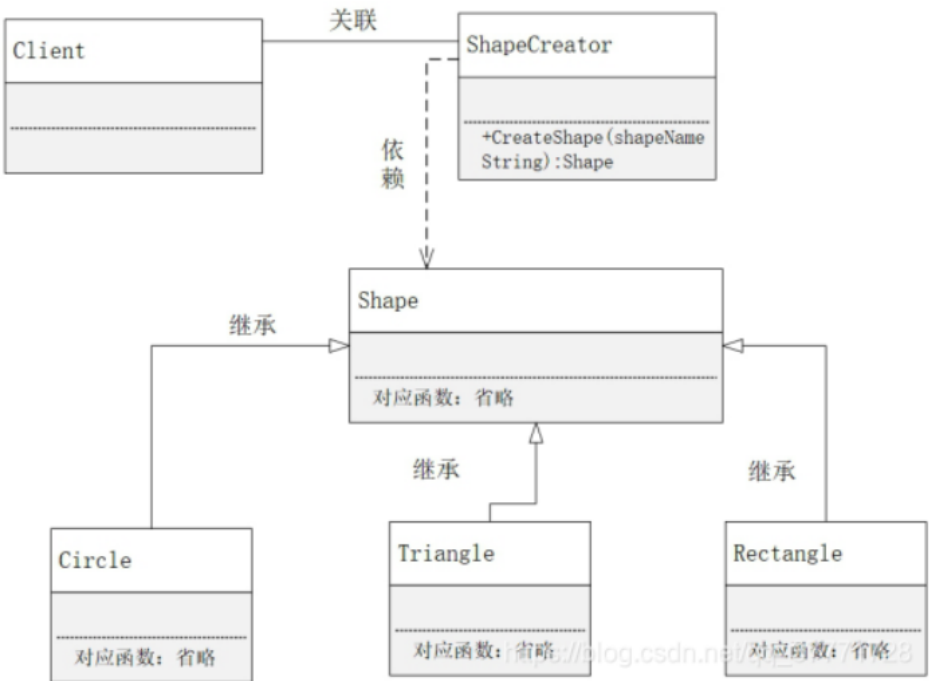
(3)客户类在每次使用图形对象之前需要先创建图形对象，有些图形的创建过程较为复杂，导致客户类代码冗长且难以维护。现需要根据面向对象设计原则对该系统进行重构，要求如下：

(1)隔离图形的创建和使用，将图形的创建过程封装在专门的类中，客户类在使用图形时无须直接创建图形对象，甚至不需要关心具体图形类类名。

(2)客户类能够方便地更换图形或使用新增图形，无须针对具体图形类编程，符合开闭原则。

请绘制重构后的结构图(类图)。

解析：

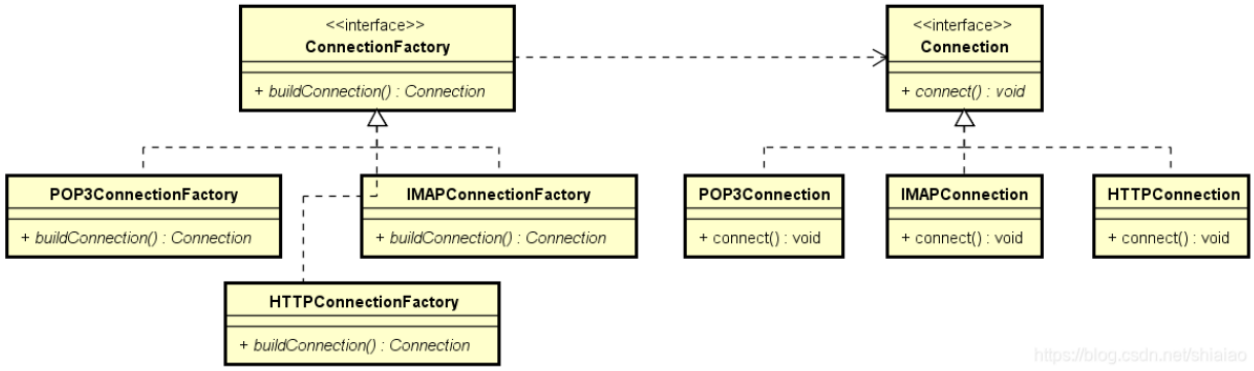


设计题

1、网络协议创建【工厂方法】

在某网络管理软件中，需要为不同的网络协议提供不同的连接类，例如针对 POP3协议的连接类 POP3Connection、针对 IMAP协议的连接类 IMAPConnection、针对 HTTP协议的连接类 HTTPConnection 等。由于网络连接对象的创建过程较为复杂，需要将其创建过程封装到专门的类中，该软件还将支持更多类型的网络协议。现采用工厂方法模式进行设计，绘制类图并编程模拟实现。

类图：



代码：

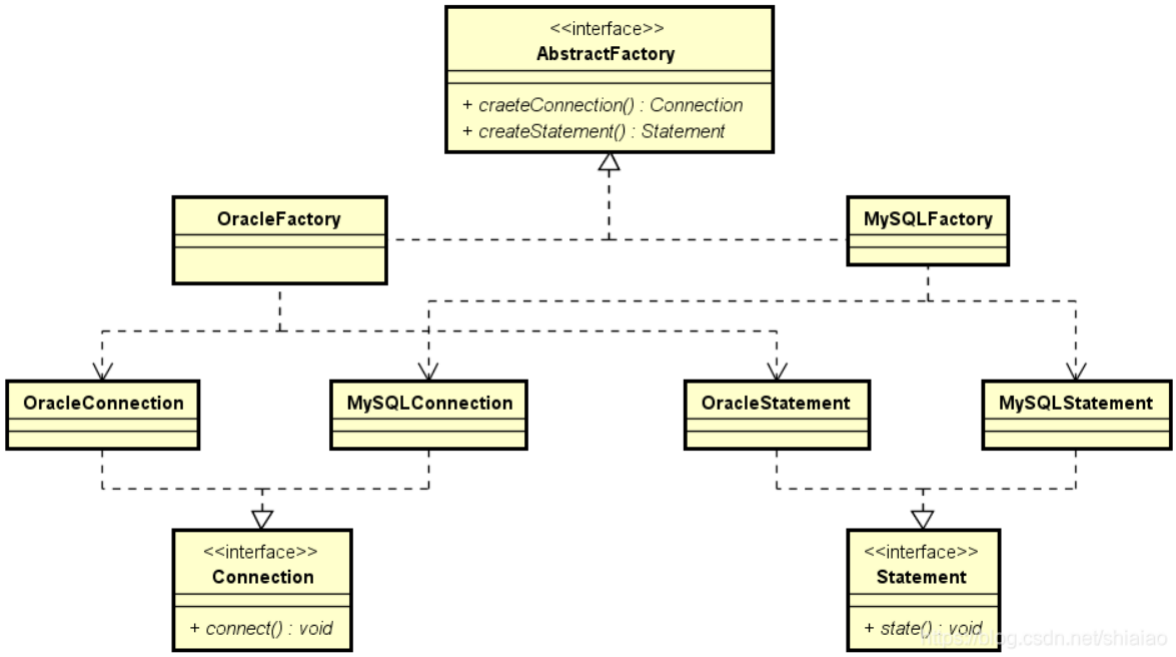
```
public interface Connection {
    void connect();
}
public class POP3Connection implements Connection{
    @Override
    public void connect() {
        System.out.println("execute POP3 protocol");
    }
}
public class IMAPConnection implements Connection {
```

```
    @Override
    public void connect() {
        System.out.println("execute IMAP protocol");
    }
}
public class HTTPConnection implements Connection{
    @Override
    public void connect() {
        System.out.println("execute HTTP connection");
    }
}
public interface ConnectionFactory {
    Connection buildConnection();
}
public class POP3ConnectionFactory implements ConnectionFactory{
    @Override
    public Connection buildConnection() {
        return new POP3Connection();
    }
}
public class IMAPConnectionFactory implements ConnectionFactory{
    @Override
    public Connection buildConnection() {
        return new IMAPConnection();
    }
}
public class HTTPConnectionFactory implements ConnectionFactory{
    @Override
    public Connection buildConnection() {
        return new HTTPConnection();
    }
}
public class Client {
    public static void main(String[] args) {
        ConnectionFactory connectionFactory=new HTTPConnectionFactory();
        Connection connection=connectionFactory.buildConnection();
        connection.connect();
    }
}
```

2、数据库连接对象、语句对象、数据库【抽象工厂】

某系统为了改进数据库操作的性能，用户可以自定义数据库连接对象Connection和语句对象 Statement,针对不同类型的数据库提供不同的连接对象和语句对象，例如提供Oracle或 MySQL专用连接类和语句类，而且用户可以通过配置文件等方式根据实际需要动态更换系统数据库。使用抽象工厂模式设计该系统，绘制对应的类图并编程模拟实现。

类图：



代码:

