设计模式期末复习

- **考试说明：**

考试时间：

2024 年 5 月 28 日，周二早 10:00~12:00，地点：待通知。

考试形式：

闭卷考试。

试题类型：选择题（20 分），连线题（10 分），简述题和程序填空题（30 分），综合题（40 分）

- **复习题如下：**

复习题目录

一、 程序填空题

(一) 创建型设计模式

1. 某系统提供一个简单计算器，具有简单的加法和减法功能，系统可以根据用户的选择实例化相应的操作类。现使用简单工厂模式设计该系统，类图如图 1 所示：

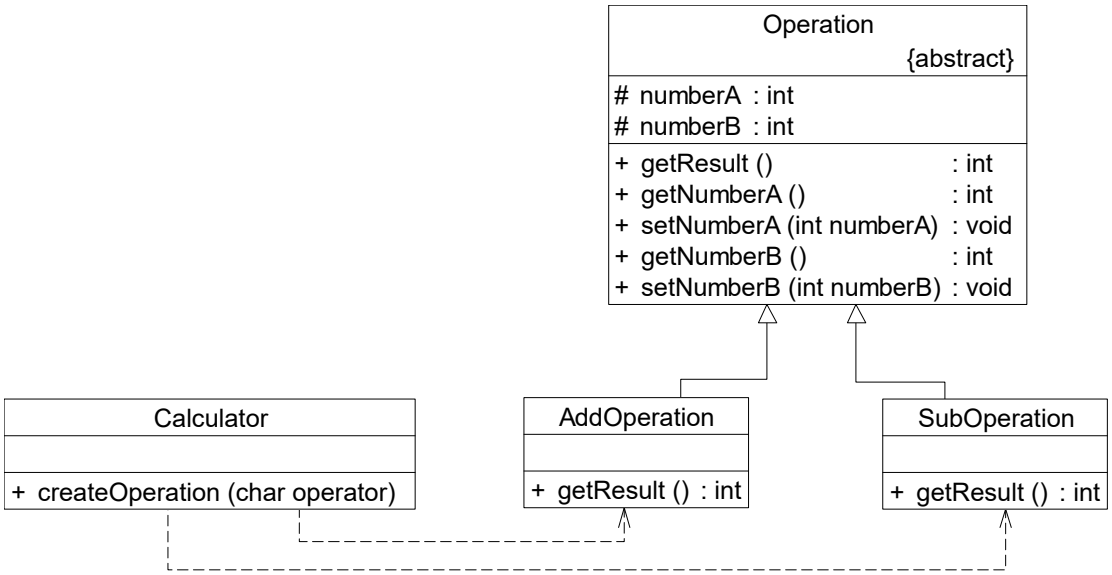


图 1 类图

在图 1 中，Operation 是抽象类，其中定义了抽象方法 getResult()，其子类 AddOperation 用于实现加法操作，SubOperation 用于实现减法操作，Calculator 是简单工厂类，工厂方法为 createOperation()，该方法接收一个 char 类型的字符参数，如果传入的参数为“+”，工厂方法返回一个 AddOperation 类型的对象，如果传入的参数为“-”，则返回一个 SubOperation 类型的对象。

【Java 代码】

```
abstract class Operation
{
    protected int numberA;
    protected int numberB;
    // numberA 和 numberB 的 Setter 方法和 Getter 方法省略
    public  ___(1)____int getResult();
}

class AddOperation extends Operation
{
    public int getResult()
    {
        return numberA + numberB;
    }
}
```

```
class SubOperation extends Operation
{
    public int getResult()
    {
        return numberA - numberB;
    }
}

class Calculator
{
    public  _____(2)_____  createOperation(char operator)
    {
        Operation op = null;
        ___(3)____
        {
            case '+':
                op =  ___(4)____;
                break;
            case '-':
                op =  ___(5)____;
                break;
        }
        ___(6)____;
    }
}

class Test
{
    public static void main(String args[])
    {
        int result;
        Operation op1 = Calculator.createOperation('+');
        op1.setNumberA(20);
        op1.setNumberB(10);
        result =  ___(7)____;
        System.out.println(result);
    }
}
```

参考答案：

(1) abstract；(2) static Operation；(3) switch(operator)；(4) new AddOperation()；(5) new SubOperation()；(6) return op；(7) op1.getResult()。

2. 某软件公司欲开发一个数据格式转换工具，可以将不同数据源如 txt 文件、数据库、Excel 表格中的数据转换成 XML 格式。为了让系统具有更好的扩展性，在未来支持新类型的数据源，开发人员使用工厂方法模式设计该转换工具的核心类。在工厂类中封装了具体转换类的初始化和创建过程，客户端只需使用工厂类即可获得具体的转换类对象，再调用其相应方法实现数据转换操作。其类图如图 1 所示：

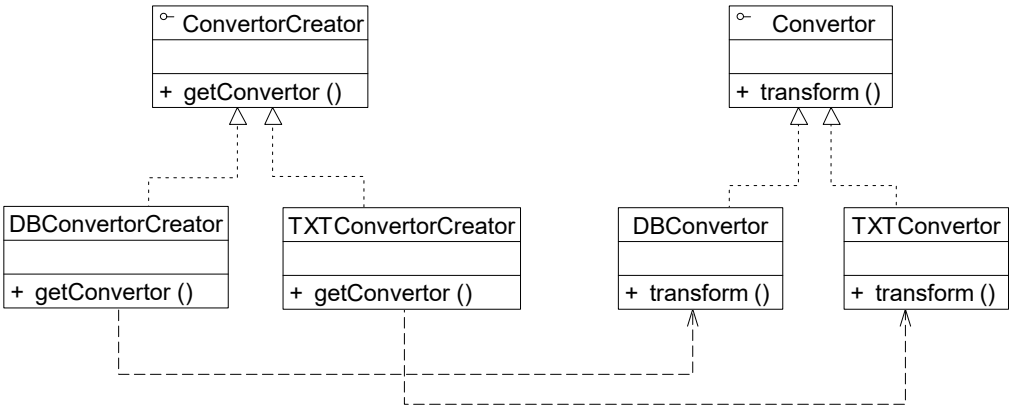


图 1 类图

在图 1 中，ConvertorCreator 是抽象工厂接口，它声明了工厂方法 getConvertor()，在其子类中实现该方法，用于创建具体的转换对象；Convertor 是抽象产品接口，它声明了抽象数据转换方法 transform()，在其子类中实现该方法，用于完成具体的数据转换操作。类 DBConvertor 和 TXTConvertor 分别用于将数据库中的数据和 txt 文件中的数据转换为 XML 格式。

【Java 代码】

```
interface ConvertorCreator
{
        ____(1)____;
}

interface Convertor
{
        public String transform();
}

class DBConvertorCreator implements ConvertorCreator
{
        public Convertor getConvertor()
        {
            ____(2)____;
        }
}

class TXTConvertorCreator implements ConvertorCreator
{
        public Convertor getConvertor()
```

```
        {
            _____(3)_____;
        }
}

class DBConvertor implements Convertor
{
        public String transform()
        {
            //实现代码省略
        }
}

class TXTConvertor implements Convertor
{
        public String transform()
        {
            //实现代码省略
        }
}

class Test
{
        public static void main(String args[])
        {
            ConvertorCreator creator;
            _____(4)_____;
            creator = new DBConvertorCreator();
            convertor = _____(5)_____;
            convertor.transform();
        }
}
```

如果需要针对一种新的数据源进行数据转换，该系统至少需要增加___(6)___个类。工厂方法模式体现了以下哪些面向对象设计原则？_____(7)_____。（多选）

A. 开闭原则　　B. 依赖倒转原则　　C. 接口隔离原则　　D. 单一职责原则　　E. 合成复用原则

参考答案：

(1) public Convertor getConvertor()；(2) return new DBConvertor()；(3) return new TXTConvertor()；(4) Convertor convertor；(5) creator.getConvertor()；(6) 2；(7) ABD。

3. 某手机游戏软件公司欲推出一款新的游戏软件，该软件能够支持 Symbian、Android 和 Windows Mobile 等多个主流的手机操作系统平台，针对不同的手机操作系统，该游戏软件提供了不同的游戏操作控制类和游戏界面控制类，并提供相应的工厂类来封装这些类的初始

化。软件要求具有较好的扩展性以支持新的操作系统平台，为了满足上述需求，采用抽象工厂模式进行设计所得类图如图 1 所示：

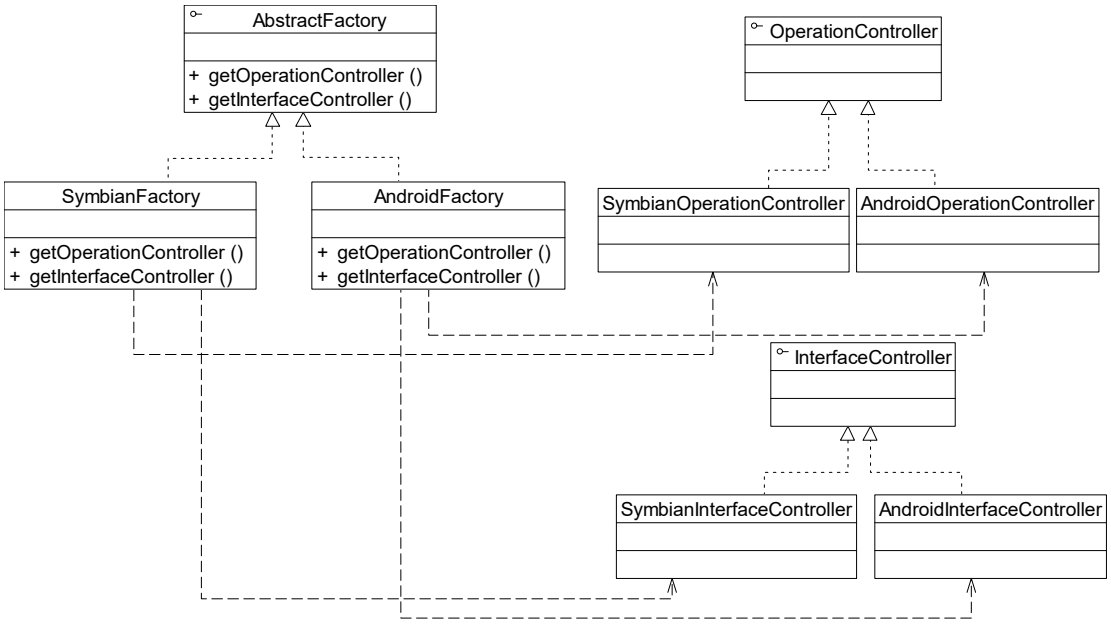


图 1 类图

在该设计方案中，具体工厂类如 SymbianFactory 用于创建 Symbian 操作系统平台下的游戏操作控制类 SymbianOperationController 和游戏界面控制类 SymbianInterfaceController，再通过它们的业务方法来实现对游戏软件的初始化和运行控制。

【Java 代码】

```
interface AbstractFactory
{
    public OperationController getOperationController();
    public InterfaceController getInterfaceController();
}

interface OperationController
{
    public void init();
    //其他方法声明省略
}

interface InterfaceController
{
    public void init();
    //其他方法声明省略
}

class SymbianFactory implements AbstractFactory
{
    public OperationController getOperationController()
    {
```

```
        ____(1)____;
    }

    public InterfaceController getInterfaceController()
    {
        ____(2)____;
    }
}

class AndroidFactory ____(3)____
{
    public OperationController getOperationController()
    {
        return new AndroidOperationController();
    }

    public InterfaceController getInterfaceController()
    {
        return new AndroidInterfaceController();
    }
}

class SymbianOperationController ____(4)____
{
    public void init() {
        //实现代码省略
    }
    //其他方法声明省略
}

class AndroidOperationController ____(5)____
{
    public void init() {
        //实现代码省略
    }
    //其他方法声明省略
}

class SymbianInterfaceController implements InterfaceController
{
    public void init() {
        //实现代码省略
    }
    //其他方法声明省略
```

```
}

class AndroidInterfaceController implements InterfaceController
{
    public void init() {
        //实现代码省略
    }
    //其他方法声明省略
}

class Test
{
    public static void main(String args[])
    {
        AbstractFactory af;
        _____(6)_____oc;
        _____(7)_____ic;
        af = new SymbianFactory();
        oc = _____(8)_____;
        ic = _____(9)_____;
        oc.init();
        ic.init();
    }
}
```

如何需要在上述设计方案中增加对 Windows Mobile 操作系统的支持，需对该设计方案进行哪些调整，简单说明实现过程。

参考答案：

(1) return new SymbianOperationController()；(2) return new SymbianInterfaceController()；(3) implements AbstractFactory；(4) implements OperationController；(5) implements OperationController；(6) OperationController；(7) InterfaceController；(8) af.getOperationController()；(9) af.getInterfaceController()。

如果需要增加对 Windows Mobile 操作系统的支持，需要增加三个类，其中 WindowsOperationController 作为 OperationController 接口的子类，WindowsInterfaceController 作为 InterfaceController 接口的子类，再对应增加一个具体工厂类 WindowsFactory 实现 AbstractFactory 接口，并实现在其中声明的工厂方法，创建 WindowsOperationController 对象和 WindowsInterfaceController 对象。

4. 为了避免监控数据显示不一致并节省系统资源，在某监控系统的设计方案中提供了一个主控中心类，该主控中心类使用单例模式进行设计，类图如图 1 所示：

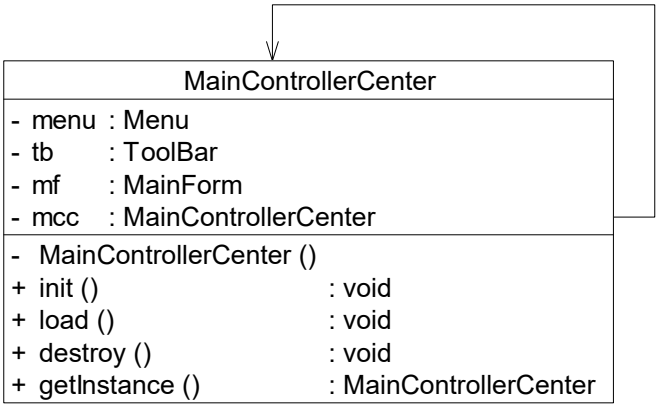


图 1 类图

在图 1 中，主控中心类 MainControllerCenter 是单例类，它包含一系列成员对象并可以初始化、显示和销毁成员对象，对应的方法分别为 init()、load()和 destroy()，此外还提供了静态工厂方法 getInstance()用于创建 MainControllerCenter 类型的单例对象。

【Java 代码】

```
class MainControllerCenter
{
    private Menu menu;   //主控中心菜单
    private ToolBar tb;   //主控中心工具栏
    private MainForm mf;   //主控中心主窗口
    private  ___(1)____MainControllerCenter mcc;

    ___(2)____MainControllerCenter{
    }

    public void init()
    {
        menu = new Menu();
        tb = new ToolBar();
        mf = new MainForm();
    }

    public void load()
    {
        menu.display();
        tb.display();
        mf.display();

    }

    public void destroy()
    {
        menu.destroy();
        tb.destroy();
```

```
            mf.destroy();
        }

        public static MainControllerCenter getInstance()
        {
            if(mcc==null)
            {
                ____________(3)____________;
            }
            return mcc;
        }
}

class Test
{
        public static void main(String args[])
        {
            MainControllerCenter mcc1,mcc2;
            mcc1 = MainControllerCenter.getInstance();
            mcc2 = MainControllerCenter.getInstance();
            System.out.println(mcc1==mcc2);
        }
}
//其他代码省略
```

编译并运行上述代码，输出结果为_____(4)_____。

在本实例中，使用了_____(5)_____（填写懒汉式或饿汉式）单例模式，其主要优点是_____(6)_____，主要缺点是_____(7)_____。

参考答案：

(1) static；(2) private；(3) mcc = new MainControllerCenter()；(4) true；(5) 懒汉式；(6)使用的时候再实例化可节约资源；(7) 必须处理好多线程访问问题，速度和反应时间相对“饿汉式”较慢。

(二) 结构型设计模式

1. 某公司欲开发一款儿童玩具汽车，为了更好地吸引小朋友的注意力，该玩具汽车在移动过程中伴随着灯光闪烁和声音提示，在该公司以往的产品中已经实现了控制灯光闪烁和声音提示的程序，为了重用先前的代码并且使得汽车控制软件具有更好的灵活性和扩展性，使用适配器模式设计该系统，所得类图如图 1 所示。

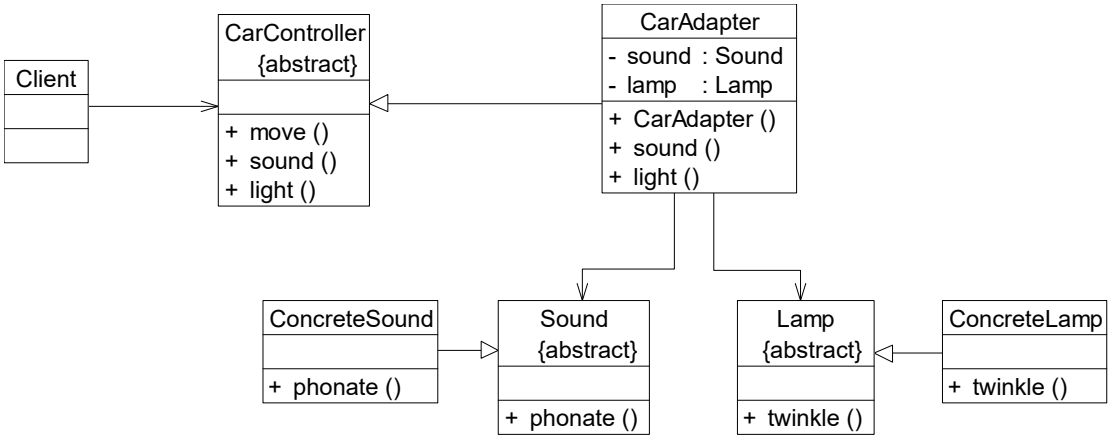


图 1 类图

在图 1 中，CarController 类是汽车控制器，它包括三个方法用于控制汽车的行为，其中 move()用于控制汽车的移动，sound()用于控制汽车的声音，light()用于控制汽车灯光的闪烁，sound()和 light()是抽象方法。Sound 类是抽象声音类，其方法 phonate()用于实现声音提示，在其子类 ConcreteSound 中实现了该方法；Lamp 类是灯光类，其方法 twinkle()用于实现灯光闪烁，在其子类 ConcreteLamp 中实现了该方法。CarAdapter 充当适配器，它通过分别调用 Sound 类的 phonate()方法和 Lamp 类的 twinkle()方法实现声音播放和灯光闪烁。

【Java 代码】

```
abstract class Sound    //抽象声音类
{    public abstract void phonate();    }

class ConcreteSound extends Sound    //具体声音类
{
      public void phonate()
      {    System.out.println("声音播放！");    }
}

abstract class Lamp    //抽象灯光类
{    public abstract void twinkle();    }

class ConcreteLamp extends Lamp    //具体灯光类
{
      public void twinkle()
      {    System.out.println("灯光闪烁！");    }
}

_____(1)_____CarController    //汽车控制器
{
      public void move()
      {    System.out.println("汽车移动！");    }
      public abstract void sound();
      public abstract void light();
}
```

```
class CarAdapter _____(2)_____ //汽车适配器
{
    private Sound sound;
    private Lamp lamp;

    public CarAdapter(Sound sound,Lamp lamp)
    {
        _____(3)_____;
        _____(4)_____;
    }

    public void sound()
    {
        _____(5)_____;   //声音播放
    }

    public void light()
    {
        _____(6)_____;   //灯光闪烁
    }
}

class Client
{
    public static void main(String args[])
    {
        Sound sound;
        Lamp lamp;
        CarController car;

        sound = new ConcreteSound();
        lamp = new ConcreteLamp();
        car = _____(7)_____;

        car.move();
        car.sound();
        car.light();
    }
}
```

在本实例中，使用了_____(8)_____（填写类适配器或对象适配器）模式。

参考答案：

(1) abstract class；(2) extends CarController；(3) this.sound = sound；(4) this.lamp = lamp；(5) sound.phonate()；(6) lamp.twinkle()；(7) new CarAdapter(sound, lamp)；(8) 对象适配器。

2. 现欲实现一个图像浏览系统，要求该系统能够显示 BMP、JPEG 和 GIF 三种格式的文件，并且能够在 Windows 和 Linux 两种操作系统上运行。系统首先将 BMP、JPEG 和 GIF 三种格式的文件解析为像素矩阵，然后将像素矩阵显示在屏幕上。系统需具有较好的扩展性以支持新的文件格式和操作系统。为满足上述需求并减少所需生成的子类数目，采用桥接设计模式进行设计所得类图如图 1 所示。