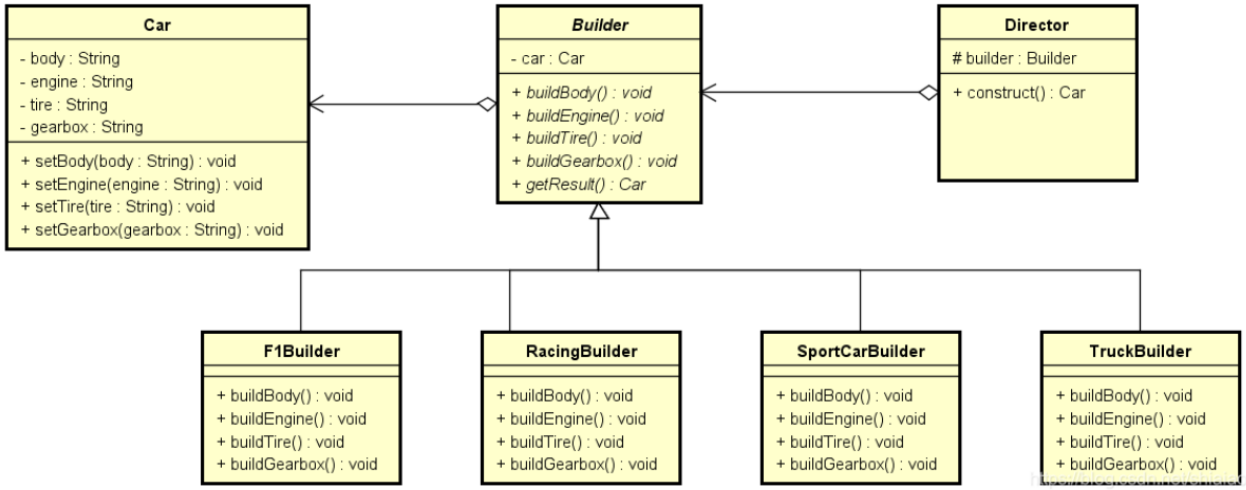
```
1  public interface Connection {
2      void connect();
3  }
4  public class MySQLConnection implements Connection{
5      @Override
6      public void connect() {
7          System.out.println("MySQL connection");
8      }
9  }
10 public class OracleConnection implements Connection{
11     @Override
12     public void connect() {
13         System.out.println("Oracle connection");
14     }
15 }
16 public interface Statement {
17     void state();
18 }
19 public class OracleStatement implements Statement{
20     @Override
21     public void state() {
22         System.out.println("Oracle Statement");
23     }
24 }
25 public class MySQLStatement implements Statement{
26     @Override
27     public void state() {
28         System.out.println("MySQL statement");
29     }
30 }
31 public interface AbstractFactory {
32     Connection createConnection();
33     Statement createStatement();
```

```
}
public class OracleFactory implements AbstractFactory{
    @Override
    public Connection createConnection() {
        return new OracleConnection();
    }

    @Override
    public Statement createStatement() {
        return new OracleStatement();
    }
}
public class MySQLFactory implements AbstractFactory{
    @Override
    public Connection createConnection() {
        return new MySQLConnection();
    }

    @Override
    public Statement createStatement() {
        return new MySQLStatement();
    }
}
type=OracleFactory
public class Client {
    public static void main(String[] args) {
        Properties properties=new Properties();
        InputStream
inputStream=Client.class.getResourceAsStream("Database/databaseType.properties");
        try {
            properties.load(inputStream);
            String type=properties.getProperty("type");
             // Database: package名称
            String className="Database."+type;
            AbstractFactory factory=(AbstractFactory)
Class.forName(className).newInstance();
            Connection connection=factory.createConnection();
            Statement statement=factory.createStatement();

            connection.connect();
            statement.state();

        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

3、赛车创建【建造者】

在某赛车游戏中，赛车包括方程式赛车、场地越野赛车、运动汽车、卡车等类型，不同类型的赛车的车身、发动机、轮胎、变速箱等部件有所区别。玩家可以自行选择赛车类型，系统将根据玩家的选择创建出一辆完整的赛车。现采用建造者模式实现赛车的构建，绘制对应的类图并编程模拟实现。

类图：



代码：

```java
public class Car {
    private String body;
    private String engine;
    private String tire;
    private String gearbox;

    public String getBody() {
        return body;
    }

    public String getEngine() {
        return engine;
    }

    public String getTire() {
        return tire;
    }

    public String getGearbox() {
        return gearbox;
    }

    public void setBody(String body) {
        this.body = body;
    }

    public void setEngine(String engine) {
        this.engine = engine;
    }
```

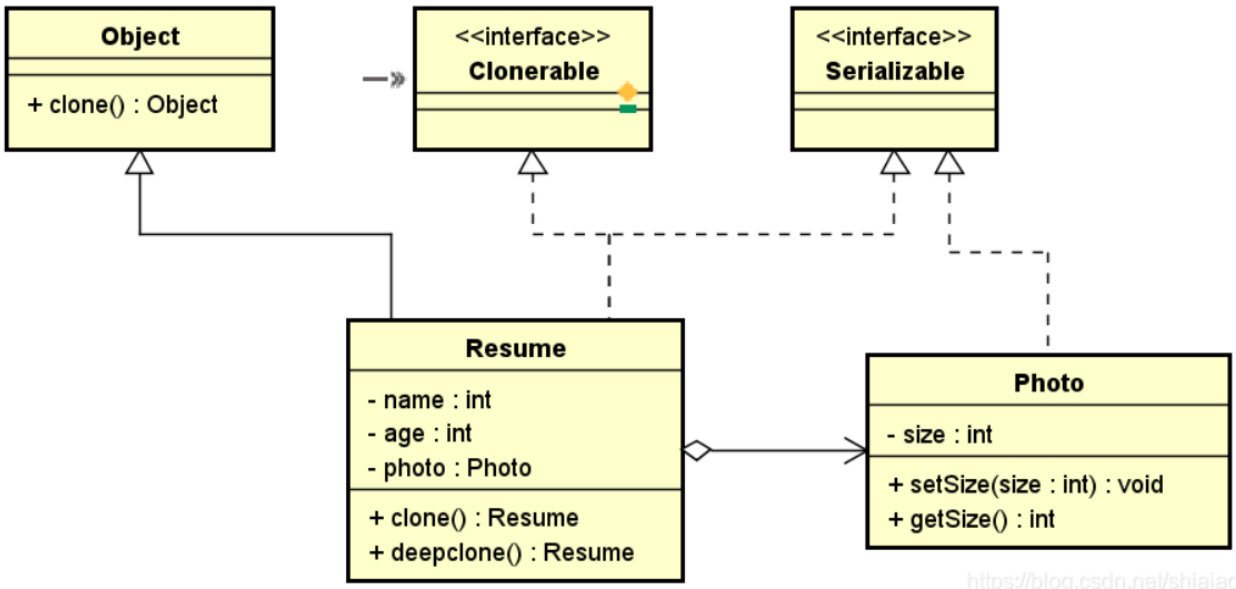
```
30
31      public void setTire(String tire) {
32          this.tire = tire;
33      }
34
35      public void setGearbox(String gearbox) {
36          this.gearbox = gearbox;
37      }
38  }
39  public abstract class Builder {
40      protected Car car = new Car();
41      public abstract void buildBody();
42      public abstract void buildEngine();
43      public abstract void buildTire();
44      public abstract void buildGearbox();
45      public Car getResult(){
46          return car;
47      }
48  }
49  public class F1Builder extends Builder{
50      @Override
51      public void buildBody() {
52          car.setBody("F1 body");
53      }
54
55      @Override
56      public void buildEngine() {
57          car.setEngine("F1 engine");
58      }
59
60      @Override
61      public void buildTire() {
62          car.setTire("F1 tire");
63      }
64
65      @Override
66      public void buildGearbox() {
67          car.setGearbox("F1 Gearbox");
68      }
69  }
70  public class RacingBuilder extends Builder{
71      @Override
72      public void buildBody() {
73          car.setBody("Racing body");
74      }
75
76      @Override
77      public void buildEngine() {
78          car.setEngine("Racing engine");
79      }
80
81      @Override
82      public void buildTire() {
83          car.setTire("Racing tire");
```

```
    }

    @Override
    public void buildGearbox() {
        car.setGearbox("Racing gearbox");
    }
}
//别的类似
public class Director {
    private Builder builder;

    public Director(Builder builder) {
        this.builder = builder;
    }
    public Car construct(){
        builder.buildBody();
        builder.buildEngine();
        builder.buildTire();
        builder.buildGearbox();
        return  builder.getResult();
    }
}

public class Client {
    public static void main(String[] args) {
        Builder builder=new TruckBuilder();
        Director director= new Director(builder);
        Car car=director.construct();
        System.out.println(car.getBody());
        System.out.println(car.getEngine());
        System.out.println(car.getTire());
        System.out.println(car.getGearbox());
    }
}
```

4、简历模板【原型，深克隆+浅克隆】

在某在线招聘网站中，用户可以创建一个简历模板。针对不同的工作岗位，可以复制该简历模板并进行适当修改后，生成一份新的简历。在复制简历时，用户可以选择是否复制简历中的照片：如果选择“是”,则照片将一同被复制，用户对新简历中的照片进行修改不会影响到简历模板中的照片，对模板进行修改也不会影响到新简历；如果选择“否”,则直接引用简历模板中的照片，修改简历模板中的照片将导致新简历中的照片一同修改，反之亦然。现采用原型模式设计该简历复制功能并提供浅克隆和深克隆两套实现方案，绘制对应的类图并编程模拟实现。

类图：



代码:

```java
public class Photo implements Serializable{

    private static final long serialVersionUID = 4455555666L;

    private int size;

    public Photo(int size) {
        super();
        this.size = size;
    }

    public int getSize() {
        return size;
    }

    public void setSize(int size) {
        this.size = size;
    }

    @Override
    public String toString() {
        return "Photo [size=" + size + "]";
    }
}
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class Resume implements Cloneable, Serializable {

    private static final long serialVersionUID = 4455666L;

```

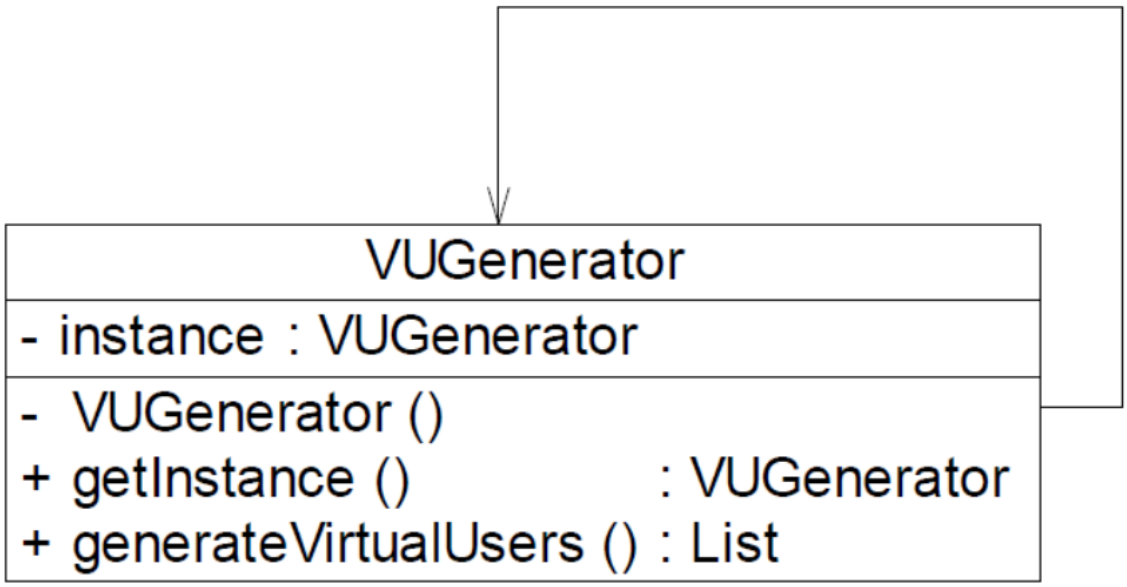
```
36      private String name;
37      private int age;
38      private Photo photo;
39
40      public Resume(String name, int age, Photo photo) {
41          super();
42          this.name = name;
43          this.age = age;
44          this.photo = photo;
45      }
46
47
48      public Photo getPhoto() {
49          return photo;
50      }
51
52      public void setPhoto(Photo photo) {
53          this.photo = photo;
54      }
55      @Override
56      public String toString() {
57          return "Resume [name=" + name + ", age=" + age + ", photo=" +
    photo.toString() + "]";
58      }
59
60      public Resume clone() {
61          Object obj = null;
62          try {
63              obj = super.clone();
64          } catch (CloneNotSupportedException e) {
65              // TODO: handle exception
66              System.err.println("Not supported clonerable");
67              ;
68          }
69          return (Resume) obj;
70      }
71
72      public Resume deepclone() {
73          Object obj = null;
74          try {
75              ByteArrayOutputStream baos = new ByteArrayOutputStream();
76              ObjectOutputStream oos = new ObjectOutputStream(baos);
77              oos.writeObject(this);
78
79              ByteArrayInputStream bais = new
    ByteArrayInputStream(baos.toByteArray());
80              ObjectInputStream ois = new ObjectInputStream(bais);
81              obj = ois.readObject();
82          } catch (IOException e) {
83              // TODO: handle exception
84              System.err.println(e.getMessage());
85          } catch (ClassNotFoundException e) {
86              // TODO Auto-generated catch block
87              e.printStackTrace();
```

```
        }

        return (Resume) obj;
    }
}
import java.util.Scanner;

public class Client {
    public static void main(String[] args) {
        Photo photo=new Photo(16);
        Resume resume=new Resume("ysa", 22, photo);
        //浅拷贝
        Resume resume1=resume.clone();
        photo.setSize(32);
        System.out.println("浅拷贝: ");
        System.out.println("resume:"+resume);
        System.out.println("resume1:"+resume1);
        //深拷贝
        Resume resume2=resume.deepclone();
        photo.setSize(64);
        System.out.println("深拷贝: ");
        System.out.println("resume:"+resume);
        System.out.println("resume2:"+resume2);

        //选择深拷贝还是浅拷贝复制照片
        Resume newResume=new Resume("zly", 25, new Photo(64));
        Scanner scanner=new Scanner(System.in);
        String str=scanner.next();
        if("y".equalsIgnoreCase(str)) {
            newResume=resume.deepclone();
            System.out.println("deepclone");
        }else if("n".equalsIgnoreCase(str)){
            newResume=resume.clone();
            System.out.println("clone");
        }
    }
}
```

5、虚拟用户生成器【单例模式】

某Web性能测试软件中包含一个虚拟用户生成器(Virtual User Generator)。为了避免生成的虚拟用户数量不一致，该测试软件在工作时只允许启动唯一一个虚拟用户生成器。采用单例模式设计该虚拟用户生成器，绘制类图并分别使用饿汉式单例、双重检测锁和IoDH 三种方式编程模拟实现。

类图:



代码:

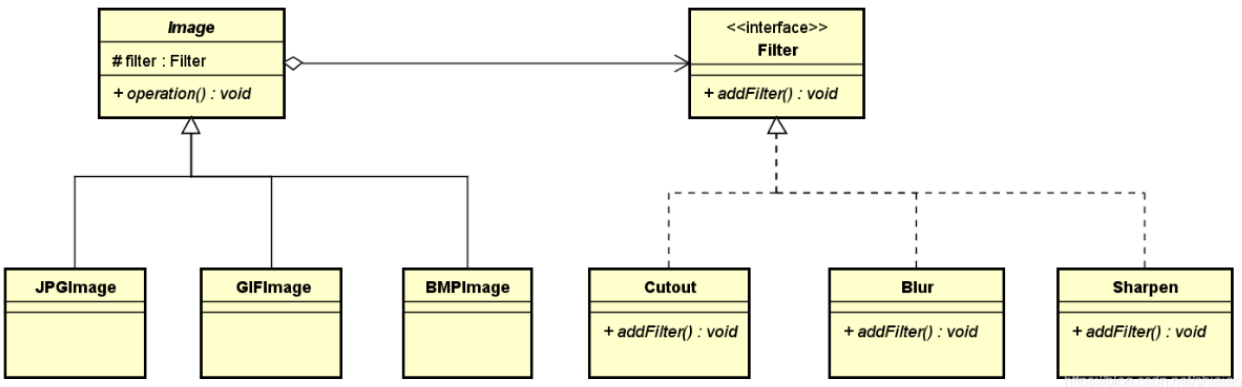
```
public class VUGenerator {

    // 饿汉式单例
    private static final VUGenerator instance = new VUGenerator();

    // 私有构造函数，防止外部实例化
    private VUGenerator() { }

    // 提供一个公共的静态方法，返回唯一实例
    public static VUGenerator getInstance() {
        return instance;
    }

    // 生成虚拟用户的方法
    public List generateVirtualUsers() {
        return null;
    }
}

public class VUGenerator {

    // 双重检测锁单例
    private volatile static VUGenerator instance = null;

    // 私有构造函数，防止外部实例化
    private VUGenerator() { }

    // 提供一个公共的静态方法，返回唯一实例
    public static VUGenerator getInstance() {
        if (instance == null) {
            synchronized (VUGenerator.class) {
                if (instance == null) {
                    instance = new VUGenerator();
```

```
                }
            }
        }
        return instance;
    }

    // 生成虚拟用户的方法
    public List generateVirtualUsers() {
        return null;
    }
}

public class VUGenerator {

    // 内部类方式单例（也称为IODH）
    private VUGenerator() { }

    private static class HolderClass {
        private final static VUGenerator instance = new VUGenerator();
    }

    // 提供一个公共的静态方法，返回唯一实例
    public static VUGenerator getInstance() {
        return HolderClass.instance;
    }

    // 生成虚拟用户的方法
    public List generateVirtualUsers() {
        return null;
    }
}
```

6、图像格式+滤镜【桥接】

某手机美图 APP软件支持多种不同的图像格式，例如 JPG、GIF、BMP等常用图像格式，同时提供了多种不同的滤镜对图像进行处理，例如木刻滤镜(Cutout)、模糊滤镜(Blur)、锐化滤镜(Sharpen)、纹理滤镜(Texture)等。现采用桥接模式设计该 APP软件，使得该软件能够为多种图像格式提供一系列图像处理滤镜，同时还能够很方便地增加新的图像格式和滤镜，绘制对应的类图并编程模拟实现。

类图：



代码：

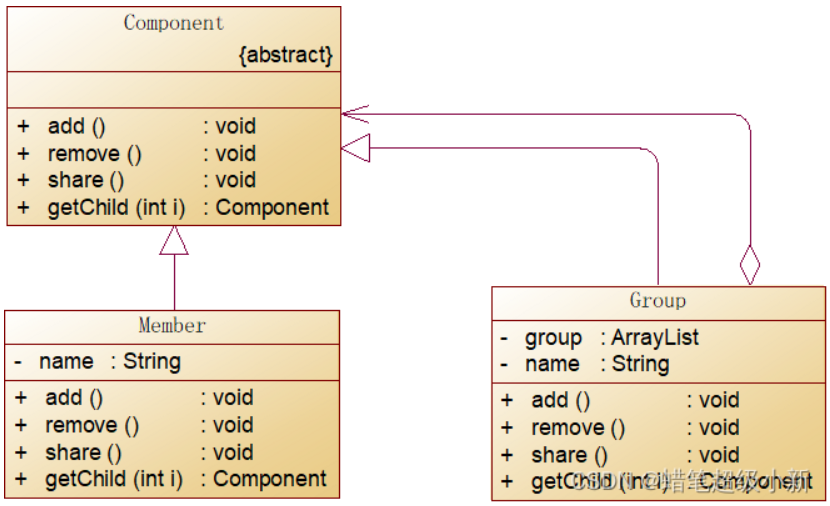
```
public abstract class Image {
    protected Filter filter;