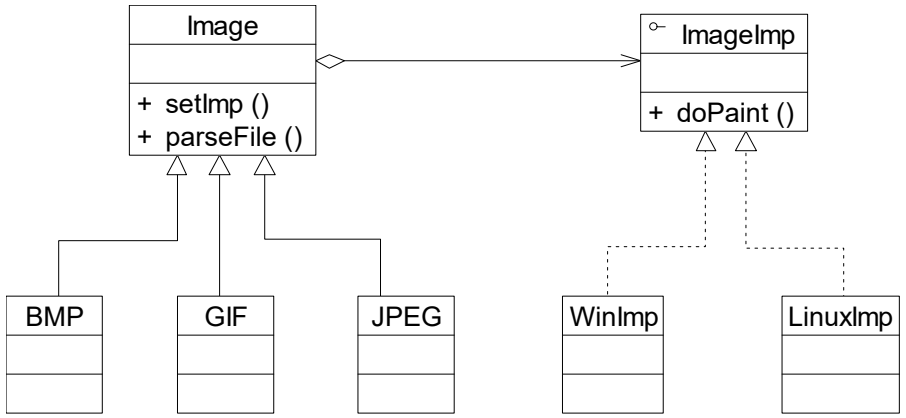


图 1 类图

采用该设计模式的原因在于：系统解析 BMP、JPEG 与 GIF 文件的代码仅与文件格式相关，而在屏幕上显示像素矩阵的代码则仅与操作系统相关。

【Java 代码】

```
class Matrix {   //各种格式的文件最终都被转化为像素矩阵
    //此处代码省略
}
interface ImageImp {
    public void doPaint(Matrix m);   //显示像素矩阵 m
}

class WinImp implements ImageImp {
    public void doPaint(Matrix m) {/*调用 Windows 系统的绘制函数绘制像素矩阵*/}
}

class LinuxImp implements ImageImp {
    public void doPaint(Matrix m) {/*调用 Linux 系统的绘制函数绘制像素矩阵*/}
}

abstract class Image {
    public void setImp(ImageImp imp) {
    _____(1)_____ = imp; }
    public abstract void parseFile(String fileName);
    protected _____(2)_____ imp;
}

class BMP extends Image {
```

```
    public void parseFile(String fileName) {
        //此处解析 BMP 文件并获得一个像素矩阵对象 m
        _____(3)_____;//显示像素矩阵 m
    }
}

class GIF extends Image {
    //此处代码省略
}

class JPEG extends Image {
    //此处代码省略
}

public class Main{
    public static void main(String[] args)
    {
        //在 Windows 操作系统上查看 demo.bmp 图像文件
        Image image1 = _____(4)_____;
        ImageImp imageImp1 = _____(5)_____;
        _____(6)_____;
        image1.parseFile("demo.bmp");
    }
}
```

现假设该系统需要支持 10 种格式的图像文件和 5 种操作系统，不考虑类 Matrix 和类 Main，若采用桥接模式则至少需要设计_____(7)_____个类。

参考答案：

(1) this.imp；(2) ImageImp；(3) imp.doPaint(m)；(4) new BMP()；(5) new WinImp()；(6) image1.setImp(imageImp1)；(7) 17。

3. 某公司的组织结构图如图 1 所示，现采用组合设计模式来设计，得到如图 2 所示的类图。

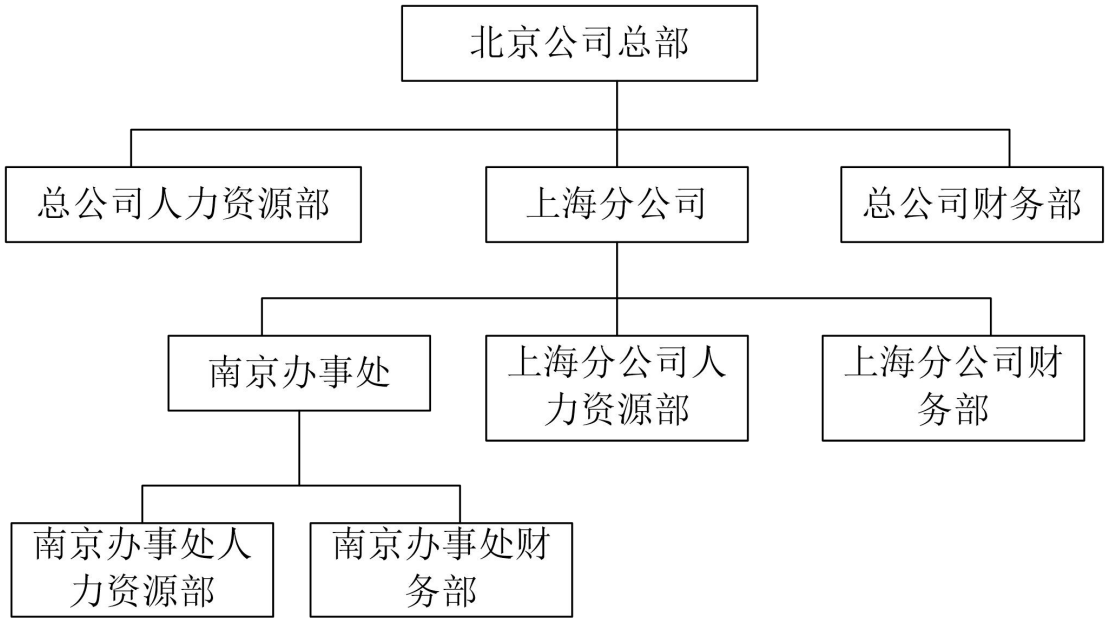


图 1 组织结构图

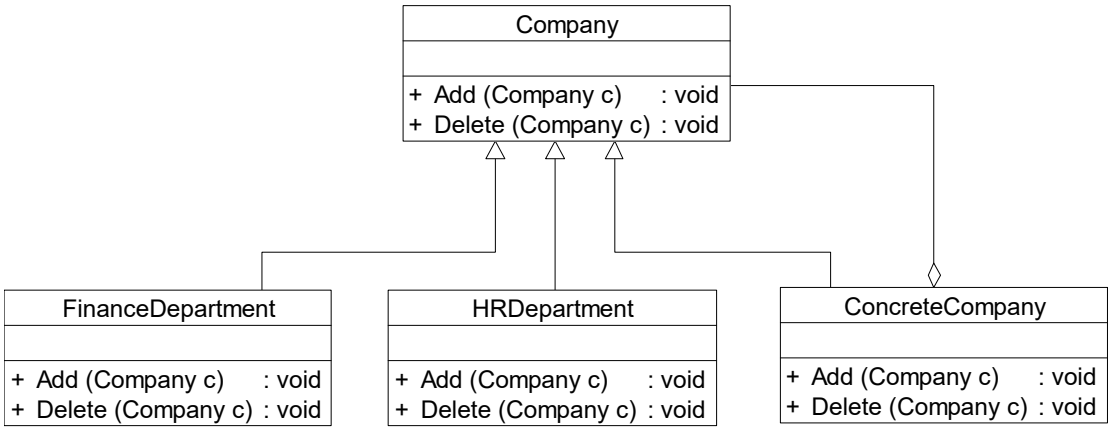


图 2 类图

其中 Company 为抽象类，定义了在组织结构图上添加（Add）和删除（Delete）分公司/办事处或者部门的方法接口。类 ConcreteCompany 表示具体的分公司或者办事处，分公司或办事处下可以设置不同的部门。类 HRDepartment 和 FinanceDepartment 分别表示人力资源部和财务部。

【Java 代码】

```
import java.util.*;
___(1)____Company {
    protected String name;
    public Company(String name) {___(2)____= name; }
    public abstract void Add(Company c); // 增加子公司、办事处或部门
    public abstract void Delete(Company c); // 删除子公司、办事处或部门
}

class ConcreteCompany extends Company {
    private List<___(3)____> children = new ArrayList<___(4)____>(); // 存储子公司、办事处或部门
```

```
    public ConcreteCompany(String name) { super(name); }
    public void Add(Company c) {___(5)____.add(c); }
    public void Delete(Company c) {___(6)____.remove(c); }
}

class HRDepartment extends Company {
    public HRDepartment(String name) { super(name); }
    // 其他代码省略
}

class FinanceDepartment extends Company {
    public FinanceDepartment(String name) { super(name); }
    // 其他代码省略
}

public class Test {
    public static void main(String[] args) {
        ConcreteCompany root = new ConcreteCompany("北京总公司");
        root.Add(new HRDepartment("总公司人力资源部"));
        root.Add(new FinanceDepartment("总公司财务部"));
        ConcreteCompany comp = new ConcreteCompany("上海分公司");
        comp.Add(new HRDepartment("上海分公司人力资源部"));
        comp.Add(new FinanceDepartment("上海分公司财务部"));
        ___(7)____;
        ConcreteCompany comp1 = new ConcreteCompany("南京办事处");
        comp1.Add(new HRDepartment("南京办事处人力资源部"));
        comp1.Add(new FinanceDepartment("南京办事处财务部"));
        ___(8)____; // 其他代码省略
    }
}
```

参考答案：

(1) abstract class；(2) this.name；(3) Company；(4) Company；(5) children；(6) children；(7) root.Add(comp)；(8) comp.Add(comp1)。

4. 某公司欲开发一套手机来电提示程序，在最简单的版本中，手机在接收到来电时会发出声音来提醒用户；在振动版本中，除了声音外，在来电时手机还能产生振动；在更高级的版本中手机不仅能够发声和产生振动，而且还会有灯光闪烁提示。现采用装饰设计模式来设计，得到如图 1 所示的类图。

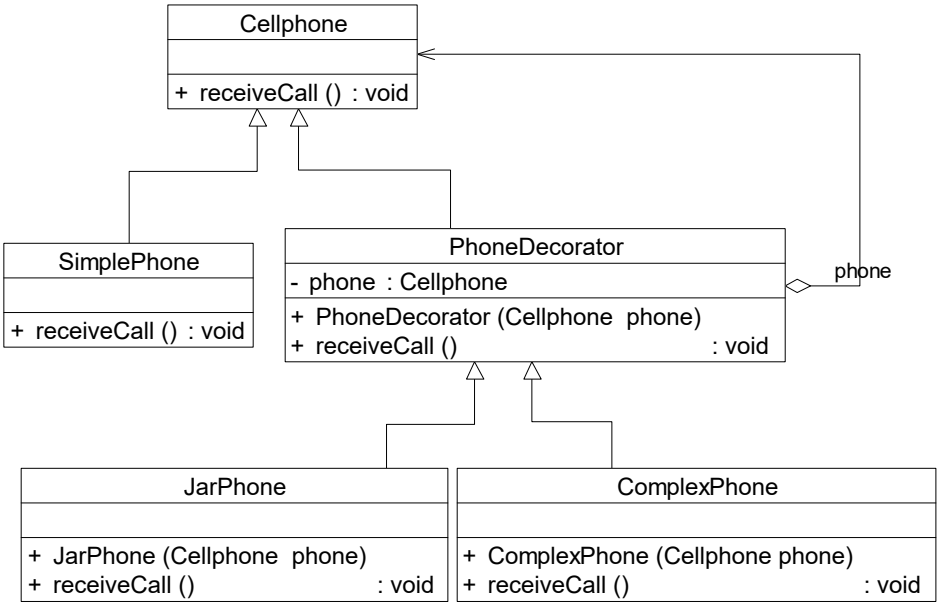


图 1 类图

其中 Cellphone 为抽象类，声明了来电方法 receiveCall()，SimplePhone 为简单手机类，提供了声音提示，JarPhone 和 ComplexPhone 分别提供了振动提示和灯光闪烁提示。PhoneDecorator 是抽象装饰者，它维持一个对父类对象的引用。

【Java 代码】

```
abstract class Cellphone
{   public abstract void receiveCall();   }

class SimplePhone extends Cellphone
{
    public void receiveCall()
    {   System.out.println("声音提示");   }
}

class PhoneDecorator extends Cellphone
{
    private     (1)     phone=null;
    public PhoneDecorator(Cellphone phone)
    {
        if(phone!=null)
        {     (2)     ;   }
        else
        {   this.phone=new SimplePhone();   }
    }
    public void receiveCall()
    {     (3)     ;   }
}

class JarPhone extends PhoneDecorator
```

```
{
    public JarPhone(Cellphone phone)
    {   ________(4)________;   }
    public void receiveCall()
    {
        super.receiveCall();
        System.out.println("振动提示");
    }
}

class ComplexPhone extends PhoneDecorator
{
    public ComplexPhone(Cellphone phone)
    {   ________(5)________;   }
    public void receiveCall()
    {
        super.receiveCall();
        System.out.println("灯光闪烁提示");
    }
}

class Client
{
    public static void main(String a[])
    {
        Cellphone p1=new ________(6)________;   //创建具有声音提示的手机
        p1.receiveCall();
        Cellphone p2=new ________(7)________;   //创建具有声音提示和振动提示的手机
        p2.receiveCall();
        Cellphone p3=new ________(8)________;   //创建具有声音提示、振动提示和灯光提示的手机
        p3.receiveCall();
    }
}
```

参考答案：

(1) Cellphone；(2) this.phone = phone；(3) phone.receiveCall()；(4) super(phone)；(5) super(phone)；(6) SimplePhone()；(7) JarPhone(p1)；(8) ComplexPhone(p2)。

5. 某信息系统需要提供一个数据读取和报表显示模块，可以将来自不同类型文件中的数据转换成 XML 格式，并对数据进行统计和分析，然后以报表方式来显示数据。由于该过程需要涉及到多个类，因此使用外观模式进行设计，其类图如图 1 所示：

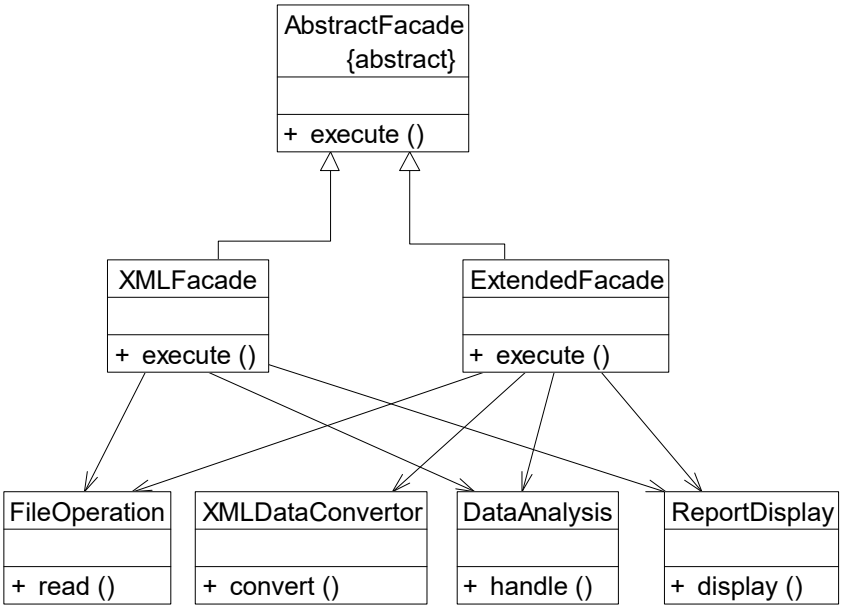


图 1 类图

在图 1 中，FileOperation 类用于读取文件、XMLDataConvertor 类用于将不同格式的文件转换为 XML 格式、DataAnalysis 类用于对 XML 数据进行统计分析、ReportDisplay 类用于显示报表。为了让系统具有更好的扩展性，在系统设计中引入了抽象外观类 AbstractFacade，它拥有多个不同的子类，如 XMLFacade，它用于与读取、分析和显示 XML 数据的类交互，ExtendedFacade 类用于与读取、转换、分析和显示非 XML 数据的类交互。

【Java 代码】

```
class FileOperation
{
    public String read(String fileName)
    {   //读取文件代码省略   }
}

class XMLDataConvertor
{
    public String convert(String fileStr)
    {   //文件格式转换代码省略   }
}

class DataAnalysis
{
    public String handle(String xmlStr)
    {   //数据分析统计代码省略   }
}

class ReportDisplay
{
    public void display(String xmlStr)
    {   //报表显示代码省略   }
```

```
}

_____(1)_____AbstractFacade
{
    public abstract void execute(String fileName);
}

class XMLFacade extends AbstractFacade
{
    private FileOperation fo;
    private DataAnalysis da;
    private ReportDisplay rd;

    public XMLFacade()
    {
        fo = new FileOperation();
        da = new DataAnalysis();
        rd = new ReportDisplay();
    }

    public void execute(String fileName)
    {
        String str = _____(2)_____; //读取文件
        String strResult = _____(3)_____; //分析数据
        _____(4)_____; //显示报表
    }
}

class ExtendedFacade extends AbstractFacade
{
    private FileOperation fo;
    private XMLDataConvertor dc;
    private DataAnalysis da;
    private ReportDisplay rd;

    public ExtendedFacade()
    {
        fo = new FileOperation();
        dc = new XMLDataConvertor();
        da = new DataAnalysis();
        rd = new ReportDisplay();
    }

    public void execute(String fileName)
```

```
    {
        String str = _____(5)_____; //读取文件
        String strXml = _____(6)_____; //转换文件
        String strResult = _____(7)_____; //分析数据
        _____(8)_____; //显示报表
    }
}

class Test
{
    public static void main(String args[])
    {
        AbstractFacade facade;
        facade = _____(9)_____;
        facade.execute("file.xml");
    }
}
```

参考答案：

(1) abstract class； (2) fo.read(fileName)； (3) da.handle(str)； (4) rd.display(strResult)； (5) fo.read(fileName)；(6) dc.convert(str)；(7) da.handle(strXml)；(8) rd.display(strResult)；(9) new XMLFacade()。

6. 某信息咨询公司推出收费的在线商业信息查询模块，需要对查询用户进行身份验证并记录查询日志，以便根据查询次数收取查询费用，现使用代理模式设计该系统，所得类图如图 1 所示：

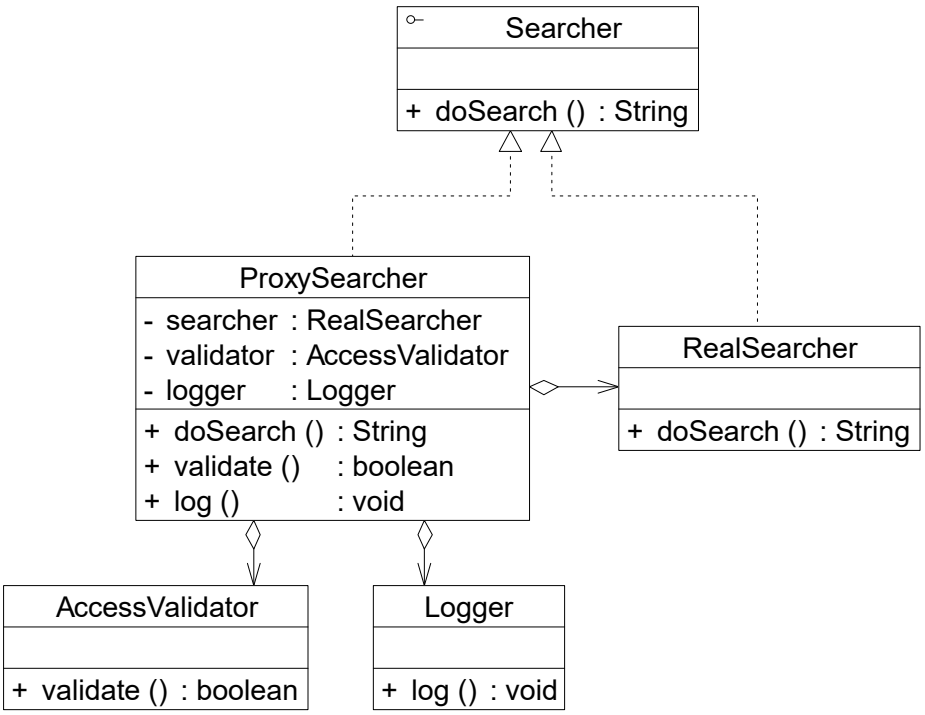


图 1 类图

在图 1 中，AccessValidator 类用于验证用户身份，它提供方法 validate()来实现身份验证；Logger 类用于记录用户查询日志，它提供方法 log()来保存日志；RealSearcher 类实现查询功能，它提供方法 doSearch()来查询信息。ProxySearcher 作为查询代理，维持对 RealSearcher 对象、AccessValidator 对象和 Logger 对象的引用。