    public Image(Filter filter) {
        super();
        this.filter = filter;
    }
    public abstract void operation();
}
public class JPGImage extends Image{

    public JPGImage(Filter filter) {
        super(filter);
        // TODO Auto-generated constructor stub
    }
    @Override
    public void operation() {
        // TODO Auto-generated method stub
        filter.addFilter();
    }
}
public class GIFImage extends Image{

    public GIFImage(Filter filter) {
        super(filter);
        // TODO Auto-generated constructor stub
    }
    @Override
    public void operation() {
        // TODO Auto-generated method stub
        filter.addFilter();
    }
}
public class BMPImage extends Image{

     public BMPImage(Filter filter) {
        super(filter);
        // TODO Auto-generated constructor stub
    }
    @Override
    public void operation() {
        // TODO Auto-generated method stub

    }
}
//滤镜接口
public interface Filter {
    public void addFilter();

}
public class Cutout implements Filter{

```

```
53      @Override
54      public void addFilter() {
55          // TODO Auto-generated method stub
56          System.out.println("add cutout filter");
57      }
58  }
59  public class Blur implements Filter{
60      @Override
61      public void addFilter() {
62          // TODO Auto-generated method stub
63          System.out.println("add Blur filter");
64      }
65  }
66  public class Sharpen implements Filter{
67      @Override
68      public void addFilter() {
69          // TODO Auto-generated method stub
70          System.out.println("add Sharpen filter");
71      }
72  }
73  public class Client {
74      public static void main(String[] args) {
75          Filter filter=new Sharpen();
76          Image image= new JPGImage(filter);
77          image.operation();
78      }
79
80  }
```

7、群组、子群组、成员【组合】

某移动社交软件要增加一个群组(Group)功能。通过设置，用户可以将自己的动态信息(包括最新动态、新上传的视频以及分享的链接等)分享给某个特定的成员(Member),也可以分享给某个群组中的所有成员；用户可以将成员添加至某个指定的群组；此外，还允许用户在一个群组中添加子群组，以便更加灵活地实现面向特定人群的信息共享。现采用组合模式设计该群组功能，绘制对应的类图并编程模拟实现。

类图：



代码：

```
public class Client {
    public static void main(String args[]) {
        //针对抽象构件编程
        Component member1,member2,group1,group2;

        group1 = new Group("小组1");
        group2 = new Group("小组2");

        member1 = new Member("小明");
        member2 = new Member("小花");

        group1.add(member1);
        group1.add(group2);
        group2.add(member2);


        member1.share();
        group1.share();
    }
}
public abstract class Component {
    public abstract void add(Component file);
    public abstract void remove(Component file);
    public abstract Component getChild(int i);
    public abstract void share();
}
public class Group extends Component {
    //定义集合fileList，用于存储AbstractFile类型的成员
    private ArrayList<Component> group=new ArrayList<Component>();
    private String name;


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

    public void add(Component file) {
       group.add(file);
    }

    public void remove(Component file) {
        group.remove(file);
    }

    public Component getChild(int i) {
        return (Component)group.get(i);
    }

    public void share() {
        System.out.println("给" + name + "群组全部成员进行分享");  //模拟分享

        //递归调用成员构件的方法
```

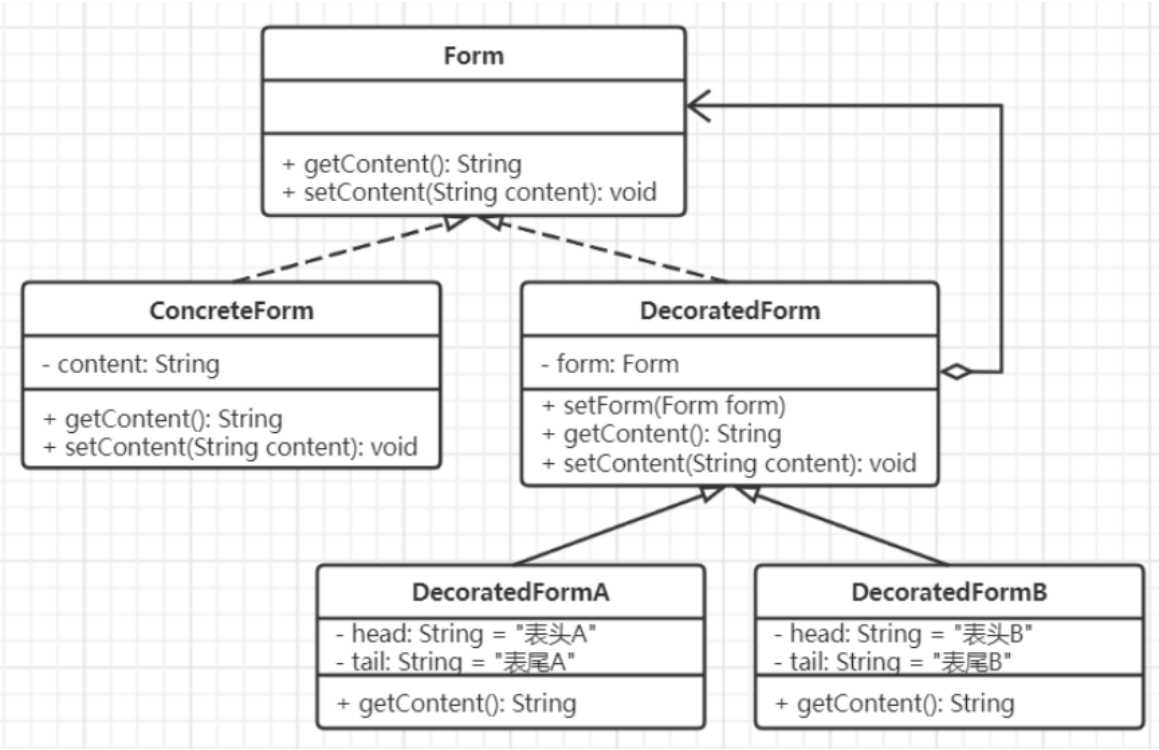
```
        for(Object obj : group) {
            ((Component)obj).share();
        }
    }
}
public class Member extends Component {
    private String name;

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

    public void add(Component file) {
       System.out.println("对不起，不支持该方法！");
    }

    public void remove(Component file) {
        System.out.println("对不起，不支持该方法！");
    }

    public Component getChild(int i) {
        System.out.println("对不起，不支持该方法！");
        return null;
    }

    public void share() {
        //模拟分享
        System.out.println("给成员" + name + "分享");
    }
}
```

8、报表添加报表头/报表尾【装饰】

在某OA系统中提供一个报表生成工具，用户可以通过该工具为报表增加表头和表尾，允许用户为报表增加多个不同的表头和表尾，用户还可以自行确定表头和表尾的次序。为了能够灵活设置表头和表尾的次序并易于增加新的表头和表尾，现采用装饰模式设计该报表生成工具，绘制对应的类图并编程模拟实现。

类图：

【下面类图中还需要添加一个公共的抽象接口】



代码：

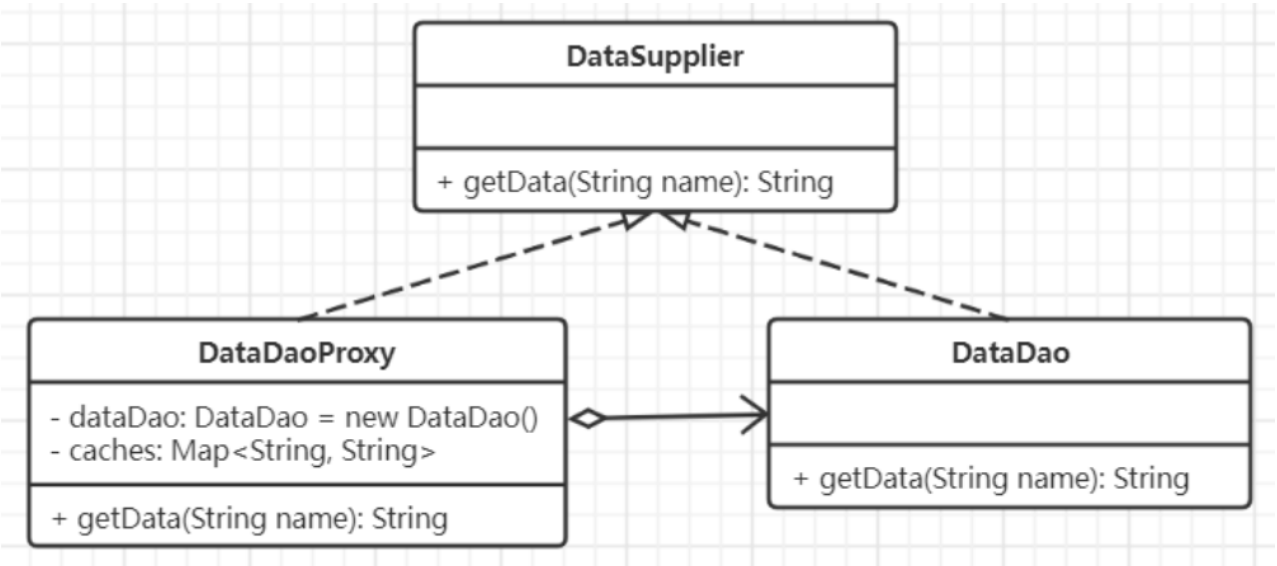
```
public interface Form {
    String getContent();

    void setContent(String content);
}
public class ConcreteForm implements Form {
    private String content;