【Java 代码】

```
class AccessValidator
{
    public boolean validate(String userId)
    {
        //身份验证实现代码省略
    }
}

class Logger
{
    public void log(String userId)
    {
        //日志记录实现代码省略
    }
}

interface Searcher
{
    public String doSearch(String userId,String keyword);
}

class RealSearcher implements Searcher
{
    public String doSearch(String userId,String keyword)
    {
        //信息查询实现代码省略
    }
}

class ProxySearcher  _____(1)_____
{
    private RealSearcher searcher = new RealSearcher();
    private AccessValidator validator;
    private Logger logger;

    public String doSearch(String userId,String keyword)
    {
        //如果身份验证成功，则执行查询
```

```
            if(_____(2)_____)
            {
                String result = searcher.doSearch(userId,keyword);
                _____(3)_____; //记录查询日志
                _____(4)_____; //返回查询结果
            }
            else
            {
                return null;
            }
        }

        public boolean validate(String userId)
        {
            validator = new AccessValidator();
            _____(5)_____;
        }

        public void log(String userId)
        {
            logger = new Logger();
            _____(6)_____;
        }
}

class Test
{
        public static void main(String args[])
        {
            _____(7)_____;   //针对抽象编程，客户端无须分辨真实主题类和代理类
            searcher = new ProxySearcher();
            String result = searcher.doSearch("Sunny","Money");
            //此处省略后续处理代码
        }
}
```

参考答案：

(1) implements Searcher； (2) validate(userId)； (3) log(userId)； (4) return result； (5) return validator.validate(userId)；(6) logger.log(userId)；(7) Searcher searcher。

(三)　行为型设计模式

1.　已知某企业欲开发一家用电器遥控系统，即用户使用一个遥控器即可控制某些家用电器的开与关。遥控器如图 1 所示。该遥控器共有 4 个按钮，编号分别是 0 至 3，按钮 0 和 2 能够遥控打开电器 1 和电器 2，按钮 1 和 3 则能遥控关闭电器 1 和电器 2。由于遥控系统需要支持形式多样的电器，因此，该系统的设计要求具有较高的扩展性。现假设需要控制客厅电视和卧室电灯，对该遥控系统进行设计所得类图如图 2 所示。

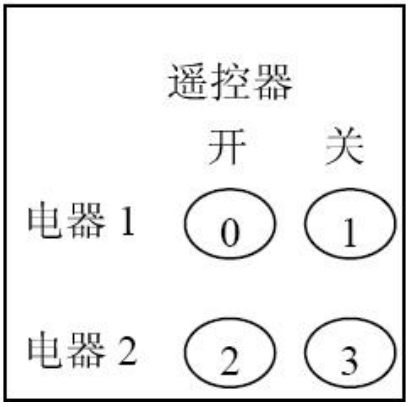


图 1　遥控器

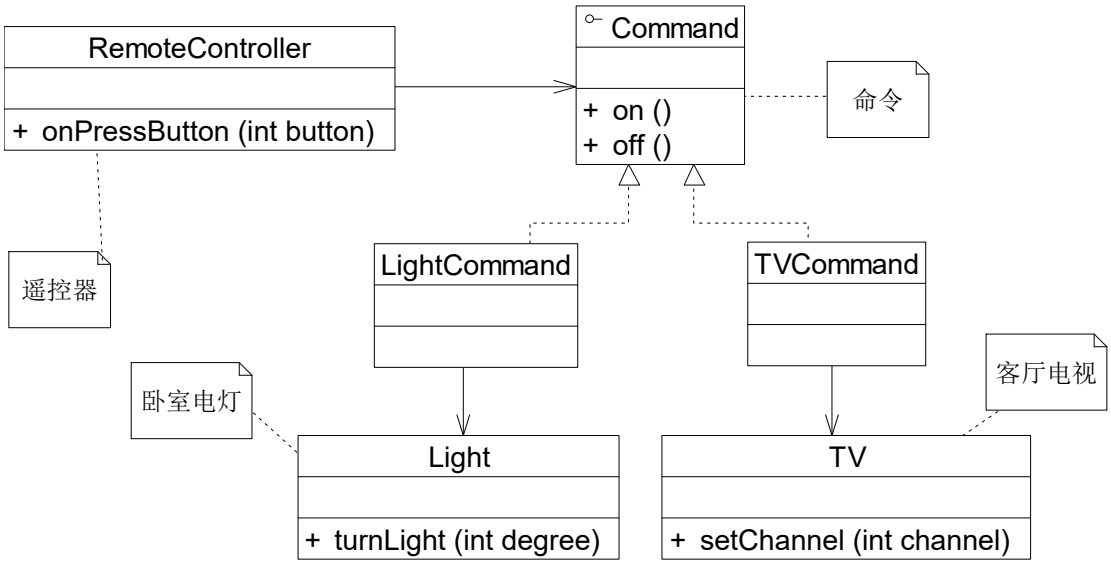


图 2　设计类图

图 2 中，类 RemoteController 的方法 onPressButton(int button)表示当遥控器按钮按下时调用的方法，参数为按键的编号；Command 接口中 on 和 off 方法分别用于控制电器的开与关；Light 中 turnLight(int degree)方法用于调整电灯灯光的强弱，参数 degree 值为 0 时表示关灯，值为 100 时表示开灯并且将灯光亮度调整到最大；TV 中 setChannel(int channel)方法表示设置电视播放的频道，参数 channel 值为 0 时表示关闭电视，为 1 时表示开机并将频道切换为第 1 频道。

【Java 代码】

```
class Light{ //电灯类
    public void turnLight(int degree) {//调整灯光亮度，0 表示关灯，100 表示亮度最大}
}
```

```
class TV{ //电视机类
    public void setChannel(int channel) {//0 表示关机，1 表示开机并切换到 1 频道}
}

interface Command{ //抽象命令类
    void on();
    void off();
}

class RemoteController{ //遥控器类
    //遥控器有 4 个按钮，按照编号分别对应 4 个 Command 对象
    protected Command[] commands = new Command[4];
    //按钮被按下时执行命令对象中的命令
    public void onPressButton(int button) {
        if(button%2==0) commands[button].on();
        else commands[button].off();
    }

    public void setCommand(int button, Command command){
    __(1)___  = command; //设置每个按钮对应的命令对象
    }
}

class LightCommand implements Command{   //电灯命令类
    protected Light light; //指向要控制的电灯对象
    public void on() {light.turnLight(100);}
    public void off() {light.__(2)___;}
    public LightCommand(Light light) {this.light = light; }
}

class TVCommand implements Command{ //电视机命令类
    protected TV tv;
    public void on() {tv.__(3)___;}
    public void off() {tv.setChannel(0);}
    public TVCommand(TV tv) {this.tv = tv;}
}

public class Test
{
    public static void main(String args[])
    {
        //创建电灯和电视对象
        Light light = new Light();
```

```
        TV tv = new TV();
        LightCommand lightCommand = new LightCommand(light);
        TVCommand tvCommand = new TVCommand(tv);
        RemoteController remoteController = new RemoteController();
        //设置按钮和命令对象
        remoteController.setCommand(0,  (4)  );
        ...//此处省略设置按钮 1、按钮 2 和按钮 3 的命令对象代码
    }
```

本题中，应用命令模式能够有效地让类 (5) 和类 (6) 、类 (7) 之间的耦合性降至最小。

参考答案：

(1) commands[button]；(2) turnLight(0)；(3) setChannel(1)；(4) lightCommand 或 tvCommand；(5) RemoteController；(6) Light；(7) TV。【注：(6)和(7)可以互换】

2. 某软件公司欲基于迭代器模式开发一套用于遍历数组元素的类库，其基本结构如图 1 所示：

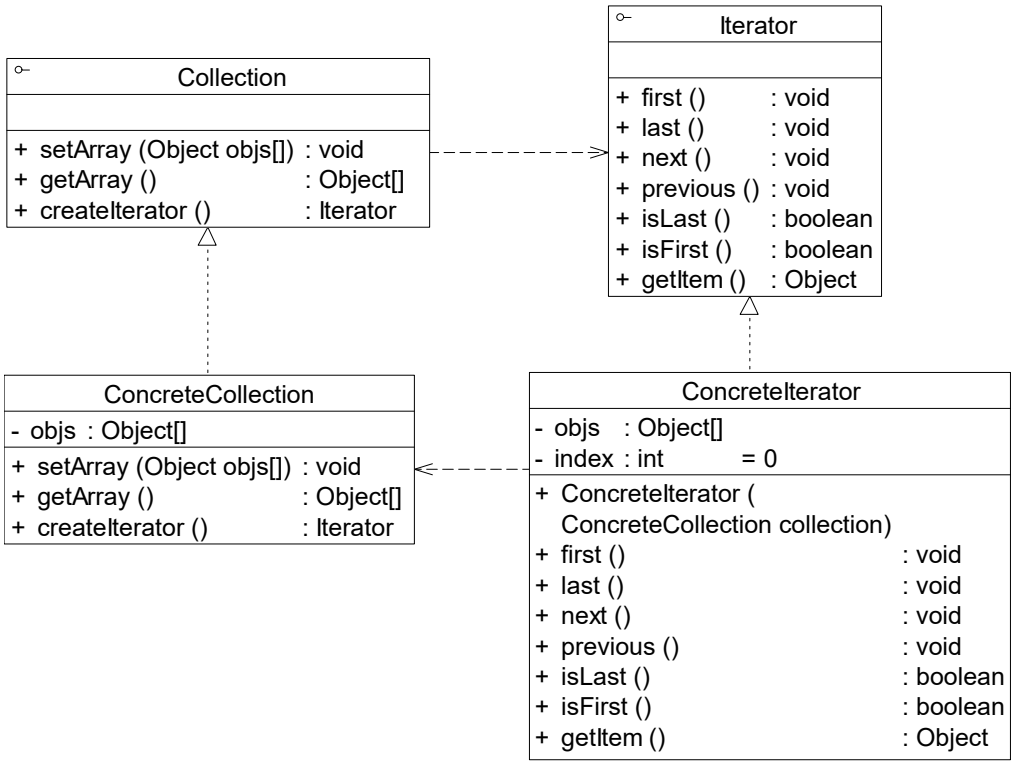


图 1 类图

在图 1 中，Collection 类是抽象聚合类，ConcreteCollection 类是具体聚合类，Iterator 类是抽象迭代器类，ConcreteIterator 类是具体迭代器类。在聚合类中提供了创建迭代器的工厂方法 createIterator()和数组的 Setter 和 Getter 方法，在迭代器中提供了用于遍历数组元素的相关方法，如 first()、last()、next()等。

【Java 代码】

```
interface Collection
{
```

```
    public void setArray(Object objs[]);   //设置数组
    public Object[] getArray();   //获取数组
    public Iterator createIterator();   //创建迭代器
}

interface Iterator
{
    public void first(); //索引指向第一个元素
    public void last(); //索引指向最后一个元素
    public void next(); //索引指向下一个元素
    public void previous(); //索引指向上一个元素
    public boolean isLast();   //判断是否最后一个元素
    public boolean isFirst(); //判断是否第一个元素
    public Object getItem();   //获取当前索引所指向的元素
}

class ConcreteCollection implements Collection
{
    private Object[] objs;

    public void setArray(Object objs[])
    {   this.objs = objs;   }

    public Object[] getArray()
    {   return this.objs;   }

    public Iterator createIterator()
    {   return ____(1)____; }
}

class ConcreteIterator implements Iterator
{
    private Object[] objs;
    private int index=0;   //索引变量，初值为 0

    public ConcreteIterator(ConcreteCollection collection)
    {   this.objs = ____(2)____;   }
    public void first()
    {   index = 0;   }
    public void last()
    {   ____(3)____;   }
    public void next()
    {
        if(index<objs.length)
```

```
            {
                ____(4)____;
            }
        }
        public void previous()
        {
            if(index>=0)
            {
                ____(5)____;
            }
        }
        public boolean isLast()
        {  ____(6)____;  }
        public boolean isFirst()
        {  ____(7)____;  }
        public Object getItem()
        {   return objs[index];   }
}

class Test
{
        public static void main(String args[])
        {
            Collection collection;
            collection = new ConcreteCollection();
            Object[] objs={"北京","上海","广州","深圳","长沙"};
            collection.setArray(objs);
            Iterator i = ____(8)____;
            i.last();
            //逆向遍历所有元素
            while(____(9)____)
            {
                System.out.println(i.getItem().toString());
                ____(10)____;
            }
        }
}
```

参考答案：

(1) new ConcreteIterator(this)；(2) collection.getArray()；(3) objs.length-1；(4) index ++；(5) index -- ； (6) return index==objs.length ； (7) return index==-1 ； (8) collection.createIterator() ；(9) !i.isFirst()；(10) i.previous()。

3. 某公司欲开发一套机房监控系统，如果机房达到某一指定温度，传感器将作出反应，将

信号传递给响应设备，如警示灯将闪烁、报警器将发出警报、安全逃生门将自动开启、隔热门将自动关闭等，每一种响应设备的行为由专门的程序来控制。为支持将来引入新类型的响应设备，采用观察者模式设计该系统，类图如图 1 所示：

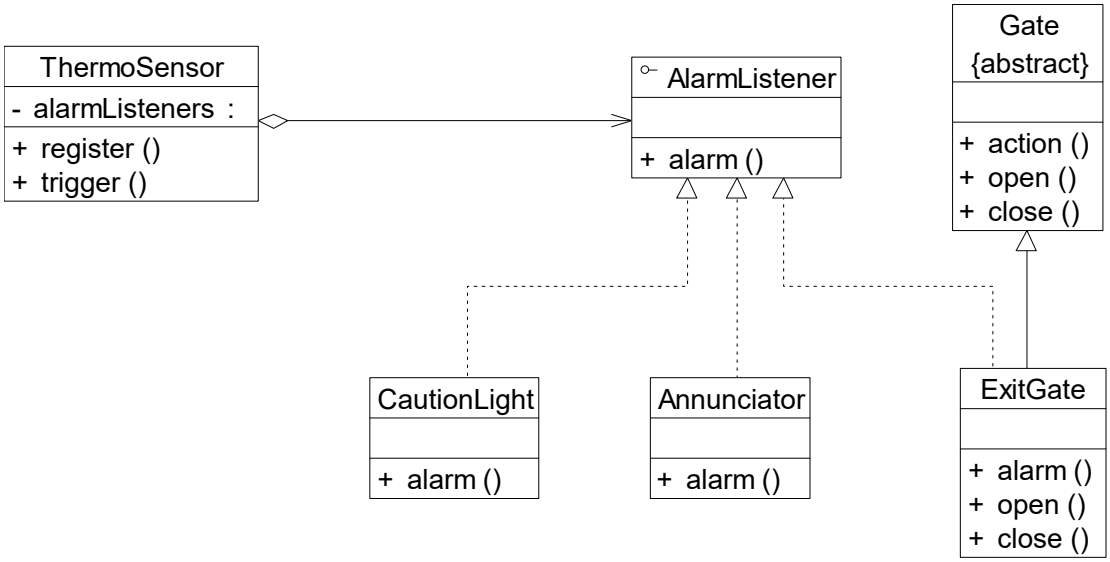


图 1 类图

在图 1 中，ThermoSensor 是温度传感器，定义了增加响应设备的方法 register()和触发方法 trigger()。AlarmListener 接口是抽象响应设备类，声明了警示方法 alarm()，而警示灯类 CautionLight、报警器类 Annunciator 和安全门类 ExitGate 是具体响应设备类，它们实现了警示方法 alarm()。ExitGate 是抽象类 Gate 类的子类，它将实现在 Gate 类中声明的 open()方法和 close()方法，用于开启逃生门并关闭隔热门，Gate 类中的 action()方法用于同时执行 open()方法和 close()方法。

【Java 代码】

```
import java.util.*;

_____(1)_____
{
    public void alarm();
}

abstract class Gate
{
    public void action()
    {
        open();
        close();
    }

    public abstract void open();
    public abstract void close();
}
```

```
class CautionLight implements AlarmListener
{
    public void alarm()
    {
        System.out.println("警示灯闪烁！");
    }
}

class Annunciator implements AlarmListener
{
    public void alarm()
    {
        System.out.println("报警器发出警报！");
    }
}

class ExitGate  _____(2)_____
{
    public void alarm()
    {
        _____(3)_____;
    }

    public void open()
    {
        System.out.println("逃生门开启！");
    }

    public void close()
    {
        System.out.println("隔热门关闭！");
    }
}

class ThermoSensor
{
    private ArrayList alarmListeners = new ArrayList();

    public void register(AlarmListener al)
    {
        _____(4)_____;
    }

    public void trigger()
```

```
    {
        for(Object obj:alarmListeners)
        {
            _____(5)_____;
        }
    }
}

class Test
{
    public static void main(String args[])
    {
        AlarmListener light,annunciator,exitGate;
        light = new CautionLight();
        annunciator = new Annunciator();
        exitGate = new ExitGate();

        ThermoSensor sensor;
        _____(6)_____;

        sensor.register(light);
        sensor.register(annunciator);
        sensor.register(exitGate);
        _____(7)_____;  //触发警报
    }
}
```

参考答案：

(1) interface AlarmListener；(2) extends Gate implements AlarmListener；(3) super.action()；(4) alarmListeners.add(al)；(5) ((AlarmListener)obj).alarm()；(6) sensor = new ThermoSensor()；(7) sensor.trigger()。

4. 传输门是传输系统中的重要装置。传输门具有 Open（打开）、Closed（关闭）、Opening（正在打开）、StayOpen（保持打开）、Closing（正在关闭）五种状态。触发状态的转换事件有 click、complete 和 timeout 三种。事件与其相应的状态转换如图 1 所示。

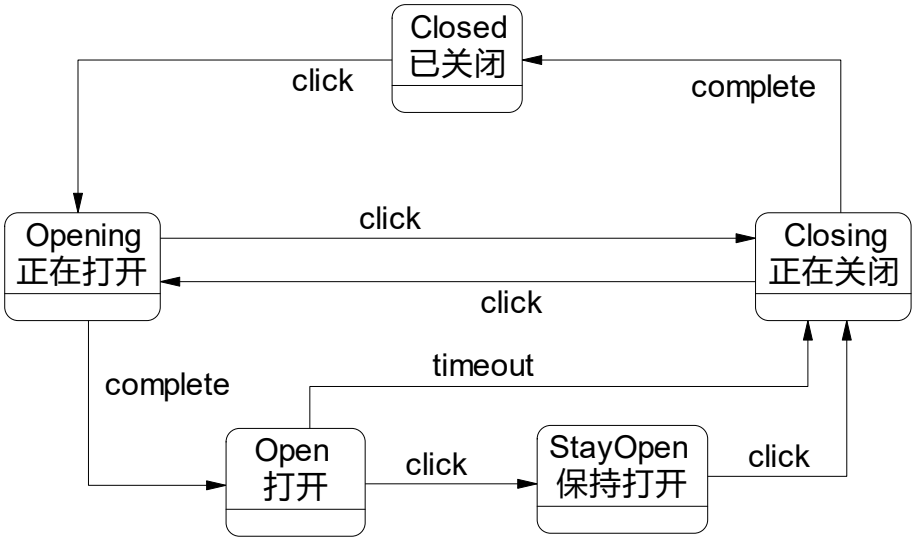


图 1 传输门响应事件与其状态转换图

下面的【Java 代码 1】与【Java 代码 2】分别用两种不同的设计思路对传输门进行状态模拟，请填补代码中的空缺。

【Java 代码 1】

```
public class Door {
    public static final int CLOSED = 1;
    public static final int OPENING = 2;
    public static final int OPEN = 3;
    public static final int CLOSING = 4;
    public static final int STAYOPEN = 5;
    private int state = CLOSED;
    //定义状态变量，用不同的整数表示不同状态

    private void setState(int state) {
        this.state = state;   //设置传输门当前状态
    }
    public void getState() {
        //此处代码省略，本方法输出状态字符串，
        //例如，当前状态为 CLOSED 时，输出字符串为“CLOSED”
    }
    public void click() {
        if(_____(1)_____)   setState(OPENING);
        else if(_____(2)_____)   setState(CLOSING);
        else if(_____(3)_____)   setState(STAYOPEN);
    }
    //发生 timeout 事件时进行状态转换
    public void timeout() {
        if(state == OPEN) setState(CLOSING);
    }
    public void complete() { //发生 complete 事件时进行状态转换
        if(state == OPENING)   setState(OPEN);
```

```
            else if(state == CLOSING)    setState(CLOSED);
        }
        public static void main(String[] args) {
            Door aDoor = new Door();
            aDoor.getState();    aDoor.click();    aDoor.getState();    aDoor.complete();
            aDoor.getState();    aDoor.click();    aDoor.getState();    aDoor.click();
            aDoor.getState();    return;
        }
}
```

【Java 代码 2】

```
public class Door {
        public final DoorState CLOSED = new DoorClosed(this);
        public final DoorState OPENING = new DoorOpening(this);
        public final DoorState OPEN = new DoorOpen(this);
        public final DoorState CLOSING = new DoorClosing(this);
        public final DoorState STAYOPEN = new DoorStayOpen(this);
        private DoorState state = CLOSED;

        //设置传输门当前状态
        public void setState(DoorState state) {
            this.state = state;
        }
        public void getState() {//根据当前状态输出对应的状态字符串
            System.out.println(state.getClass().getName());
        }
        public void click() {_____(4)_____;}   //发生 click 事件时进行状态转换
        public void timeout() {_____(5)____;}   //发生 timeout 事件时进行状态转换
        public void complete() {_____(6)____;}   //发生 complete 事件时进行状态转换
        public static void main(String[] args){
            Door aDoor = new Door();
            aDoor.getState();    aDoor.click();    aDoor.getState();    aDoor.complete();
            aDoor.getState();    aDoor.timeout();    aDoor.getState();    return;
        }
}
public abstract class DoorState {   //定义所有状态类的基类
        protected Door door;
        public DoorState(Door door) {
            this.door = door;
        }
        public abstract void click() {}
        public abstract void complete() {}
        public abstract void timeout() {}
}
```

```
class DoorClosed extends DoorState { //定义一个基本的 Closed 状态
    public DoorClosed (Door door) {super(door); }
    public void click() {_____(7)____;}
    //该类定义的其余代码省略
}
//其余代码省略
```

参考答案：

(1) state == CLOSED || state ==CLOSING；(2) state == OPENING || state == STAYOPEN；(3) state == OPEN ；(4) state.click() ；(5) state.timeout() ；(6) state.complete() ；(7) door.setState(door.OPENING)。

5. 某软件公司现欲开发一款飞机模拟系统，该系统主要模拟不同种类飞机的飞行特征与起飞特征。需要模拟的飞机种类及其特征如表 1 所示。

表 1

飞机种类	起飞特征	飞行特征
直升机(Helicopter)	垂直起飞(VerticalTakeOff)	亚音速飞行(SubSonicFly)
客机(AirPlane)	长距离起飞(LongDistanceTakeOff)	亚音速飞行(SubSonicFly)
歼击机(Fighter)	长距离起飞(LongDistanceTakeOff)	超音速飞行(SuperSonicFly)
鹞式战斗机(Harrier)	垂直起飞(VerticalTakeOff)	超音速飞行(SuperSonicFly)

为支持将来模拟更多种类的飞机，采用策略设计模式(Strategy)设计的类图如图 1 所示。

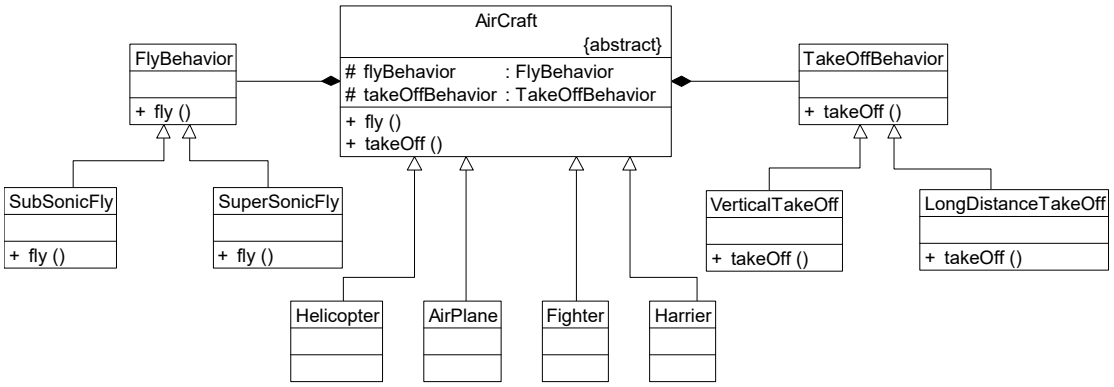


图 1 类图

图 1 中，AirCraft 为抽象类，描述了抽象的飞机，而类 Helicopter、AirPlane、Fighter 和 Harrier 分别描述具体的飞机种类，方法 fly()和 takeOff()分别表示不同飞机都具有飞行特征和起飞特征；类 FlyBehavior 与 TakeOffBehavior 为抽象类，分别用于表示抽象的飞行行为与起飞行为；类 SubSonicFly 与 SuperSonicFly 分别描述亚音速飞行和超音速飞行的行为；类 VerticalTakeOff 与 LongDistanceTakeOff 分别描述垂直起飞与长距离起飞的行为。

【Java 代码】

```
interface FlyBehavior{
    public void fly();
}

class SubSonicFly implements FlyBehavior{
    public void fly() {System.out.println("亚音速飞行！ ");}
```

```
}

class SuperSonicFly implements FlyBehavior{
    public void fly() {System.out.println("超音速飞行！");}
}

interface TakeOffBehavior{
    public void takeOff();
}

class VerticalTakeOff implements TakeOffBehavior{
    public void takeOff() {System.out.println("垂直起飞！");}
}

class LongDistanceTakeOff implements TakeOffBehavior{
    public void takeOff() {System.out.println("长距离起飞！");}
}

abstract class AirCraft{
    protected ____(1)____;
    protected ____(2)____;
    public void fly() { ____(3)____; }
    public void takeOff() { ____(4)____; }
}

class Helicopter ____(5)____ AirCraft {
    public Helicopter() {
        flyBehavior = new ____(6)____;
        takeOffBehavior = new ____(7)____;
    }
}
//其他代码省略
```

参考答案：

(1) FlyBehavior flyBehavior；(2) TakeOffBehavior takeOffBehavior；(3) flyBehavior.fly()；(4) takeOffBehavior.takeOff()；(5) extends；(6) SubSonicFly()；(7) VerticalTakeOff()。

6. 已知某类库开发商提供了一套类库，类库中定义了 Application 类和 Document 类，它们之间的关系如图 1 所示，其中，Application 类表示应用程序自身，而 Document 类则表示应用程序打开的文档。Application 类负责打开一个已有的以外部形式存储的文档，如一个文件，一旦从该文件中读出信息后，它就由一个 Document 对象表示。

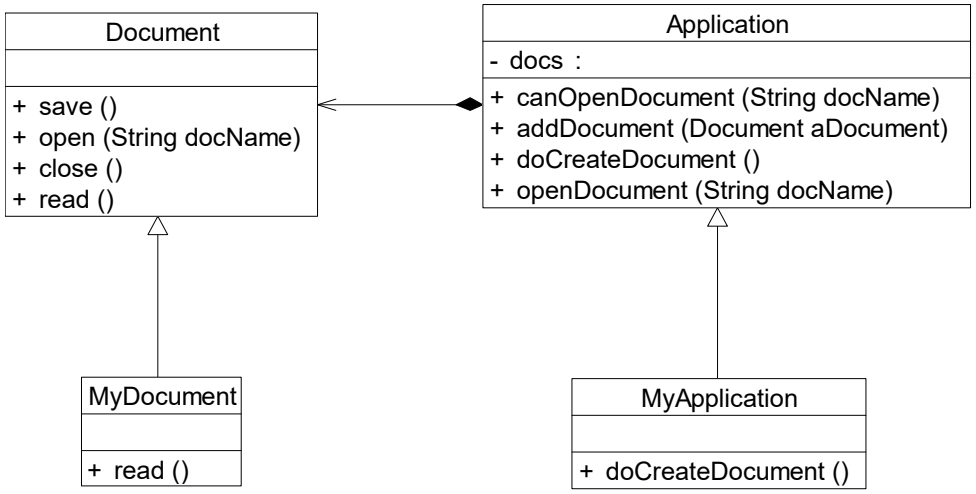


图 1 类图

当开发一个具体的应用程序时，开发者需要分别创建自己的 Application 和 Document 子类，例如图 1 中的类 MyApplication 和类 MyDocument，并分别实现 Application 和 Document 类中的某些方法。

已知 Application 类中的 openDocument 方法采用了模板方法设计模式，该方法定义了打开文档的每一个步骤，如下所示：

(1) 首先检查文档是否能够被打开，若不能打开，则给出出错信息并返回；

(2) 创建文档对象；

(3) 通过文档对象打开文档；

(4) 通过文档对象读取文档信息；

(5) 将文档对象加入到 Application 的文档对象集合中。

【Java 代码】

```
abstract class Document {
    public void save() { /*  存储文档数据，此处代码省略  */}
    public void open(String docName) { /*打开文档，此处代码省略  */}
    public void close() { /*  关闭文档，此处代码省略  */}
    public abstract void read(String docName);
}

abstract class Application {
    private Vector<____(1)____> docs; /*文档对象集合*/
    public boolean canOpenDocument(String docName){
        /*判断是否可以打开指定文档，返回真值时表示可以打开，
          返回假值表示不可打开，此处代码省略*/
    }
    public void addDocument(Document aDocument){
        /*将文档对象添加到文档对象集合中*/
        docs.add(____(2)____);
    }
    public abstract Document doCreateDocument(); /*创建一个文档对象*/
    public void openDocument(String docName) { /*打开文档*/
        if(_____(3)_____) {
```

```
                System.out.println("文档无法打开！");
                return;
            }
            ____(4)____adoc = _____(5)_____;
            _____(6)_____;
            _____(7)_____;
            _____(8)_____;
        }
}
```

参考答案：

(1) Document； (2) aDocument； (3) !canOpenDocument(docName)； (4) Document； (5) doCreateDocument(); (6) adoc.open(docName); (7) adoc.read(docName); (8) addDocument(adoc)。

二、 综合题：

(一) UML 和面向对象设计原则

1. 一种某唱片播放器不仅可以播放唱片，而且可以连接计算机并把计算机中的歌曲刻录到唱片上（同步歌曲）。连接计算机的过程中还可自动完成充电。

关于唱片，还有如下描述信息：

（1） 每首歌曲的描述信息包括：歌曲的名字，谱写这首歌曲的艺术家以及演奏这首歌曲的艺术家。只有两首歌曲的这三部分信息完全相同时，才认为它们是同一首歌曲。艺术家可能是一名歌手或一支由 2 人或 2 名以上的歌手组成的乐队。一名歌手可以不属于任何乐队，也可以属于一个或多个乐队。

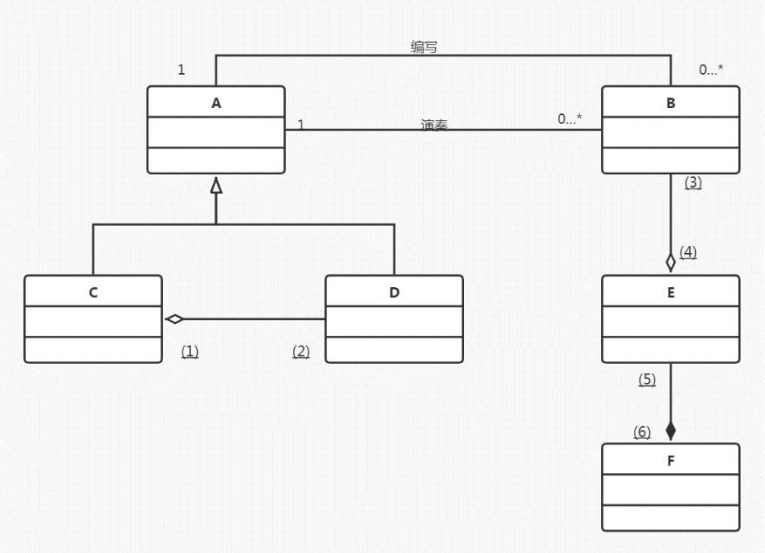
（2） 每章唱片有多条音轨构成；一条音轨中只包含一首歌曲或为空，一首歌曲可分布在多条音轨上，通一首歌曲在一张唱片中最多只能出现一次。

（3） 每条音轨都有一个开始位置和持续时间。一张唱片中音轨的次序是非常重要的，因此对于任意一条音轨，播放器需要准确地指导它的下一条音轨和上一条音轨是什么（如果存在的话）。

根据上述描述，采用面向对象方法对其进行分析或设计，得到了如表 1-3 所示的类列表和如图 1-16 所示的初始类图。

表 1-3 类列表

类名	说明	类名	说明
Artist	艺术家	Musician	歌手
Song	歌曲	Track	音轨
Band	乐队	Album	唱片



【问题 1】根据题干中的描述，使用表 1-3 给出的类的名称，给图 1-16 中 A~F 所对应的类。

【问题 2】根据题干中的描述，给出图 1-16 中（1）~（6）处的多重性。

【问题 3】图 1-16 中缺少了一条关联，请指出这条关联两端所对应的类以及每一端的多重性。

参考答案：

[问题 1] A: Artist　　B: Song　　C: Band　　D: Musician

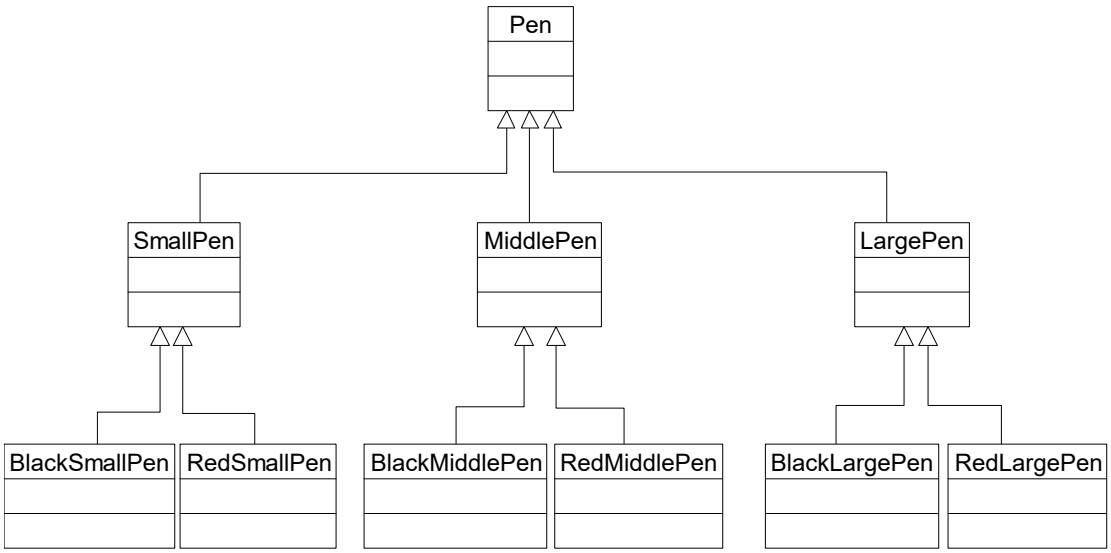
E: Track　　F: Album

[问题 2] (1) 0..*; (2) 2..*; (3) 0..1; (4) 1..*; (5) 1..*; (6) 1..1。

[问题 3]

类	多重度
Track	0..2
Track	0..2

2.　在某绘图软件中提供了多种大小不同的画笔(Pen)，并且可以给画笔指定不同颜色，某设计人员针对画笔的结构设计了如下类图：

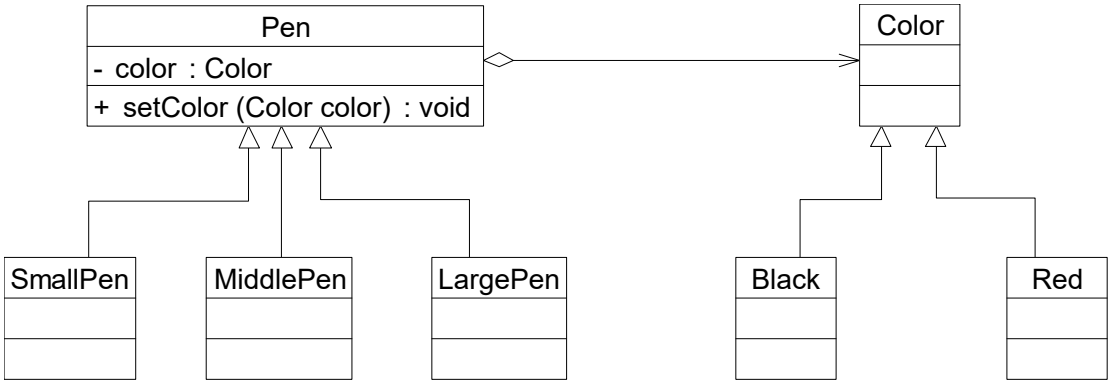


通过仔细分析，设计人员发现该类图存在非常严重的问题，如果需要增加一种新的大小的笔或者增加一种新的颜色，都需要增加很多子类，如增加一种绿色，则对应每一种大小的笔都需要增加一支绿色笔，系统中类的个数急剧增加。

试根据依赖倒转原则和合成复用原则对该设计方案进行重构，使得增加新的大小的笔和增加新的颜色都较为方便。

参考答案：

本练习可以通过依赖倒转原则和合成复用原则进行重构，重构方案如下所示：



在本重构方案中，将笔的大小和颜色设计为两个继承结构，两者可以独立变化，根据依赖倒转原则，建立一个抽象的关联关系，将颜色对象注入到画笔中；再根据合成复用原则，画笔在保持原有方法的同时还可以调用颜色类的方法，保持原有性质不变。如果需要增加一种新的画笔或增加一种新的颜色，只需对应增加一个具体类即可，且客户端可以针对高层类 Pen 和 Color 编程，在运行时再注入具体的子类对象，系统具有良好的可扩展性，满足开闭原则。（注：本重构方法即为桥接模式，在第 4 章将对该模式进行进一步讲解并提供实例代码来实现该模式。）

3. 结合面向对象设计原则分析：正方形是否是长方形的子类？

参考答案：

可使用里氏代换原则来分析正方形类是否是长方形类的子类，具体分析过程如下：

```
class Rectangle        //长方形
{
```

```
    private double width;
    private double height;

    public Rectangle(double width,double height)
    {
        this.width=width;
        this.height=height;
    }
    public double getHeight()
    {
        return height;
    }
    public void setHeight(double height)
    {
        this.height = height;
    }
    public double getWidth()
    {
        return width;
    }
    public void setWidth(double width)
    {
        this.width = width;
    }
}

class Square extends Rectangle      //正方形
{
    public Square(double size)
    {
        super(size,size);
    }

    public void setHeight(double height)
    {
        super.setHeight(height);
        super.setWidth(height);
    }
    public void setWidth(double width)
    {
        super.setHeight(width);
        super.setWidth(width);
    }
}
```

```
class Client
{
    public static void main(String args[])
    {
        Rectangle r;
        r = new Square(0.0);
        r.setWidth(5.0);
        r.setWidth(10.00);
        double area = calculateArea(r);
        if(50.00==area)
        {
            System.out.println("这是长方形或长方形的子类！");
        }
        else
        {
            System.out.println("这不是长方形！");
        }
    }

    public static double calculateArea(Rectangle r)
    {
        return r.getHeight() * r.getWidth();
    }
}
```

由代码输出可以得知，我们在客户端代码中使用长方形类来定义正方形对象，将输出“这不是长方形！”，即将正方形作为长方形的子类，在使用正方形替换长方形之后正方形已经不再是长方形，接受基类对象的地方接受子类对象时出现问题，违反了里氏代换原则，因此从面向对象的角度分析，正方形不是长方形的子类，它们都可以作为四边形类的子类。关于该问题的进一步讨论，大家可以参考其他相关资料，如 Bertrand Meyer 的基于契约设计(Design By Contract)，在长方形的契约(Contract)中，长方形的长和宽是可以独立变化的，但是正方形破坏了该契约。