    @Override
    public String getContent() {
        return content;
    }

    @Override
    public void setContent(String content) {
        this.content = content;
    }
}
public class DecoratedForm implements Form {
    private Form form;

    public void setForm(Form form) {
        this.form = form;
    }

    @Override
    public String getContent() {
        return form.getContent();
    }

    @Override
```

```
    public void setContent(String content) {
        form.setContent(content);
    }
}
public class DecoratedFormA extends DecoratedForm {
    private static final String head = "表头A";
    private static final String tail = "表尾A";

    @Override
    public String getContent() {
        return head + "\n" + super.getContent() + "\n" + tail;
    }
}
public class DecoratedFormB extends DecoratedForm {
    private static final String head = "表头B";
    private static final String tail = "表尾B";

    @Override
    public String getContent() {
        return head + "\n" + super.getContent() + "\n" + tail;
    }
}
public class Main {
    public static void main(String[] args) throws IOException, SAXException {
        ConcreteForm form = new ConcreteForm();
        form.setContent("报表内容……");
        XmlParse xmlParse = new XmlParse("decoratorConfig.xml");
        DecoratedForm decoratedForm1 = (DecoratedForm)
xmlParse.objectOfTag("decorator1");
        decoratedForm1.setForm(form);
        System.out.println(decoratedForm1.getContent());
        System.out.println();
        // 多重装饰
        DecoratedForm decoratedForm2 = (DecoratedForm)
xmlParse.objectOfTag("decorator2");
        decoratedForm2.setForm(decoratedForm1);
        System.out.println(decoratedForm2.getContent());
    }
}
```

9、辅助存储【缓冲代理】

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，然后将数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。

类图：



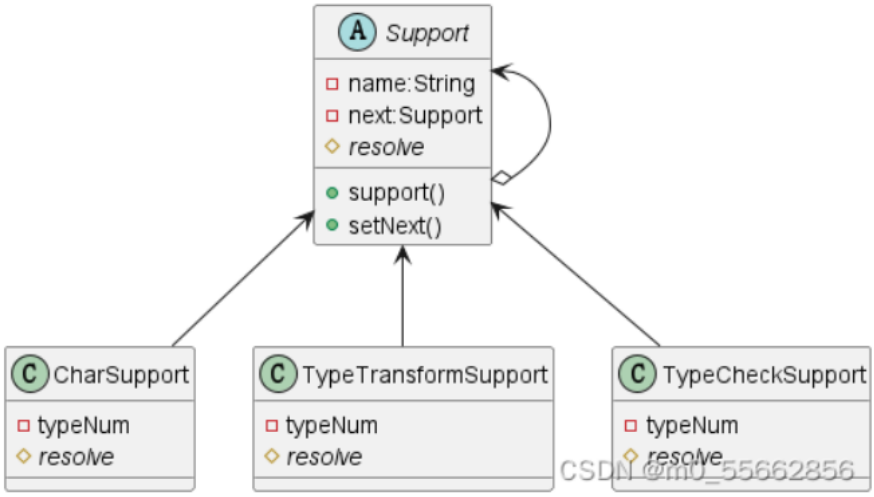

```java
public interface DataSupplier {
    String getData(String name);
}

public class DataDao implements DataSupplier {
    @Override
    public String getData(String name) {
        return "Data of " + name;
    }
}

public class DataDaoProxy implements DataSupplier {
    private final DataDao dataDao = new DataDao();

    private final Map<String, String> caches = new HashMap<>();

    @Override
    public String getData(String name) {
        String data = caches.get(name);
        if (data == null) {
            data = dataDao.getData(name);
            caches.put(name, data);
            System.out.println("存入缓存: (" + name + ", " + data + ")");
        }
        return data;
    }
}

public class Main {
    public static void main(String[] args) throws IOException, SAXException {
        XmlParse xmlParse = new XmlParse("dataSupplierConfig.xml");
        DataSupplier dataSupplier = (DataSupplier)
xmlParse.objectOfTag("supplier");
        System.out.println(dataSupplier.getData("count"));
        System.out.println(dataSupplier.getData("count"));
    }
```

```
}
```

10、数据过滤器【职责链】

在某 Web框架中采用职责链模式来组织数据过滤器，不同的数据过滤器提供了不同的功能，例如字符编码转换过滤器、数据类型转换过滤器、数据校验过滤器等，可以将多个过滤器连接成一个过滤器链，进而对数据进行多次处理。根据以上描述，绘制对应的类图并编程模拟实现。

类图：



CSDN @m0_55662856

代码：

```
public class Trouble {
    private int number;//定义trouble的类型，对应过滤器类型
    public Trouble(int number){
        this.number = number;
    }
    public int getNumber(){
        return this.number;
    }
    public String toString(){
        switch (number){
            case 1: return "[Trouble " + number + " CheckTrouble]";
            case 2: return "[Trouble " + number + " TypeTransformTrouble]";
            case 3: return "[Trouble " + number + " TypeCheckTrouble]";
            default: return "No such trouble";
        }
    }
}
public abstract class Support {
    private String name;
    private Support next;
    public Support(String name){
        this.name = name;
    }
    public Support setNext(Support next){
        this.next = next;
```

```
        return next;
    }
    public final void support(Trouble trouble){
        if(resolve(trouble)){
            done(trouble);
        }else if(next != null){
            next.support(trouble);
        }else {
            fail(trouble);
        }
    }
    public String toString(){
        return "[" + name + "]";
    }
    protected abstract boolean resolve(Trouble trouble);
    protected void done(Trouble trouble){
        System.out.println(trouble + " is resolved by " + this + ".");
    }
    protected void fail(Trouble trouble){
        System.out.println(trouble + " cannot be resolved");
    }
}
public class CharSupport extends Support{
    private int typeNum = 1;
    public CharSupport(String name) {
        super(name);
    }

    @Override
    protected boolean resolve(Trouble trouble) {
        if(trouble.getNumber() == typeNum){
            return true;
        }else {
            return false;
        }
    }
}
public class TypeTransformSupport extends Support{
    private int typeNum = 2;
    public TypeTransformSupport(String name) {
        super(name);
    }

    @Override
    protected boolean resolve(Trouble trouble) {
        if(trouble.getNumber() == typeNum){
            return true;
        }else {
            return false;
        }
    }
}
public class TypeCheckSupport extends Support{
    private int typeNum = 3;
```

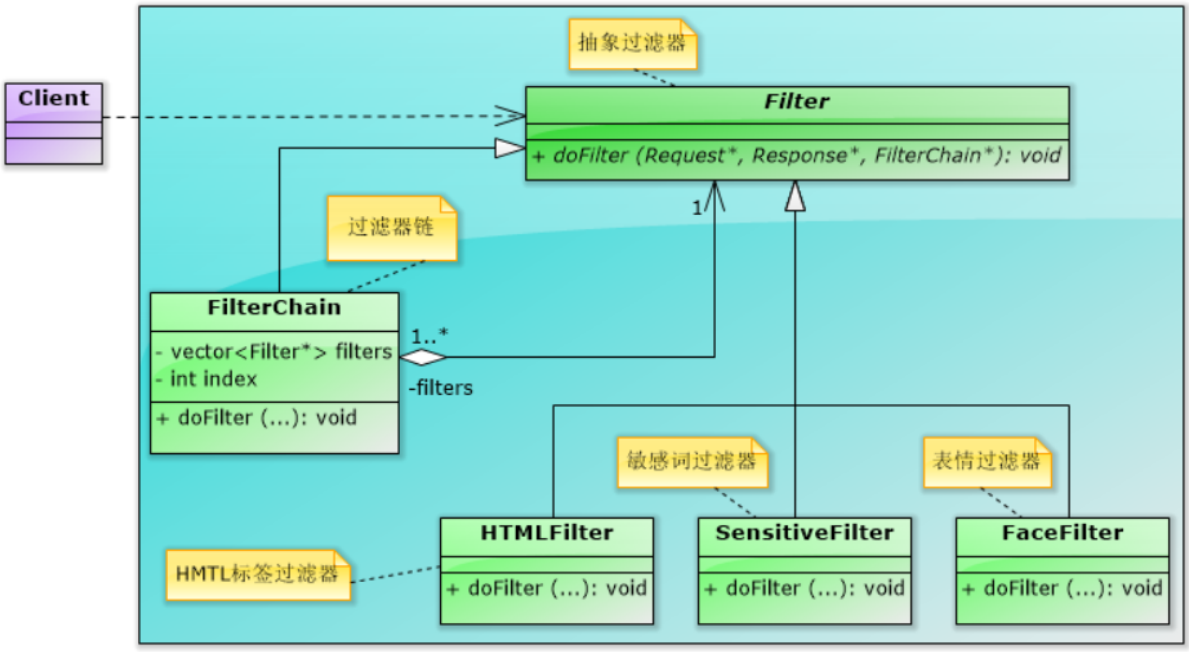
```
    public TypeCheckSupport(String name) {
        super(name);
    }