4. 某游戏公司现欲开发一款面向儿童的模拟游戏，该游戏主要模拟现实世界中各种鸭子的发声特征、飞行特征和外观特征。游戏需要模拟的鸭子种类及其特征如下表所示：

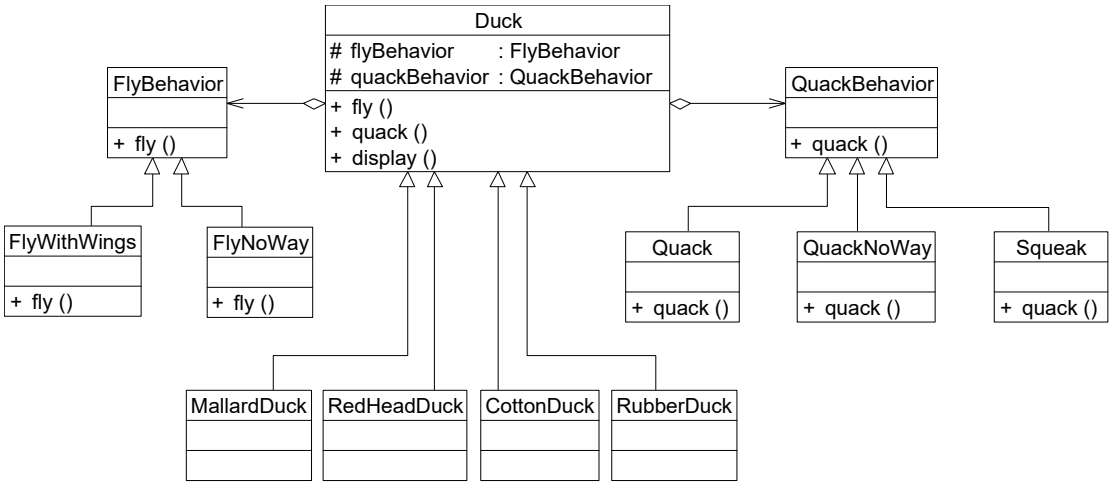
鸭子种类	发声特征	飞行特征	外观特征
灰鸭 (MallardDuck)	发出“嘎嘎”声 (Quack)	用翅膀飞行 (FlyWithWings)	灰色羽毛
红头鸭 (RedHeadDuck)	发出“嘎嘎”声 (Quack)	用翅膀飞行 (FlyWithWings)	灰色羽毛、头部红色
棉花鸭 (CottonDuck)	不发声 (QuackNoWay)	不能飞行 (FlyNoWay)	白色
橡皮鸭	发出橡皮与空气摩擦	不能飞行	黑白橡皮颜色

(RubberDuck)	的声音 (Squeak)	(FlyNoWay)	

为支持将来能够模拟更多种类鸭子的特征，选择一种合适的设计模式设计该模拟游戏，提供相应的解决方案。

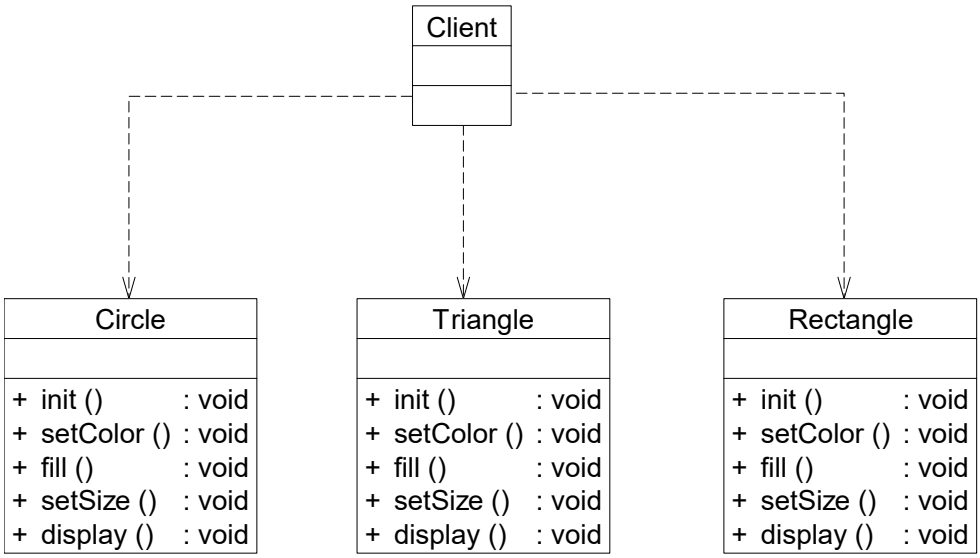
参考答案：

本题可使用策略模式，参考类图如下所示：



在类图中，Duck 为抽象类，描述了抽象的鸭子，而类 MallardDuck、RedHeadDuck、CottonDuck 和 RubberDuck 分别描述各种具体的鸭子，方法 fly()、quack()和 display()分别表示不同种类的鸭子都具有飞行特征、发声特征和外观特征；类 FlyBehavior 和 QuackBehavior 为抽象类，分别用于表示抽象的飞行行为和发声行为；类 FlyWithWings 和 FlyNoWay 分别描述用翅膀飞行的行为和不能飞行的行为；类 Quack、QuackNoWay 和 Squeak 分别描述发出“嘎嘎”声的行为、不发声的行为和发出橡皮与空气摩擦声的行为。

5. 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显示图形，图形库设计人员设计的初始类图如下所示：



在该图形库中，每个图形类（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。

客户类(Client)在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。

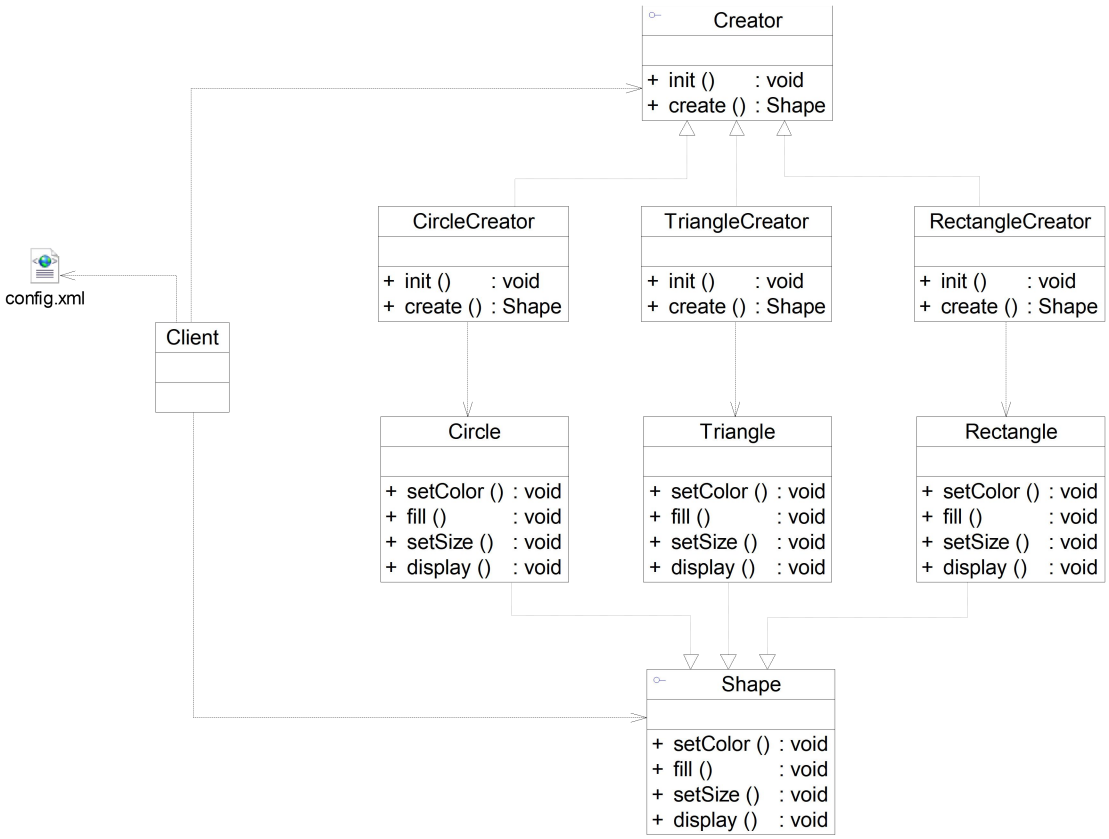
参考答案：

本练习可以通过单一职责原则和依赖倒转原则进行重构，具体过程可分为如下两步：

(1) 由于图形对象的创建过程较为复杂，因此可以将创建过程封装在专门的类中（这种专门用于创建对象的类称为工厂类），将对象的创建和使用分离，符合单一职责原则；

(2) 引入抽象的图形类 Shape，并对应提供一个抽象的创建类，将具体图形类作为 Shape 的子类，而具体的图形创建类作为抽象创建类的子类，根据依赖倒转原则，客户端针对抽象图形类和抽象图形创建类编程，而将具体的图形创建类类名存储在配置文件中。

重构之后的类图如下所示：



通过上述重构，在抽象类 Creator 中声明了创建图形对象的方法 create()，在其子类中实现了该方法，用于创建具体的图形对象。客户端针对抽象 Creator 编程，而将其具体子类类名存储在配置文件 config.xml 中，如果需要更换图形，只需在配置文件中更改 Creator 的具

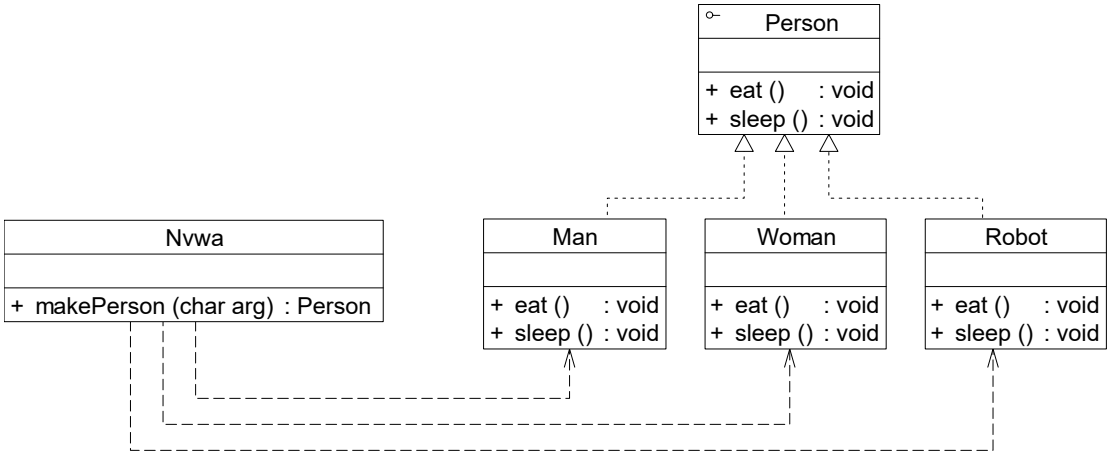
体子类类名即可；如果需要增加图形，则对应增加一个新的 Creator 子类用于创建新增图形对象，再修改配置文件，在配置文件中存储新的图形创建类类名。更换和增加图形都无须修改源代码，完全符合开闭原则。（注：本重构方法即为工厂方法模式，在第 3 章将对该模式进行进一步讲解并提供实例代码来实现该模式。）

(二)　创建型设计模式

6.　使用简单工厂模式模拟女娲(Nvwa)造人(Person)，如果传入参数“M”，则返回一个 Man 对象，如果传入参数“W”，则返回一个 Woman 对象，用 Java 语言实现该场景。现需要增加一个新的 Robot 类，如果传入参数“R”，则返回一个 Robot 对象，对代码进行修改并注意“女娲”的变化。（简单工厂）

参考答案：

简单工厂模式。参考类图如下所示：



分析：在本实例中，Nvwa 类充当工厂类，其中定义了工厂方法 makePerson()，Person 类充当抽象产品类，Man、Woman 和 Robot 充当具体产品类。工厂方法 makePerson()的代码如下所示：

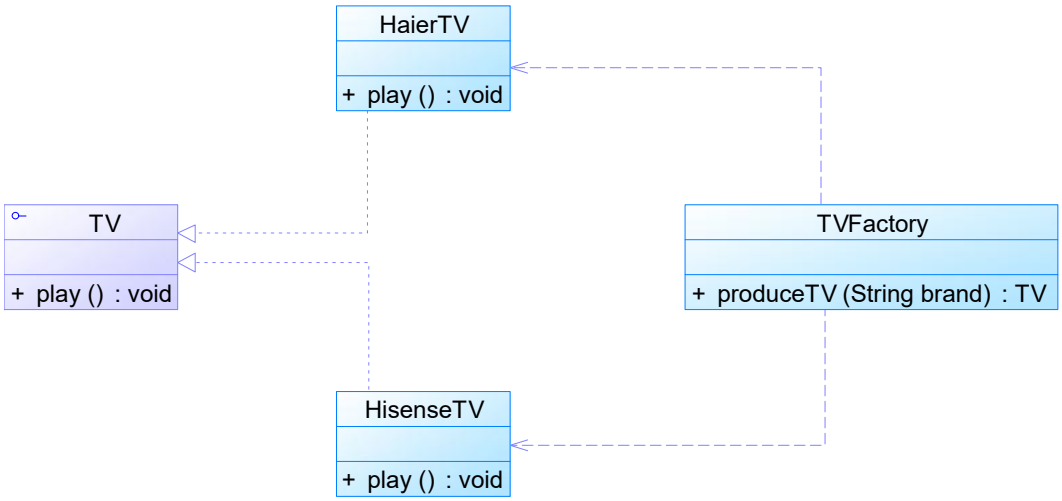
```
public static Person makePerson(char arg)
{
    Person person = null;
    switch(arg)
    {
        case 'M':
        person = new Man();    break;
        case 'W':
        person = new Woman();    break;
        case 'R':
        person = new Robot();    break;
    }
    return person;
```

```
}
```

如果需要增加一个新的具体产品，则必须修改 makePerson()方法中的判断语句，需增加一个新的 case 语句，违背了开闭原则。

7. 某电视机厂专为各知名电视机品牌代工生产各类电视机，当需要海尔牌电视机时只需要在调用该工厂的工厂方法时传入参数“Haier”，需要海信电视机时只需要传入参数“Hisense”，工厂可以根据传入的不同参数返回不同品牌的电视机。现使用简单工厂模式来模拟该电视机工厂的生产过程。（简单工厂，课堂例题）

参考答案：

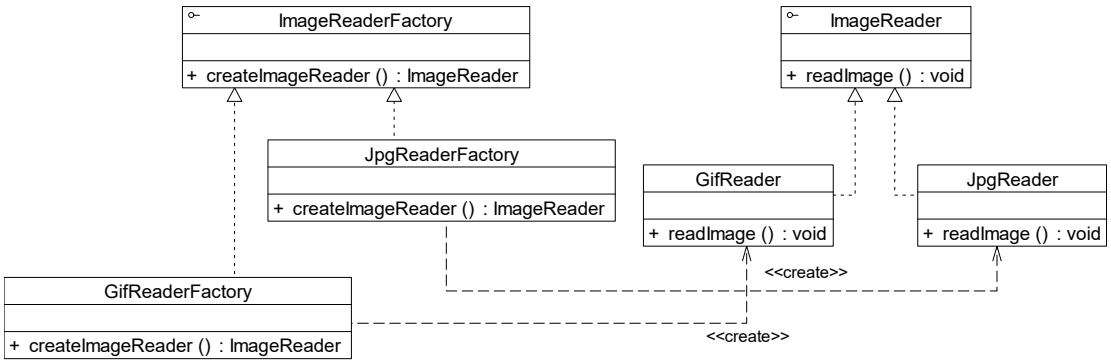


代码见课本、学习通课程资料。

8. 现需要设计一个程序来读取多种不同类型的图片格式，针对每一种图片格式都设计一个图片读取器(ImageReader)，如 GIF 图片读取器(GifReader)用于读取 GIF 格式的图片、JPG 图片读取器(JpgReader)用于读取 JPG 格式的图片。图片读取器对象通过图片读取器工厂 ImageReaderFactory 来创建，ImageReaderFactory 是一个抽象类，用于定义创建图片读取器的工厂方法，其子类 GifReaderFactory 和 JpgReaderFactory 用于创建具体的图片读取器对象。使用工厂方法模式实现该程序的设计。

参考答案：

工厂方法模式。参考类图如下所示：

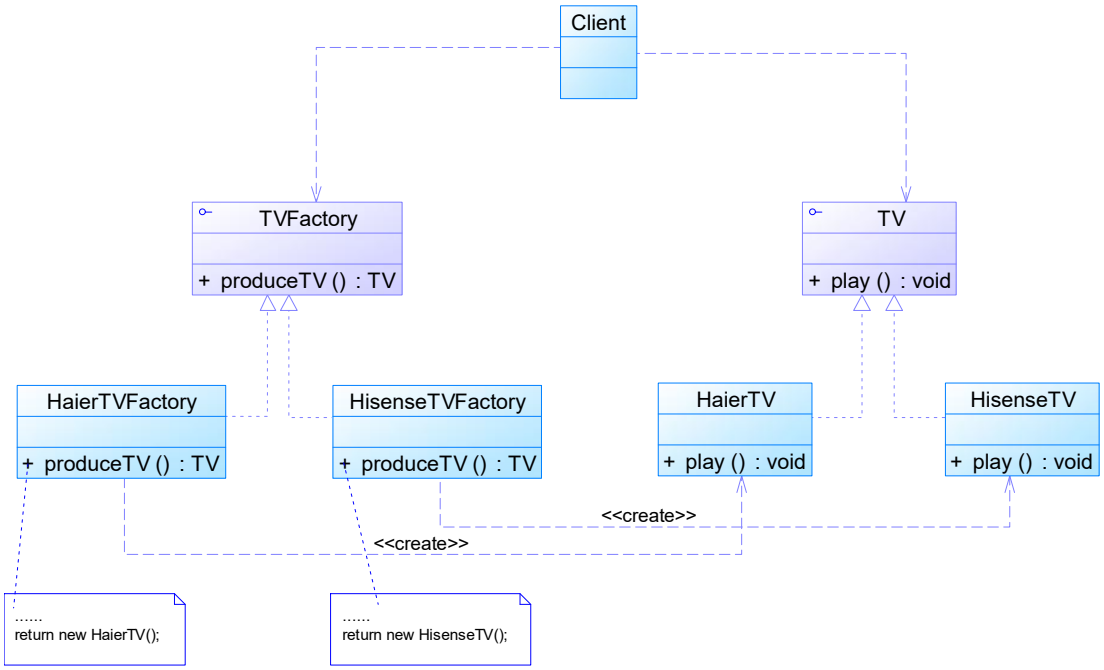


分析：在本实例中，ImageReaderFactory 充当抽象工厂，GifReaderFactory 和 JpgReaderFactory 充当具体工厂，ImageReader 充当抽象产品，GifReader 和 JpgReader 充当具体产品。

代码略。

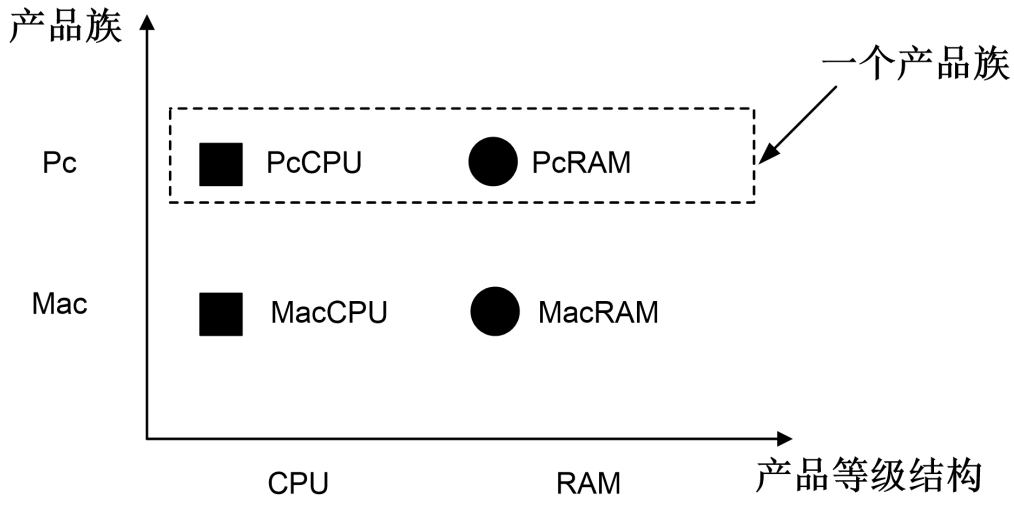
9. 为每种品牌的电视机提供一个子工厂，海尔工厂专门负责生产海尔电视机，海信工厂专门负责生产海信电视机，如果需要生产 TCL 电视机或创维电视机，只需要对应增加一个新的 TCL 工厂或创维工厂即可，原有的工厂无须做任何修改，使得整个系统具有更加的灵活性和可扩展性。（工厂方法，课堂例题）

参考答案：



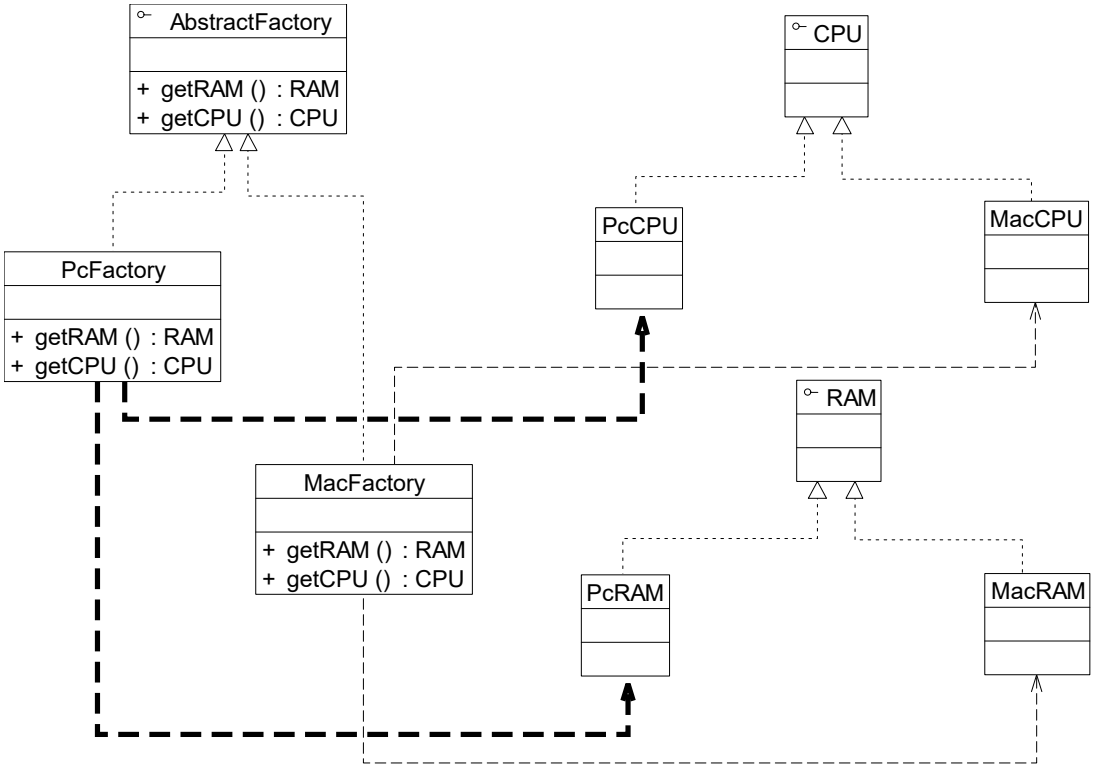
代码见教材或学习通课程资料。

10. 计算机包含内存(RAM)、CPU 等硬件设备，根据下面的“产品等级结构-产品族”示意图，使用抽象工厂模式实现计算机设备创建过程并绘制相应的类图。



参考代码：

抽象工厂模式。参考类图如下所示：

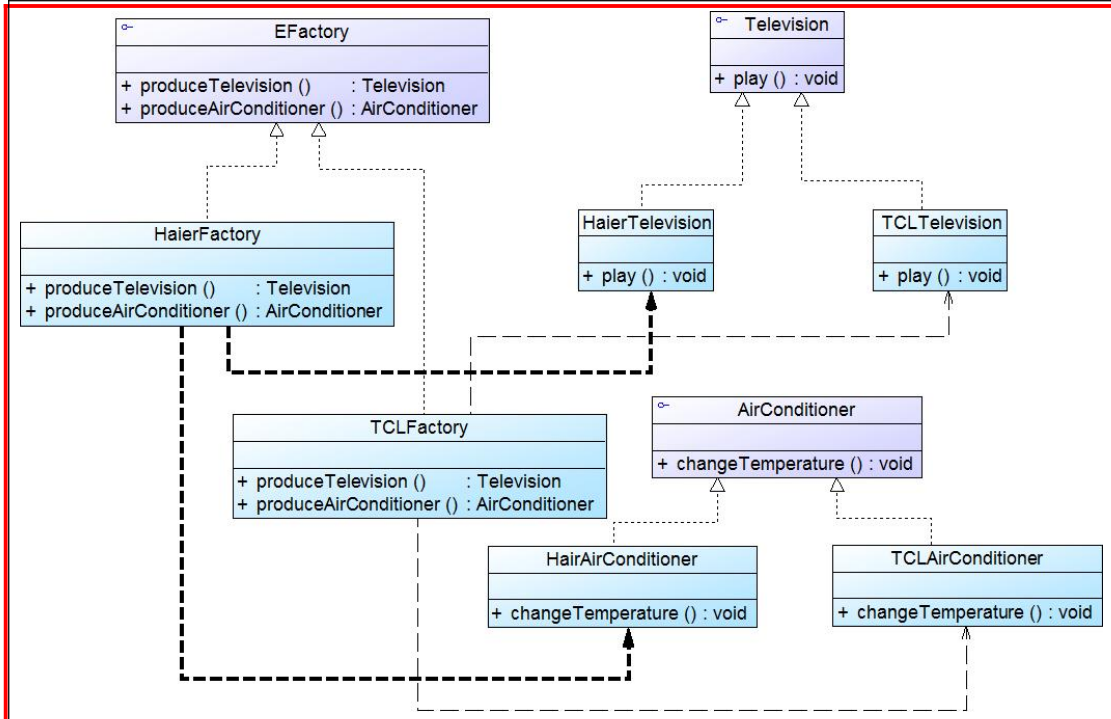


分析：在本实例中，AbstractFactory 充当抽象工厂，PcFactory 和 MacFactory 充当具体工厂，CPU 和 RAM 充当抽象产品，PcCPU、MacCPU、PcRAM 和 MacRAM 充当具体产品。CPU、PcCPU 和 MacCPU 构成一个产品等级结构，RAM、PcRAM 和 MacRAM 构成一个产品等级结构，PcCPU 和 PcRAM 构成一个产品族，MacCPU 和 MacRAM 构成一个产品族。

代码略。

11. 一个电器工厂可以产生多种类型的电器，如海尔工厂可以生产海尔电视机、海尔空调等，TCL 工厂可以生产 TCL 电视机、TCL 空调等，相同品牌的电器构成一个产品族，而相同类型的电器构成了一个产品等级结构，现使用抽象工厂模式模拟该场景。（抽象工厂方法模式，课堂例题）

参考答案：

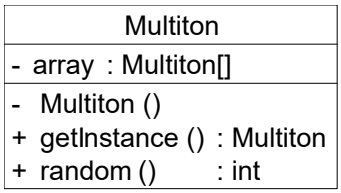


代码见教材或学习通课程资料。

12. 使用单例模式的思想实现多例模式，确保系统中某个类的对象只能存在有限个，如两个或三个，设计并编写代码实现一个多例类。

参考答案：

单例模式。参考类图如下所示：



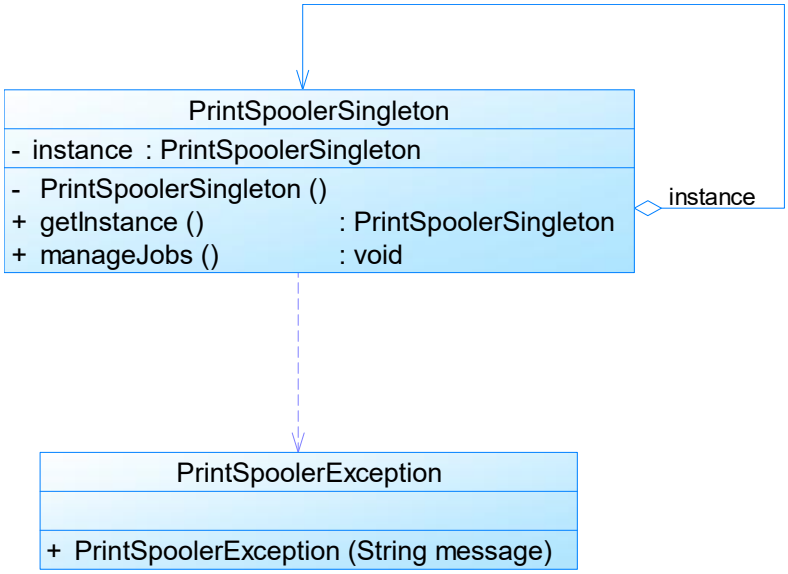
分析：多例模式(Multiton Pattern)是单例模式的一种扩展形式，多例类可以有多个实例，而且必须自行创建和管理实例，并向外界提供自己的实例，可以通过静态集合对象来存储这些实例。多例类 Multiton 的代码如下所示：

```
import java.util.*;

public class Multiton
{
    //定义一个数组用于存储四个实例
    private static Multiton[] array = {new Multiton(), new Multiton(), new Multiton(), new Multiton()};
    //私有构造函数
    private Multiton()
    {
    }
    //静态工厂方法，随机返回数组中的一个实例
```

```
    public static Multiton getInstance()
    {
        return array[random()];
    }
    //随机生成一个整数作为数组下标
    public static int random()
    {
        Date d = new Date();
        Random random = new Random();
        int value = Math.abs(random.nextInt());
        value = value % 4;
        return value;
    }
    public static void main(String args[])
    {
        Multiton m1,m2,m3,m4;
        m1 = Multiton.getInstance();
        m2 = Multiton.getInstance();
        m3 = Multiton.getInstance();
        m4 = Multiton.getInstance();

        System.out.println(m1==m2);
        System.out.println(m1==m3);
        System.out.println(m1==m4);
    }
}
```

13. 在操作系统中，打印池(Print Spooler)是一个用于管理打印任务的应用程序，通过打印池用户可以删除、中止或者改变打印任务的优先级，在一个系统中只允许运行一个打印池对象，如果重复创建打印池则抛出异常。现使用单例模式来模拟实现打印池的设计。（单例模式，课堂例题）

参考答案：



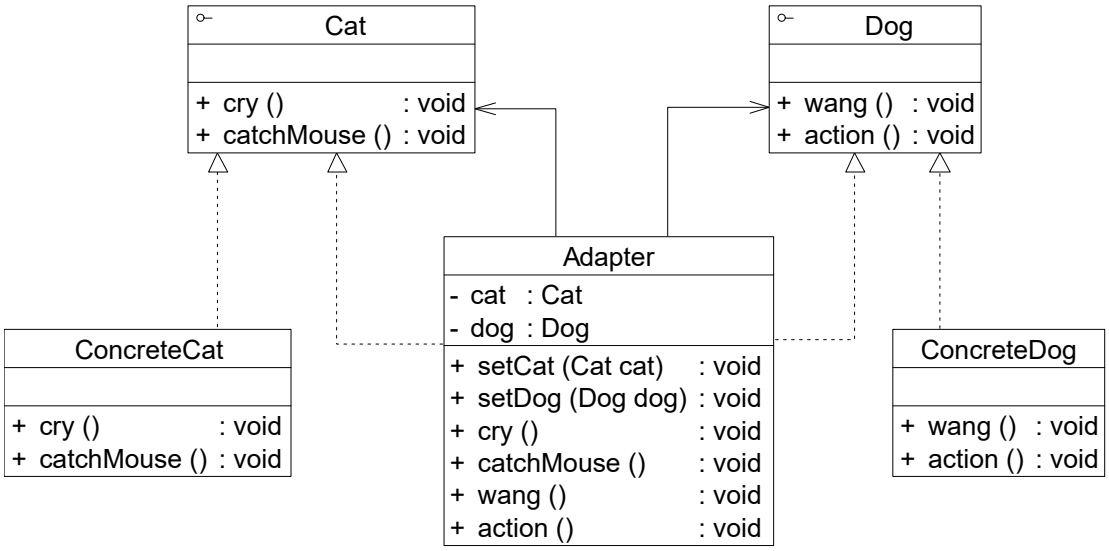
代码见教材或学习通课程资料。

(三) 结构型设计模式

14. 实现一个双向适配器实例，使得猫(Cat)可以学狗(Dog)叫，狗可以学猫抓老鼠。绘制相应类图并使用代码编程模拟。

参考答案：

适配器模式。参考类图如下所示：



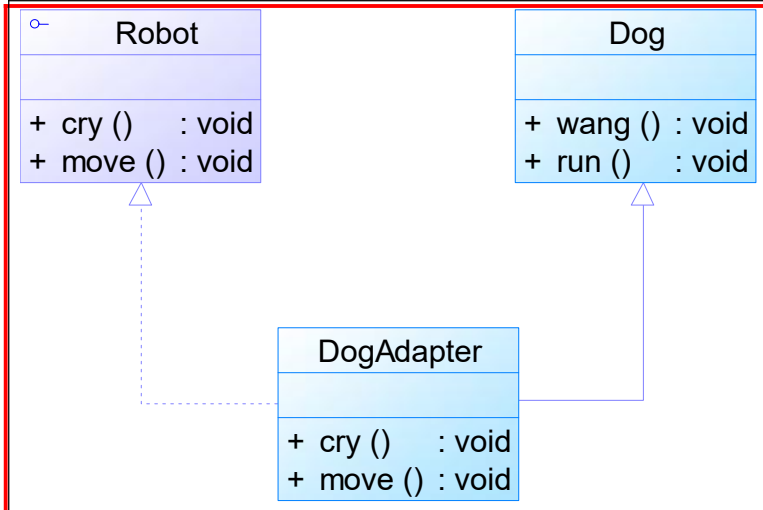
分析：在本实例中，Adapter 充当适配器，Cat 和 Dog 既充当抽象目标，又充当抽象适配者。如果客户端针对 Cat 编程，则 Cat 充当抽象目标，Dog 充当抽象适配者，ConcreteDog 充当具体适配者；如果客户端针对 Dog 编程，则 Dog 充当抽象目标，Cat 充当抽象适配者，ConcreteCat 充当具体适配者。本实例使用对象适配器，Adapter 类的代码如下所示：

```
class Adapter implements Cat, Dog
```

```
{
        private Cat cat;
        private Dog dog;
        public void setCat(Cat cat)
        {
          this.cat = cat;
        }
        public void setDog(Dog dog)
        {
          this.dog = dog;
        }
        public void cry()       //猫学狗叫
        {
          dog.wang();
        }
        public void catchMouse()
        {
          cat.catchMouse();
        }
        public void wang()
        {
          dog.wang();
        }
        public void action()     //狗学猫抓老鼠
        {
          cat.catchMouse();
        }
}
```

15. 现需要设计一个可以模拟各种动物行为的机器人，在机器人中定义了一系列方法，如机器人叫喊方法 cry()、机器人移动方法 move()等。如果希望在不修改已有代码的基础上使得机器人能够像狗一样叫，像狗一样跑，使用适配器模式进行系统设计。（适配器模式，课堂例题）

参考答案：

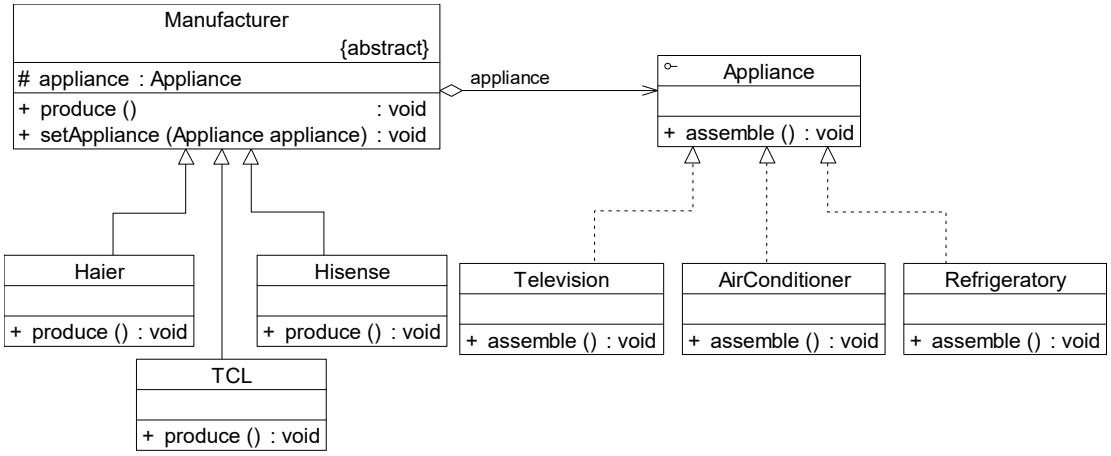


代码见教材或学习通课程资料。

16. 海尔(Haier)、TCL、海信(Hisense)都是家电制造商，它们都生产电视机(Television)、空调(Air Conditioner)、冰箱(Refrigeratory)。现需要设计一个系统，描述这些家电制造商以及它们所制造的电器，要求绘制类图并用代码模拟实现。

参考答案：

桥接模式。参考类图如下所示：



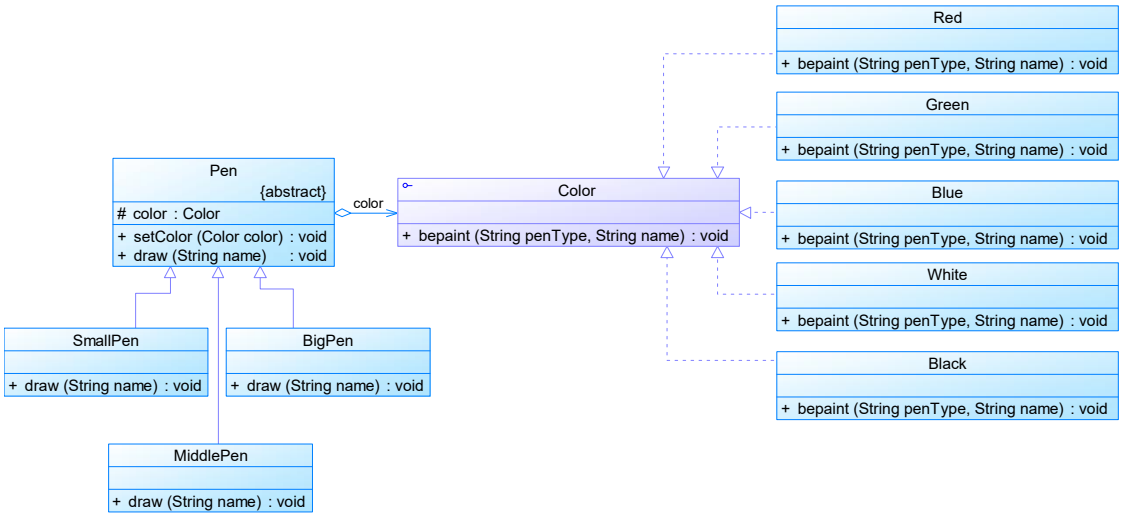

```
abstract class Manufacturer
{
    protected Appliance appliance;
    public abstract void produce();
    public void setAppliance(Appliance appliance)
    {
        this.appliance = appliance;
    }
}
```

```
class Haier extends Manufacturer
{
    public void produce()
    {
        System.out.println("生产海尔电器！");
        appliance.assemble();    //调用实现类的业务方法
    }
}
```

分析：在本实例中，Manufacturer 充当抽象类角色，Haier、TCL 和 Hisense 充当扩充抽象类角色，Appliance 充当抽象实现类，Television、AirConditioner 和 Refrigeratory 充当具体实现类。本实例部分代码如下所示：

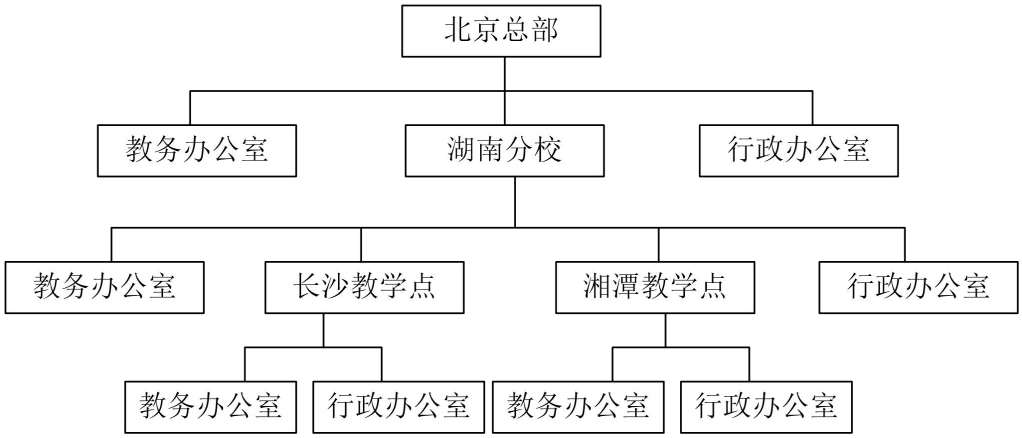
17. 现需要提供大中小 3 种型号的画笔，能够绘制 5 种不同颜色，如果使用蜡笔，我们需要准备 3*5=15 支蜡笔，也就是说必须准备 15 个具体的蜡笔类。而如果使用毛笔的话，只需要 3 种型号的毛笔，外加 5 个颜料盒，用 3+5=8 个类就可以实现 15 支蜡笔的功能。使用桥接模式来模拟毛笔的使用过程。（桥接模式，课堂例题）