    @Override
    protected boolean resolve(Trouble trouble) {
        if(trouble.getNumber() == typeNum){
            return true;
        }else {
            return false;
        }
    }
}
public class Test {
    public static void main(String[] args) {
        Support charSupport = new CharSupport("charSupport");
        Support typeTransformSupport = new
TypeTransformSupport("typeTransformSupport");
        Support typeCheckSupport = new TypeCheckSupport("typeCheckSupport");

        charSupport.setNext(typeTransformSupport).setNext(typeCheckSupport);

        for(int i = 1; i <= 4; i++){
            charSupport.support(new Trouble(i));
        }
    }
}
```

11、字符过滤器【模拟JavaWeb的双向过滤器，变式的职责链模式，结合了组合模式】

①实际开发中，在业务处理之前，通常需要进行权限检查、数据校验、逻辑检查等，然后才开始真正的业务逻辑。可以把这些功能分散到一个功能链中。

②过滤器：**每个过滤器负责自己的处理，然后转交给下一个过滤器**，直到把所有的过滤器都走完（如权限检查、字符转换等）。

类图：



代码：

```
//行为型模式：职责链模式
//场景：字符过滤（模拟JavaWeb的双向过滤器）
//当客户端发送给服务器时，request字符串被各个过滤器按顺序处理，
//而返回的response是逆着过滤器被调用顺序被处理的。（可参考下面的技巧）
//思路细节技巧：
//(1)Filter的doFilter方法改为doFilter(Request*,Resopnse*,FilterChain*),有
//FilterChain指针，为利用FilterChain调用下一个Filter做准备
//(2)FilterChain继承Filter,这样，FilterChain既是FilterChain又是Filter,那么
//FilterChain就可以调用Filter的方法doFilter(Request*,Resopnse*,FilterChain*)
//(3)FilterChain的doFilter(Request*,Resopnse*,FilterChain*)中，有index标记了执
//行到第几个Filter,当所有Filter执行完后request处理后，就会return,以倒序继续执
//行response处理

#include <iostream>
#include <string>
#include <vector>

using namespace std;

//**********************************************字符串替换函数
**********************************
void Replace(std::basic_string<char>& s, const std::basic_string<char>& src,
const std::basic_string<char>& dest);


//*****************************辅助类*****************
//请求类
class CRequest{
private:
    string strContent;//请求的内容
public:
    CRequest(string content);
    void SetContent(string content);
```

```
    string& GetContent();
};

//响应类
class CResponse{
private:
    string strContent;//响应的内容
public:
    CResponse(string content);
    void SetContent(string content);
    string& GetContent();
};

//***************************抽象职责类**************
//Filter（抽象过滤器）
class CFilterChain;
class CFilter{
public:
    virtual ~CFilter();
    virtual bool IsChain();
    virtual vector<CFilter*>* GetFilters();
    virtual void DoFilter(CRequest* req, CResponse* res, CFilterChain* chain) =
0;
};
//FilterChain（过滤器链）
//注意，FilterChain继承自Filter，是为了这个链可以像其他过滤器(如HTMLFilter)一样地被加入到其
他链中
//从而，可以将两条FilterChain连接成一条。
class CFilterChain : public CFilter{
private:
    vector<CFilter*> vFilter;
    int iIdx;
public:
    CFilterChain();
    ~CFilterChain();
    bool IsChain();
    vector<CFilter*>* GetFilters();
    CFilterChain& AddFilter(CFilter* filter);
      //将链本身作为一个过滤器，也实现了doFilter功能，但这里只是简单地调用链中各个
    //过滤器并记录下一个过滤器的索引
    void DoFilter(CRequest* req, CResponse* res, CFilterChain* chain);
};

//HtmlFilter
class CHtmlFilter : public CFilter{
public:
    ~CHtmlFilter();
    //当客户端发送给服务器时，request字符串被各个过滤器后按顺序处理，
    //而返回的response逆着过滤器被调用顺序被处理。（可参考下面的技巧）
    void DoFilter(CRequest* req, CResponse* res, CFilterChain* chain);
};

//SensitiveFilter（敏感关键字过滤器）
class CSensitiveFilter : public CFilter{
```

```
public:
    ~CSensitiveFilter();
    void DoFilter(CRequest* req, CResponse* res, CFilterChain* chain);
};

//FaceFilter（表情过滤器）
class CFaceFilter : public CFilter{
public:
    ~CFaceFilter();
    void DoFilter(CRequest* req, CResponse* res, CFilterChain* chain);
};

void DelFilter(CFilter* pRoot);

//*************************************************字符串替换函数
*********************************
void Replace(std::basic_string<char>& s, const std::basic_string<char>& src,
const std::basic_string<char>& dest)
{
    std::basic_string<char>::size_type pos = 0;
    while (true)
    {
        pos = s.find(src, pos);
        if (std::basic_string<char>::npos == pos)
            break;

        s.replace(pos, src.size(), dest);
        pos += src.size();
    }
}
//*************************************************************************************
*********
//****************************辅助类*****************
//请求类
CRequest::CRequest(string content){strContent = content;}
void CRequest::SetContent(string content){strContent = content;}
string& CRequest::GetContent(){return strContent;}

//响应类
CResponse::CResponse(string content){strContent = content;}
void CResponse::SetContent(string content){strContent = content;}
string& CResponse::GetContent(){return strContent;}


//****************************抽象职责类**************
//职责链

//Filter（抽象过滤器）
CFilter::~CFilter(){}
bool CFilter::IsChain(){return false;}
vector<CFilter*>* CFilter::GetFilters(){return NULL;}
//FilterChain（过滤器链）
//注意，FilterChain继承自Filter，是为了这个链可以像其他过滤器(如HTMLFilter)一样地被加入到其
他链中
```

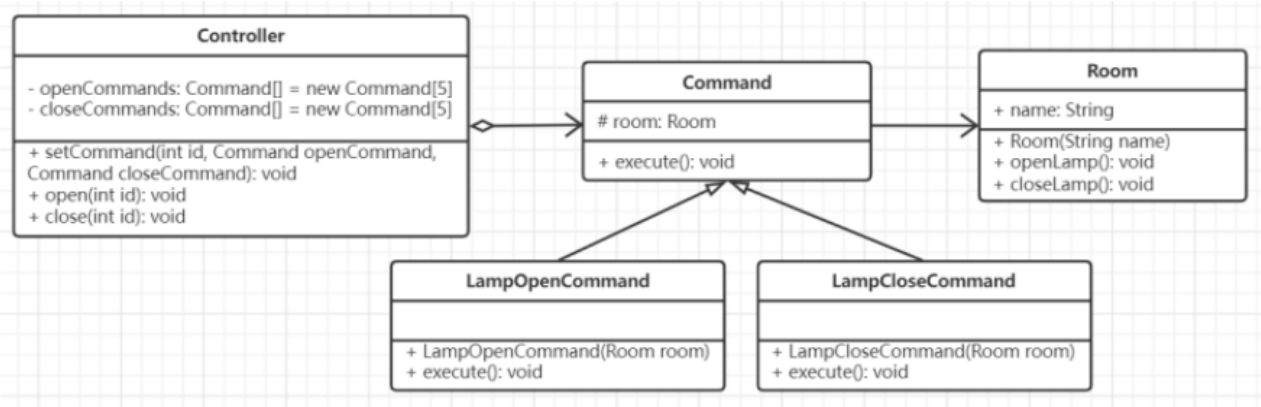
```
//从而，可以将两条FilterChain连接成一条。

CFilterChain::CFilterChain(){iIdx = 0;}
CFilterChain::~CFilterChain(){ cout << "~CFilterChain" << endl;}
bool CFilterChain::IsChain(){return true;}
vector<CFilter*>* CFilterChain::GetFilters(){return &vFilter;}
CFilterChain& CFilterChain::AddFilter(CFilter* filter){vFilter.push_back(filter);
return *this;}
//将链本身作为一个过滤器，也实现了doFilter功能，但这里只是简单地调用链中各个
//过滤器并记录下一个过滤器的索引
void CFilterChain::DoFilter(CRequest* req, CResponse* res, CFilterChain* chain){
    if(iIdx == vFilter.size())          return;
    CFilter* pFilter = vFilter[iIdx++];
    pFilter->DoFilter(req, res, chain);
}

//HtmlFilter
//当客户端发送给服务器时，request字符串被各个过滤器后按顺序处理，
//而返回的response逆着过滤器被调用顺序被处理。（可参考下面的技巧）
CHtmlFilter::~CHtmlFilter(){cout << "~CHtmlFilter" << endl;}
void CHtmlFilter::DoFilter(CRequest* req, CResponse* res, CFilterChain* chain){
    //1、处理html标签
    Replace(req->GetContent(), "<", "[");
    Replace(req->GetContent(), ">", "]");
    req->GetContent() += "-->HtmlFilter";
    //2、调用下一个过滤器（注意里面可能会递归）
    chain->DoFilter(req, res, chain);
    //3、在前2步之后，再处理response，可以达到栈式（逆序）处理response字符串-->技巧!
    res->GetContent() += "-->HtmlFilter";
}