参考答案：



代码见教材或学习通课程资料。

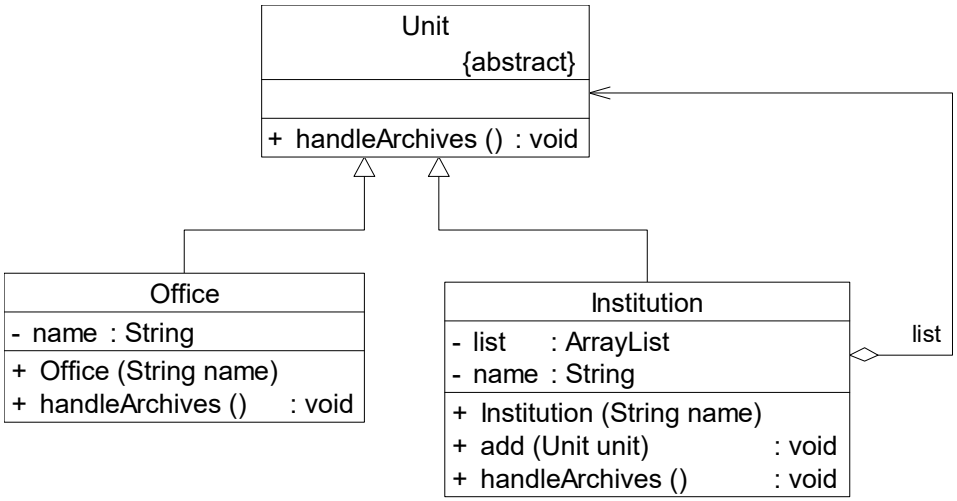
18. 某教育机构组织结构如下图所示：



在该教育机构的 OA 系统中可以给各级办公室下发公文，现采用组合模式设计该机构的组织结构，绘制相应的类图并编程模拟实现，在客户端代码中模拟下发公文。

参考答案：

组合模式。参考类图如下所示：



分析：本实例使用了安全组合模式，Unit 充当抽象构件角色，Office 充当叶子构件角色，Institution 充当容器构件角色。本实例代码如下所示：

```
abstract class Unit
{
    public abstract void handleArchives();
}

class Office extends Unit
{
    private String name;
    public Office(String name)
    {
        this.name = name;
    }
    public void handleArchives()
    {
        System.out.println(this.name + "处理公文！ ");
```

```
    }
}

class Institution extends Unit
{
    private ArrayList list = new ArrayList();
    private String name;
    public Institution(String name)
    {
        this.name = name;
    }
    public void add(Unit unit)
    {
        list.add(unit);
    }
    public void handleArchives()
    {
        System.out.println(this.name + "接收并下发公文：");
        for(Object obj : list)
        {
            ((Unit)obj).handleArchives();
        }
    }
}
```

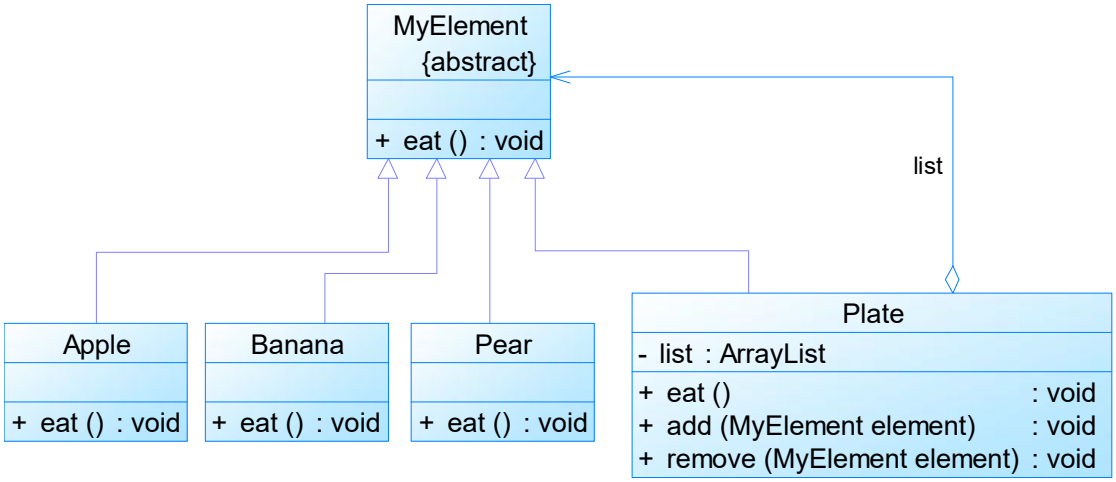
在客户类中创建树形结构，代码如下所示：

```
class Client
{
    public static void main(String args[])
    {
        Institution bjHeadquarters,hnSubSchool,csTeachingPost,xtTeachingPost;
        Unit tOffice1,tOffice2,tOffice3,tOffice4,aOffice1,aOffice2,aOffice3,aOffice4;
        bjHeadquarters = new Institution("北京总部");
        hnSubSchool = new Institution("湖南分校");
        csTeachingPost = new Institution("长沙教学点");
        xtTeachingPost = new Institution("湘潭教学点");
        tOffice1 = new Office("北京教务办公室");
        tOffice2 = new Office("湖南教务办公室");
        tOffice3 = new Office("长沙教务办公室");
        tOffice4 = new Office("湘潭教务办公室");
        aOffice1 = new Office("北京行政办公室");
        aOffice2 = new Office("湖南行政办公室");
        aOffice3 = new Office("长沙行政办公室");
        aOffice4 = new Office("湘潭行政办公室");
        csTeachingPost.add(tOffice3);
```

```
            csTeachingPost.add(aOffice3);
            xtTeachingPost.add(tOffice4);
            xtTeachingPost.add(aOffice4);
            hnSubSchool.add(csTeachingPost);
            hnSubSchool.add(xtTeachingPost);
            hnSubSchool.add(tOffice2);
            hnSubSchool.add(aOffice2);
            bjHeadquarters.add(hnSubSchool);
            bjHeadquarters.add(tOffice1);
            bjHeadquarters.add(aOffice1);
            bjHeadquarters.handleArchives();
        }
    }
```

19. 在水果盘(Plate)中有一些水果，如苹果(Apple)、香蕉(Banana)、梨子(Pear)，当然大水果盘中还可以有小水果盘，现需要对盘中的水果进行遍历（吃），当然如果对一个水果盘执行“吃”方法，实际上就是吃其中的水果。使用组合模式模拟该场景。（组合模式，课堂例题）

参考答案：

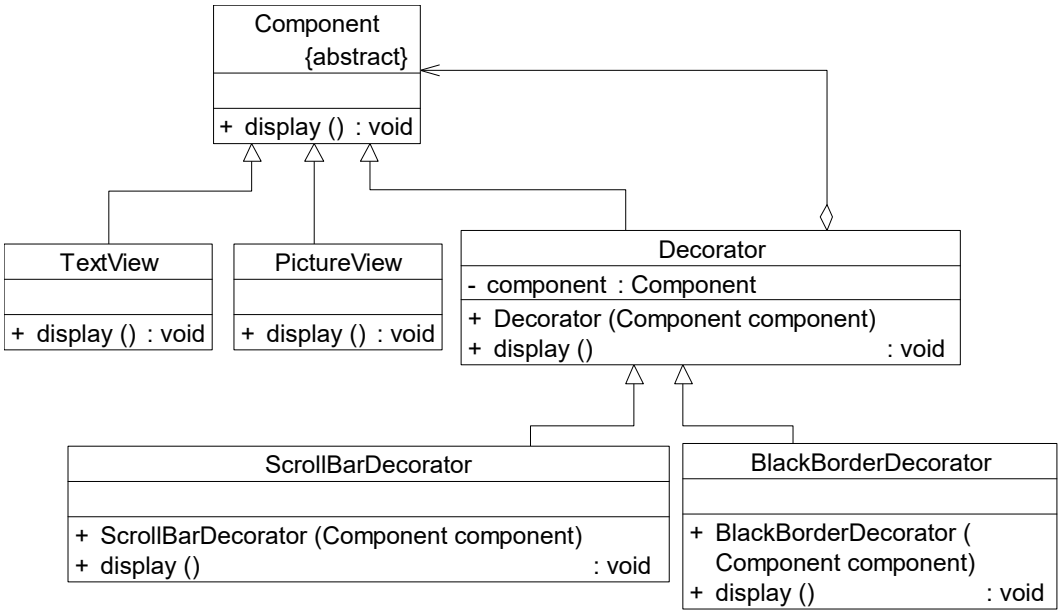


代码见教材或学习通课程资料。

20. 某系统中的文本显示组件类(TextView)和图片显示组件类(PictureView)都继承了组件类(Component)，分别用于显示文本内容和图片内容，现需要构造带有滚动条、或者带有黑色边框、或者既有滚动条又有黑色边框的文本显示组件和图片显示组件，为了减少类的个数可使用装饰模式进行设计，绘制类图并编程模拟实现。

参考答案：

装饰模式。参考类图如下所示：

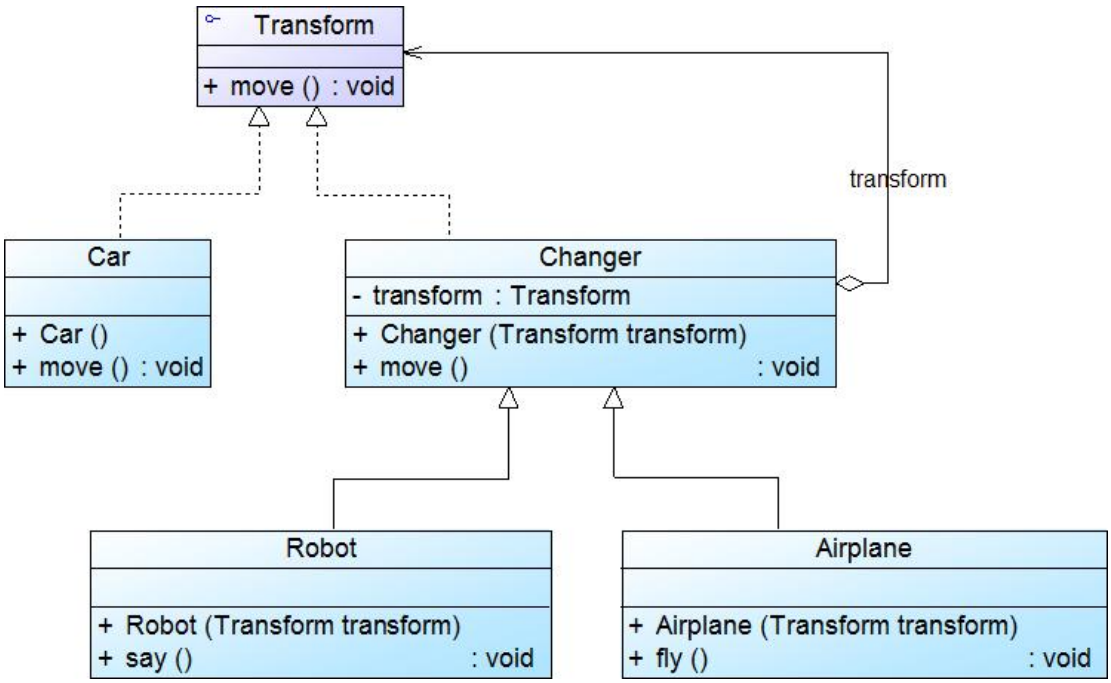


分析：本实例使用了透明装饰模式，Component 充当抽象组件角色，TextView 和 PictureView 充当具体组件角色，Decorator 充当抽象装饰角色，ScrollBarDecorator 和 BlackBorderDecorator 充当具体装饰角色。其中，Decorator 类和 ScrollBarDecorator 类示例代码如下：

```
class Decorator extends Component
{
    private Component component;
    public Decorator(Component component)
    {
        this.component = component;
    }
    public void display()
    {
        component.display();
    }
}

class ScrollBarDecorator extends Decorator
{
    public ScrollBarDecorator(Component component)
    {
        super(component);
    }
    public void display()
    {
        System.out.println("增加滚动条");
        super.display();
    }
}
```

21. 变形金刚在变形之前是一辆汽车，它可以在陆地上移动。当它变成机器人之后除了能够在陆地上移动之外，还可以说话；如果需要，它还可以变成飞机，除了在陆地上移动还可以在天空中飞翔。（装饰器模式，课堂例题）

参考答案：

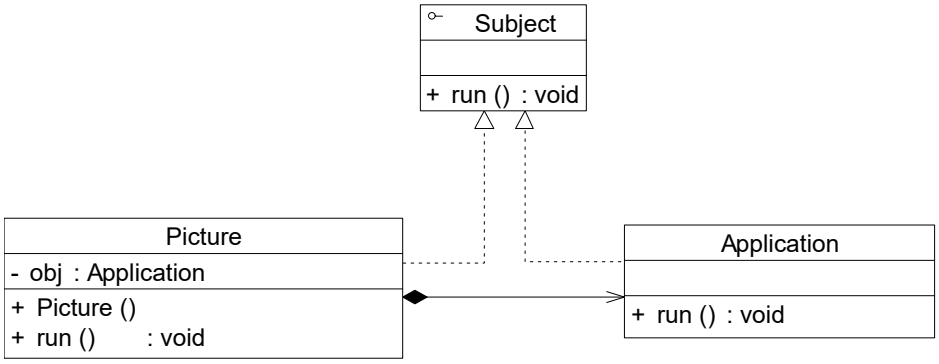


代码见教材或学习通课程资料。

22. 应用软件所提供的桌面快捷方式是快速启动应用程序的代理，桌面快捷方式一般使用一张小图片来表示(Picture)，通过调用快捷方式的 run()方法将调用应用软件(Application)的 run()方法。使用代理模式模拟该过程，绘制类图并编程模拟实现。

参考答案：

代理模式。参考类图如下所示：



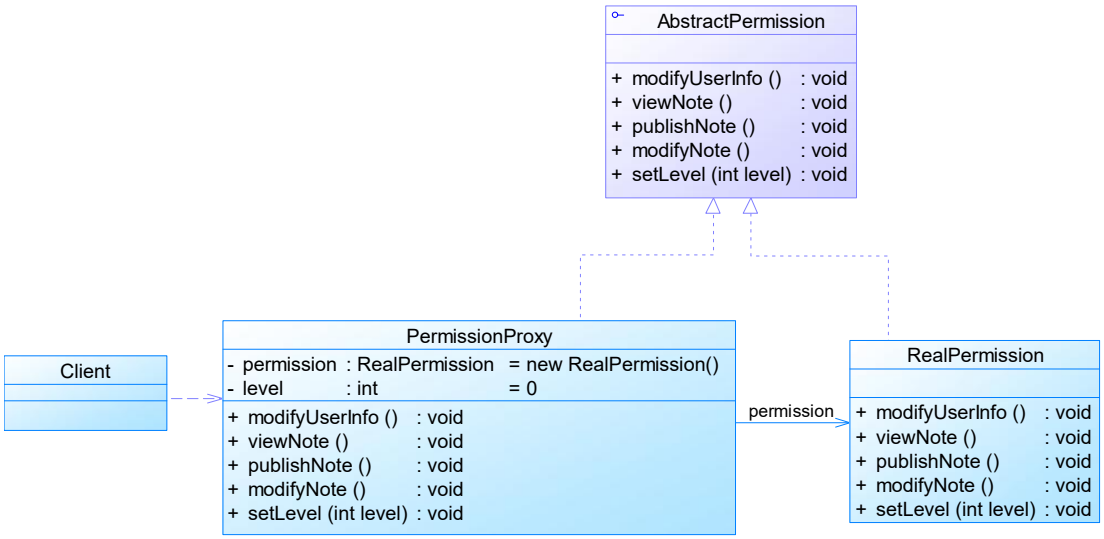
分析：在本实例中，Subject 充当抽象主题角色，Application 充当真实主题角色，Picture 充当代理主题角色，其中，Picture 类代码如下所示：

```
class Picture implements Subject
{
    private Application obj;
```

```
    public Picture()
    {
        obj = new Application();
    }
    public void run()
    {
        obj.run();
    }
}
```

23. 在一个论坛中已注册用户和游客的权限不同，已注册的用户拥有发帖、修改自己的注册信息、修改自己的帖子等功能；而游客只能看到别人发的帖子，没有其他权限。使用代理模式来设计该权限管理模块。（代理模式，课堂例题）

参考答案：



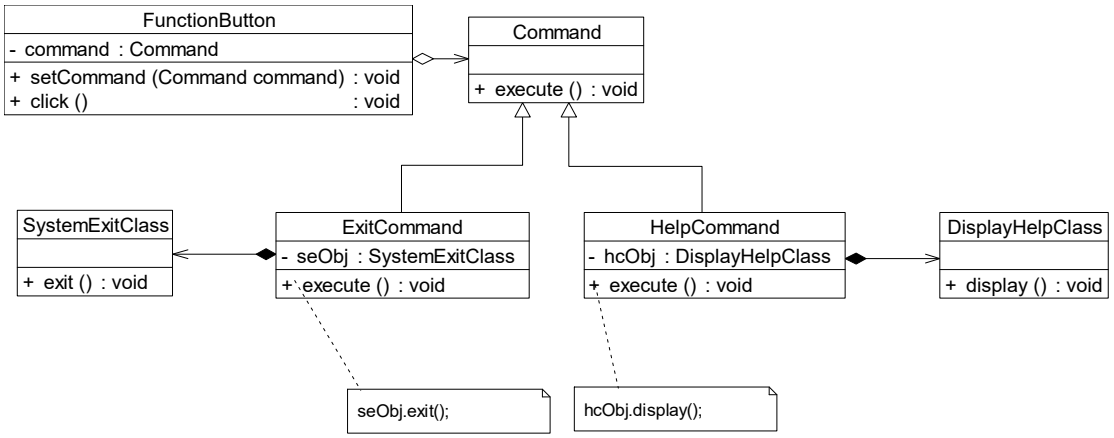
代码见教材或学习通课程资料。

(四) 行为型设计模式

24. 为了用户使用方便，某系统提供了一系列功能键，用户可以自定义功能键的功能，如功能键 FunctionButton 可以用于退出系统(SystemExitClass)，也可以用于打开帮助界面(DisplayHelpClass)。用户可以通过修改配置文件来改变功能键的用途，现使用命令模式来设计该系统，使得功能键类与功能类之间解耦，相同的功能键可以对应不同的功能。