//SensitiveFilter（敏感关键字过滤器）
CSensitiveFilter::~CSensitiveFilter(){cout << "~CSensitiveFilter" << endl;}
void CSensitiveFilter::DoFilter(CRequest* req, CResponse* res, CFilterChain*
chain){
    //1、处理敏感关键词
    Replace(req->GetContent(), "敏感", "");
    Replace(req->GetContent(), "被就业", "就业");
    req->GetContent() += "-->SensitiveFilter";
    //2、调用下一个过滤器（注意里面可能会递归）
    chain->DoFilter(req, res, chain);
    //3、在前2步之后，再处理response，可以达到栈式（逆序）处理response字符串-->技巧!
    res->GetContent() += "-->SensitiveFilter";
}

//FaceFilter（表情过滤器）
CFaceFilter::~CFaceFilter(){cout << "~CFaceFilter" << endl;}
void CFaceFilter::DoFilter(CRequest* req, CResponse* res, CFilterChain* chain){
    //1、处理敏感关键词
    Replace(req->GetContent(), ":)", "^V^");
    req->GetContent() += "-->FaceFilter";
    //2、调用下一个过滤器（注意里面可能会递归）
    chain->DoFilter(req, res, chain);
    //3、在前2步之后，再处理response，可以达到栈式（逆序）处理response字符串-->技巧!
```

```
    res->GetContent() += "-->FaceFilter";
}


void DelFilter(CFilter* pRoot){
    vector<CFilter*>* pFilters = pRoot->GetFilters();
    if(pFilters != NULL){
        for(vector<CFilter*>::iterator it = pFilters->begin(); it != pFilters->end(); it++){
            CFilter* pFilter = (*it);
            if(pFilter->IsChain())      DelFilter(pFilter);
            else                                delete pFilter;
        }
    }
    delete pRoot;
}
void main()
{
    string strReq = "大家好:),<script>,敏感,被就业,网络授课没感觉,因为看不见大家伙儿";
    CRequest oRequest("Request : " + strReq);
    CResponse oResponse("Response : ");

    //创建3个过滤器
    CFilter* pHtmlFilter = new CHtmlFilter();//HTML标签过滤
    CFilter* pSensitiveFilter = new CSensitiveFilter();//敏感关键词过滤
    CFilter* pFaceFilter = new CFaceFilter(); //表情过滤

    CFilterChain* pMainFilterChain = new CFilterChain();//第1条过滤链（有两个过滤器）
    pMainFilterChain->AddFilter(pHtmlFilter);
    pMainFilterChain->AddFilter(pSensitiveFilter);

    CFilterChain* pSubFilterChain = new CFilterChain(); //第2条过滤链（只有一个过滤器）
    pSubFilterChain->AddFilter(pFaceFilter);

    //交两条过滤链合成一条过滤链（共有3个过滤器）
    pMainFilterChain->AddFilter(pSubFilterChain);

    //开始过滤
    pMainFilterChain->DoFilter(&oRequest, &oResponse, pMainFilterChain);

    cout << oRequest.GetContent() << endl;
    cout << oResponse.GetContent() << endl;

    DelFilter(pMainFilterChain);
}
```

12、灯具开关【指定个数】

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。

类图：



代码：

```
public abstract class Command {
    protected Room room;

    public abstract void execute();
}
public class LampCloseCommand extends Command {
    public LampCloseCommand(Room room) {
        this.room = room;
    }

    @Override
    public void execute() {
        room.closeLamp();
    }
}
public class LampOpenCommand extends Command {
    public LampOpenCommand(Room room) {
        this.room = room;
    }

    @Override
    public void execute() {
        room.openLamp();
    }
}
public class Controller {
    private final Command[] openCommands = new Command[5];
    private final Command[] closeCommands = new Command[5];

    public void setCommand(int id, Command openCommand, Command closeCommand) {
        openCommands[id - 1] = openCommand;
        closeCommands[id - 1] = closeCommand;
```

```
    }

    public void open(int id) {
        Command openCommand = openCommands[id - 1];
        if (openCommand == null) {
            System.out.println("开关 " + id + " 未绑定房间! ");
            return;
        }
        openCommand.execute();
    }

    public void close(int id) {
        Command closeCommand = closeCommands[id - 1];
        if (closeCommand == null) {
            System.out.println("开关 " + id + " 未绑定房间! ");
            return;
        }
        closeCommand.execute();
    }
}
public class Room {
    public final String name;

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

    public void openLamp() {
        System.out.println(name + "开灯");
    }

    public void closeLamp() {
        System.out.println(name + "关灯");
    }
}
public class Main {
    public static void main(String[] args) {
        Controller controller = new Controller();
        for (int i = 1; i <= 5; i++) {
            String roomName = "房间" + i;
            controller.setCommand(i, new LampOpenCommand(new Room(roomName)),
                    new LampCloseCommand(new Room(roomName)));
        }
        controller.open(1);
        controller.open(2);
        controller.close(1);
        controller.close(2);
    }
}
```

13、旅游系统【中介者】

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(Travel Companies Subsystem)、宾馆子系统(Hotels Subsystem)、餐厅子系统(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、餐厅子系统、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。

代码：

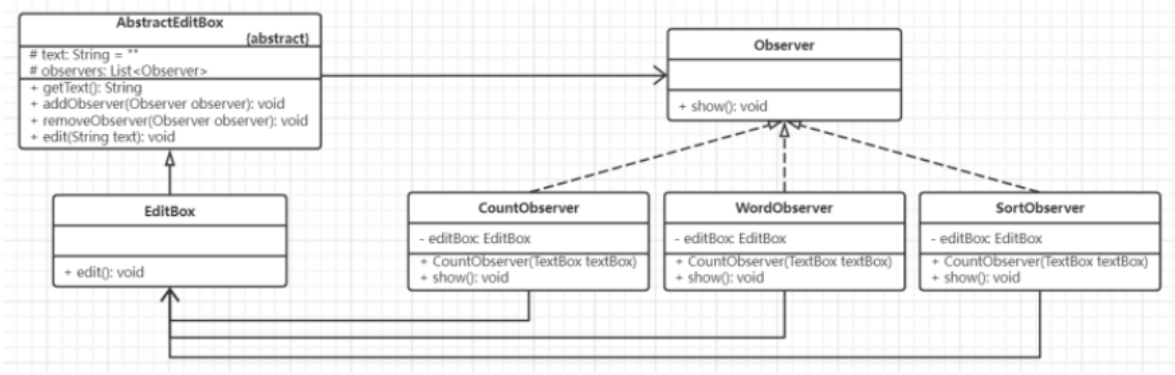
```
public class AirportSubsystem extends SubSystem {
    @Override
    public void getInformation() {
        // 实现获取机场信息的方法
    }
}

public class HotelsSubsystem extends SubSystem {
    @Override
    public void getInformation() {
        // 实现获取酒店信息的方法
    }
}

public interface Mediator {
    void shareInformation();
}

public class TravelCompaniesSubsystem extends SubSystem {
    @Override
    public void getInformation() {
        // 实现获取旅游公司信息的方法
    }
}

public class RestaurantsSubsystem extends SubSystem {
    @Override
    public void getInformation() {
        // 实现获取餐厅信息的方法
    }
}

public abstract class SubSystem {
    public abstract void getInformation();
}

public class System implements Mediator {
    AirportSubsystem airportSubsystem;
    HotelsSubsystem hotelsSubsystem;
    TravelCompaniesSubsystem travelCompaniesSubsystem;
```

```
    RestaurantsSubsystem restaurantsSubsystem;
    TourismAttractionsSubsystem tourismAttractionsSubsystem;

    @Override
    public void shareInformation() {
        // 实现信息分享的方法
    }
}

public class TourismAttractionsSubsystem extends SubSystem {
    @Override
    public void getInformation() {
        // 实现获取旅游景点信息的方法
    }
}
```

14、文本编辑，多个文本信息统计区【观察者】

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。

类图：



代码：

```
public interface Observer {
    void show();
}
public class CountObserver implements Observer {
    private final EditBox editBox;

    public CountObserver(EditBox editBox) {
        this.editBox = editBox;
    }

    @Override
```

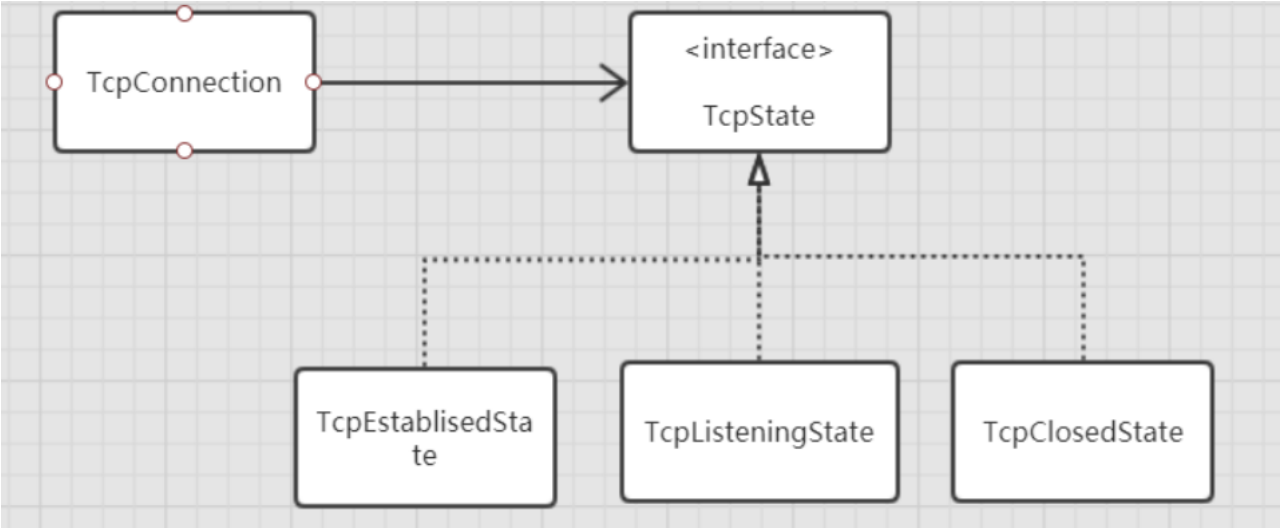
```
    public void show() {
        String text = editBox.getText();
        int wordCount = text.split("\\s+").length;
        System.out.println("单词数: " + wordCount);
        int charCount = text.codePointCount(0, text.length());
        System.out.println("字符数: " + charCount);
    }
}
public class SortObserver implements Observer {
    private final EditBox editBox;

    public SortObserver(EditBox editBox) {
        this.editBox = editBox;
    }

    @Override
    public void show() {
        String[] words = editBox.getText().split("\\s+");
        Map<String, Integer> wordCount = new HashMap<>(16);
        for (String word : words) {
            Integer count = wordCount.get(word);
            wordCount.put(word, count == null ? 1 : count + 1);
        }
        System.out.println(wordCount);
    }
}
public class WordObserver implements Observer {
    private final EditBox editBox;

    public WordObserver(EditBox editBox) {
        this.editBox = editBox;
    }

    @Override
    public void show() {
        String text = editBox.getText();
        List<String> words =
Arrays.stream(text.split("\\s+")).distinct().sorted().collect(Collectors.toList())
;
        System.out.println("单词: " + words);
    }
}
public abstract class AbstractEditBox {
    protected String text = "";
    protected List<Observer> observers = new ArrayList<>();

    public String getText() {
        return text.trim();
    }

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

```

```
64      public void removeObserver(Observer observer) {
65          observers.remove(observer);
66      }
67
68      public abstract void edit(String text);
69  }
70  public class EditBox extends AbstractEditBox {
71      @Override
72      public void edit(String text) {
73          System.out.println("edit: " + text);
74          this.text = text;
75          observers.forEach(Observer::show);
76      }
77  }
78  public class Main {
79      public static void main(String[] args) {
80          EditBox editBox = new EditBox();
81          editBox.addObserver(new CountObserver(editBox));
82          editBox.addObserver(new WordObserver(editBox));
83          editBox.addObserver(new SortObserver(editBox));
84          editBox.edit("Hello, my name is LiHua");
85      }
86  }
```

15、TCP连接多种状态【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。

类图;



代码:

```java
package com.headFirst.state;

/**
 *  LISTEN - 侦听来自远方TCP端口的连接请求；
    SYN-SENT -在发送连接请求后等待匹配的连接请求；
    SYN-RECEIVED - 在收到和发送一个连接请求后等待对连接请求的确认；
    ESTABLISHED- 代表一个打开的连接，数据可以传送给用户；
    FIN-WAIT-1 - 等待远程TCP的连接中断请求，或先前的连接中断请求的确认；
    FIN-WAIT-2 - 从远程TCP等待连接中断请求；
    CLOSE-WAIT - 等待从本地用户发来的连接中断请求；
    CLOSING -等待远程TCP对连接中断的确认；
    LAST-ACK - 等待原来发向远程TCP的连接中断请求的确认；
    TIME-WAIT -等待足够的时间以确保远程TCP接收到连接中断请求的确认；
    CLOSED - 没有任何连接状态；
    ==============================================================
 *  服务器端正常情况下TCP状态迁移：
 *  CLOSED->LISTEN->SYN收到 ->ESTABLISHED->CLOSE_WAIT->LAST_ACK->CLOSED
 * @author jwt
 * @date 2019年9月3日
 */
public class TcpConnection {

    private TcpState establisedState;//已连接，代表一个打开的连接，数据可以传送给用户；
    private TcpState closedState;   //关闭，没有任何连接状态；
    private TcpState listeningState; //监听，侦听来自远方TCP端口的连接请求；
    private TcpState state;

    public TcpConnection() {
        establisedState = new TcpEstablisedState(this);
        closedState = new TcpClosedState(this);
        listeningState = new TcpListeningState(this);
        state = closedState;//默认是关闭状态
    }
    public void open(){
        state.open();
    }
    public void close(){
        state.close();
    }
    public void acknowledge(){
        state.acknowledge();
    }




    public TcpState getEstablisedState() {
        return establisedState;
    }
    public TcpState getClosedState() {
        return closedState;
    }
    public TcpState getListeningState() {
        return listeningState;
```

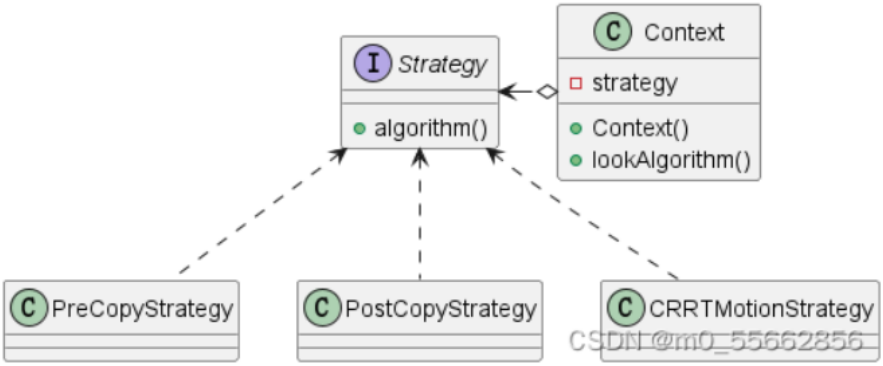
```
    }
    public TcpState getState() {
        return state;
    }
    public void setState(TcpState state) {
        this.state = state;
    }

}


package com.headFirst.state;

public interface TcpState {

    void open();
    void close();
    void acknowledge();

}

//具体的状态类
package com.headFirst.state;

public class TcpClosedState implements  TcpState {

    private TcpConnection connection;
    public TcpClosedState(TcpConnection connection){
        this.connection = connection;
    }
    @Override
    public void open() {
        System.out.println("CLOSED -> LISTEN:开始监听来自远方的tcp连接! ");
        connection.setState(connection.getListeningState());
    }
    @Override
    public void close() {

    }

    @Override
    public void acknowledge() {

    }

}

package com.headFirst.state;

public class TcpEstablisedState implements TcpState {
    private TcpConnection tcpConnection;
    public TcpEstablisedState(TcpConnection tcpConnection) {
        this.tcpConnection = tcpConnection;
    }
```

```
109
110     @Override
111     public void open() {
112
113     }
114
115     @Override
116     public void close() {
117         System.out.println("ESTABLISHED -> CLOSED: 连接已关闭! ");
118         tcpConnection.setState(tcpConnection.getClosedState());
119     }
120
121     @Override
122     public void acknowledge() {
123
124     }
125
126 }
127
128 package com.headFirst.state;
129
130 public class TcpListeningState implements TcpState{
131     private TcpConnection tcpConnection;
132     public TcpListeningState(TcpConnection tcpConnection) {
133         this.tcpConnection = tcpConnection;
134     }
135     @Override
136     public void open() {
137
138     }
139     @Override
140     public void close() {
141
142     }
143     @Override
144     public void acknowledge() {
145         System.out.println("LISTEN->ESTABLISHED: 确认，已连接! ");
146         tcpConnection.setState(tcpConnection.getEstablisedState());
147     }
148
149 }
150
151 package com.headFirst.state;
152
153 public class App {
154
155     public static void main(String[] args) {
156         TcpConnection connection = new TcpConnection();
157         connection.open();//开始监听tcp请求
158         connection.acknowledge();//已连接
159         connection.close();//关闭
160
161     }
162 }
```

16、虚拟机迁移算法

在某云计算模拟平台中提供了多种虚拟机迁移算法，例如动态迁移算法中的 PreCopy(预拷贝)算法、Post-Copy(后拷贝)算法、CR/RT-Motion算法等，用户可以灵活地选择所需的虚拟机迁移算法，也可以方便地增加新算法。现采用策略模式进行设计，绘制对应的类图并编程模拟实现。

类图：



代码：

```
public interface Strategy {
    public void algorithm();
}

public class PreCopyStrategy implements Strategy{
    @Override
    public void algorithm() {
        System.out.println("Use Pre-Copy algorithm");
    }
}
public class PostCopyStrategy implements Strategy{
    @Override
    public void algorithm() {
        System.out.println("Use Post-Copy algorithm");
    }
}
public class CRRTMotionStrategy implements Strategy{
    @Override
    public void algorithm() {
        System.out.println("Use CR/RT-Motion algorithm");
    }
}
public class Context {
    private Strategy strategy;
    public Context(Strategy strategy){
        this.strategy = strategy;
    }
```

```
    public void lookAlgorithm(){
        strategy.algorithm();
    }
}
public class Test {
    public static void main(String[] args) {
        Strategy preStrategy = new PreCopyStrategy();
        Strategy postStrategy = new PostCopyStrategy();
        Strategy CRRTStrategy = new CRRTMotionStrategy();

        Context preContext = new Context(preStrategy);
        Context postContex = new Context(postStrategy);
        Context CRRTContex = new Context(CRRTStrategy);

        preContext.lookAlgorithm();
        postContex.lookAlgorithm();
        CRRTContex.lookAlgorithm();
    }
}
```