参考答案：

命令模式。参考类图如下所示：



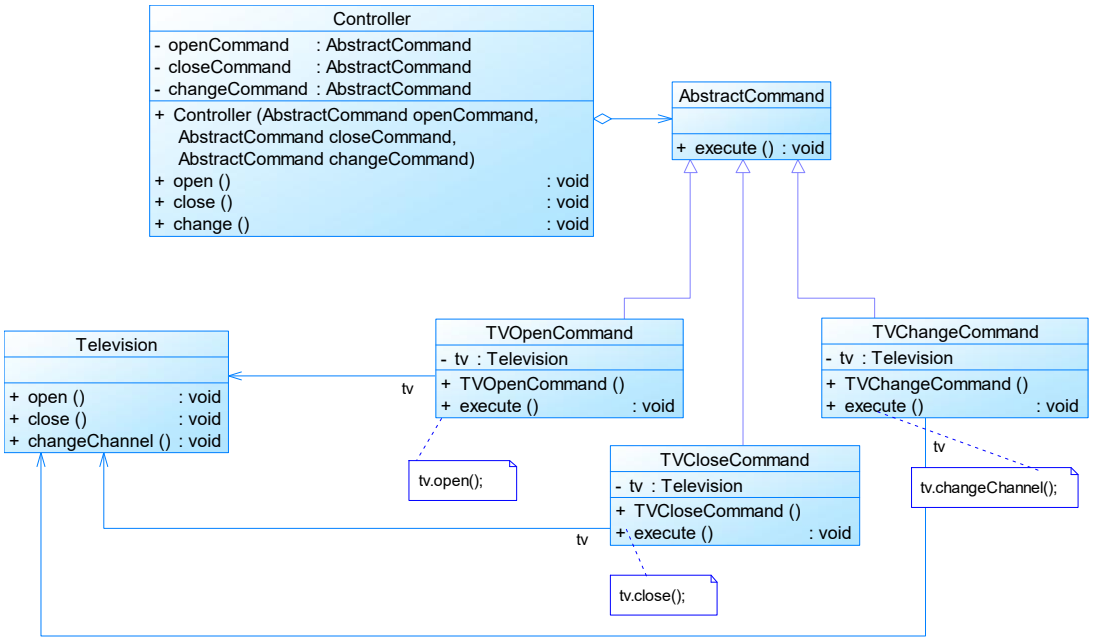
分析：在本实例中，FunctionButton 充当请求调用者，SystemExitClass 和 DisplayHelpClass 充当请求接收者，而 ExitCommand 和 HelpCommand 的充当具体命令类。其中，FunctionButton 类、Command 类、ExitCommand 类和 SystemExitClass 类代码如下所示：

```
class FunctionButton
{
    private Command command;
    public void setCommand(Command command)
    {
        this.command = command;
    }
    public void click()
    {
        command.execute();
    }
}

abstract class Command
{
    public abstract void execute();
}

class ExitCommand extends Command
{
    private SystemExitClass seObj;
    public ExitCommand()
    {
        seObj = new SystemExitClass();
    }
    public void execute()
    {
        seObj.exit();
    }
}
```

```
class SystemExitClass
{
    public void exit()
    {
        System.out.println("退出系统！");
    }
}
```

25. 电视机是请求的接收者，遥控器是请求的发送者，遥控器上有一些按钮，不同的按钮对应电视机的不同操作。抽象命令角色由一个命令接口来扮演，有三个具体的命令类实现了抽象命令接口，这三个具体命令类分别代表三种操作：打开电视机、关闭电视机和切换频道。显然，电视机遥控器就是一个典型的命令模式应用实例。（命令模式，课堂例题）

参考答案：

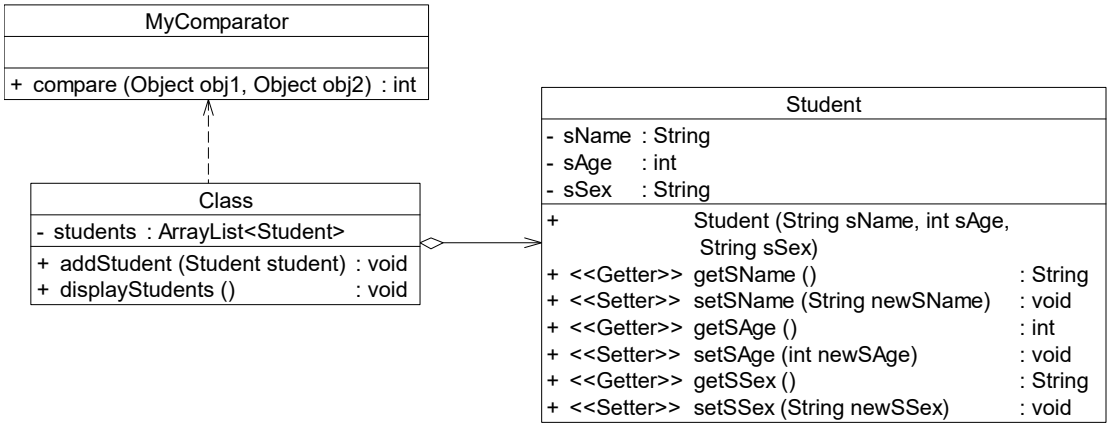


代码见教材或学习通课程资料。

26. 某教务管理系统中一个班级(Class)包含多个学生(Student)，使用 Java 内置迭代器实现对学生信息的遍历，要求按学生年龄由大到小的次序输出学生信息。用 Java 语言模拟实现该过程。

参考答案：

迭代器模式。参考类图如下所示：



分析：在本实例中，Class 类充当聚合类，在其中定义了一个 ArrayList 类型的集合用于存储 Student 对象，为了实现按学生年龄由大到小的次序输出学生信息，自定义一个比较器类 MyComparator 实现了 Comparator 接口并实现在接口中声明的 compare()方法。在 Class 类的 displayStudents()方法中创建一个比较器对象用于排序，再创建一个迭代器对象用于遍历集合。本实例代码如下所示：

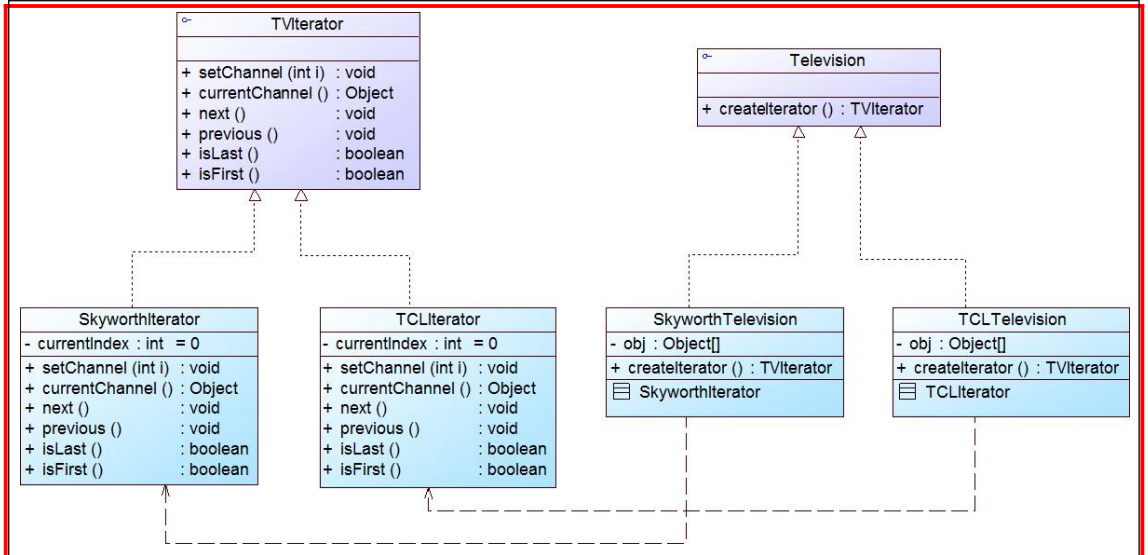
```
import java.util.*;

class Class
{
    private ArrayList<Student> students = new ArrayList<Student>();

    public void addStudent(Student student)
    {
        students.add(student);
    }

    public void displayStudents()
    {
        Comparator comp = new MyComparator();
        Collections.sort(students,comp);
        Iterator i = students.iterator();
        while(i.hasNext())
        {
            Student student = (Student)i.next();
            System.out.println("姓名：" + student.getSName() + "，年龄：" +
student.getSAge());
        }
    }
}

class MyComparator implements Comparator
{
    public int compare(Object obj1,Object obj2)
```

```
    {
        Student s1=(Student)obj1;
        Student s2=(Student)obj2;
        if(s1.getSAge()<s2.getSAge())
        {
            return 1;
        }
        else
        {
            return 0;
        }
    }
}

class Student
{
    private String sName;
    private int sAge;
    private String sSex;

    public Student(String sName,int sAge,String sSex)
    {
        this.sName = sName;
        this.sAge = sAge;
        this.sSex = sSex;
    }

    public void setSName(String sName) {
        this.sName = sName;
    }

    public void setSAge(int sAge) {
        this.sAge = sAge;
    }

    public void setSSex(String sSex) {
        this.sSex = sSex;
    }

    public String getSName() {
        return (this.sName);
    }

    public int getSAge() {
```

```
            return (this.sAge);
        }

        public String getSSex() {
            return (this.sSex);
        }
}

class MainClass
{
        public static void main(String args[])
        {
            Class obj = new Class();
            Student student1,student2,student3,student4;
            student1 = new Student("杨过",20,"男");
            student2 = new Student("令狐冲",22,"男");
            student3 = new Student("小龙女",18,"女");
            student4 = new Student("王语嫣",19,"女");
            obj.addStudent(student1);
            obj.addStudent(student2);
            obj.addStudent(student3);
            obj.addStudent(student4);
            obj.displayStudents();
        }
}
//输出结果如下：
//姓名：令狐冲，年龄：22
//姓名：杨过，年龄：20
//姓名：王语嫣，年龄：19
//姓名：小龙女，年龄：18
```

27. 电视机遥控器就是一个迭代器的实例，通过它可以实现对电视机频道集合的遍历操作，模拟电视机遥控器的实现。（迭代器模式，课堂例题）

参考答案：

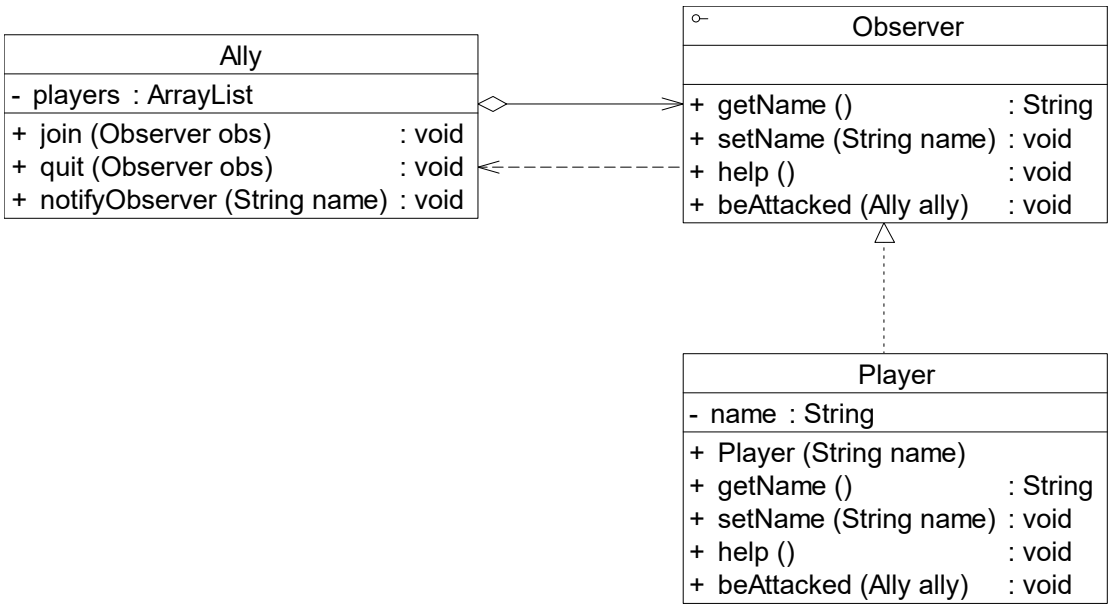


代码见教材或学习通课程资料。

28. 某在线游戏支持多人联机对战，每个玩家都可以加入某一战队组成联盟，当战队中某一成员受到敌人攻击时将给所有盟友发送通知，盟友收到通知后将作出响应。使用观察者模式设计并实现该过程。

参考答案：

观察者模式



分析：在本实例中，Observer 充当抽象观察者角色，Player 充当具体观察者角色，Ally 充当观察目标角色。其中，Ally 类和 Player 类代码如下所示：

```
class Player implements Observer
{
    private String name;

    public Player(String name)
    {
```

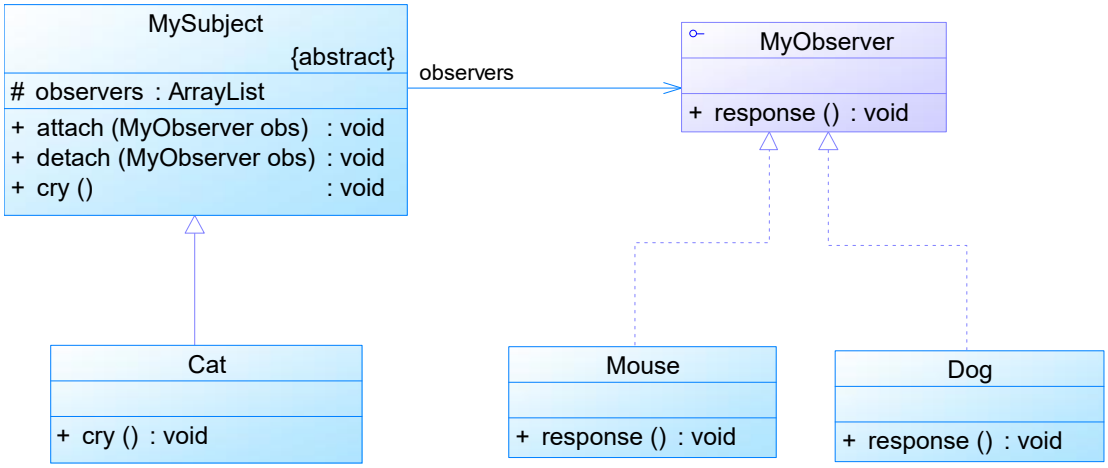
```
            this.name = name;
        }

        public void setName(String name)
        {
            this.name = name;
        }

        public String getName()
        {
            return this.name;
        }

        public void help()
        {
            System.out.println("坚持住，" + this.name + "来救你！");
        }

        public void beAttacked(Ally ally)
        {
            System.out.println(this.name + "被攻击！");
            ally.notifyObserver(name);
        }
}

class Ally
{
        private ArrayList<Observer> players = new ArrayList<Observer>();

        public void join(Observer obs)
        {
            players.add(obs);
        }

        public void quit(Observer obs)
        {
            players.remove(obs);
        }

        public void notifyObserver(String name)
        {
            System.out.println("紧急通知，盟友" + name + "遭受敌人攻击！");
            for(Object obs : players)
            {
```

```
                if (!((Observer)obs).getName().equalsIgnoreCase(name))
                {
                    ((Observer)obs).help();
                }
            }
        }
    }
```

在本实例中，应用了两次观察者模式，当一个游戏玩家 Player 对象的 beAttacked()方法被调用时，将调用 Ally 的 notifyObserver()方法来进行处理，而在 notifyObserver()方法中又将调用其他 Player 对象的 help()方法。Player 的 beAttacked()方法、Ally 的 notifyObserver()方法以及 Player 的 help()方法构成了一个简单的触发链。

29. 假设猫是老鼠和狗的观察目标，老鼠和狗是观察者，猫叫老鼠跑，狗也跟着叫，使用观察者模式描述该过程。（观察者模式，课堂例题）

参考答案：

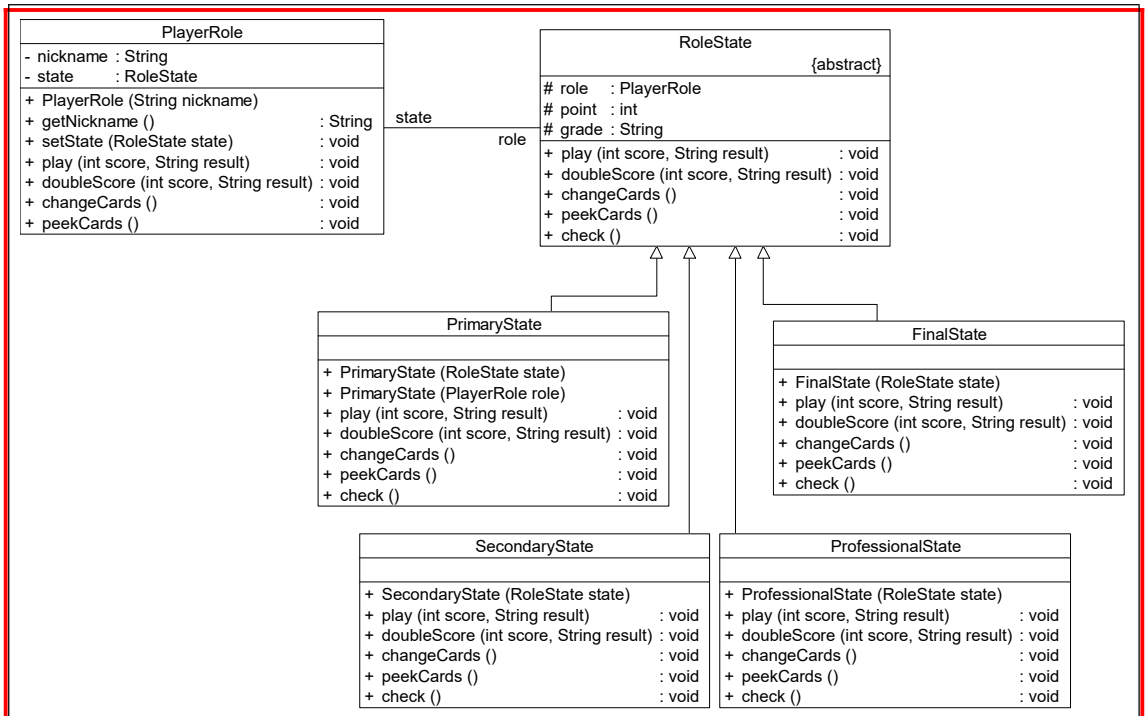


代码见教材或学习通课程资料。

30. 某纸牌游戏软件中，人物角色具有入门级(Primary)、熟练级(Secondary)、高手级(Professional)和骨灰级(Final)四种等级，角色的等级与其积分相对应，游戏胜利将增加积分，失败则扣除积分。入门级具有最基本的游戏功能 play() ，熟练级增加了游戏胜利积分加倍功能 doubleScore()，高手级在熟练级基础上再增加换牌功能 changeCards()，骨灰级在高手级基础上再增加偷看他人的牌功能 peekCards()。现使用状态模式来设计该系统，绘制类图并编程实现。

参考答案：

状态模式。参考类图如下所示：



分析：在本实例中，PlayerRole 充当环境类角色，RoleState 充当抽象状态类，PrimaryState、SecondaryState、ProfessionalState 和 FinalState 充当具体状态类。本实例部分代码如下所示：

```
class PlayerRole   //环境类
{
    private String nickname;
    private RoleState state;
    public PlayerRole(String nickname)
    {
        this.nickname = nickname;
    }
    public String getNickname()
    {
        return this.nickname;
    }
    public void setState(RoleState state)
    {
        this.state = state;
    }
    public void play(int score, String result)
    {
        state.play(score,result);
    }
    public void doubleScore(int score, String result)
    {
        state.doubleScore(score,result);
    }
    public void changeCards()
```

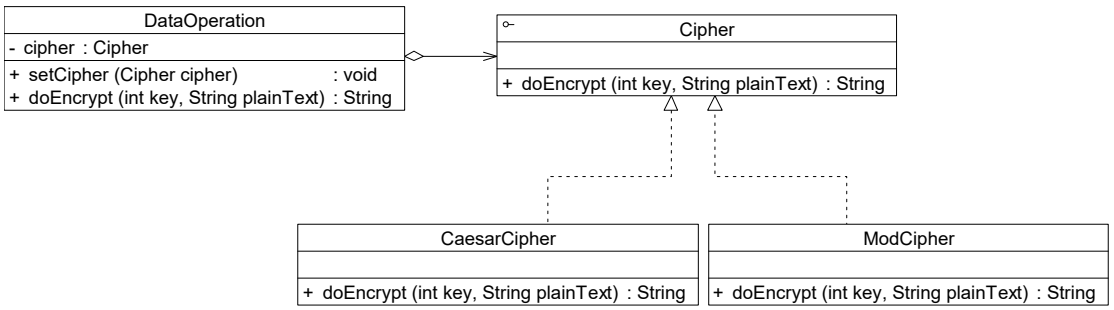
```
    {
        state.changeCards();
    }
    public void peekCards()
    {
        state.peekCards();
    }
}

abstract class RoleState   //抽象状态类
{
    protected PlayerRole role;
    protected int point;   //积分
    protected String grade;   //等级
    public abstract void play(int score, String result);
    public abstract void doubleScore(int score, String result);
    public abstract void changeCards();
    public abstract void peekCards();
    public abstract void check();
}

class PrimaryState extends RoleState     //具体状态类
{
    public PrimaryState(PlayerRole role)
    {
        this.point = 0;
        this.grade = "入门级";
        this.role = role;
    }
    public PrimaryState(RoleState state)
    {
        this.point = state.point;
        this.grade = "入门级";
        this.role = state.role;
    }
    public void play(int score, String result)
    {
        if(result.equalsIgnoreCase("win"))   //获胜
        {
            this.point += score;
            System.out.println("玩家" + this.role.getNickname() + "获胜，增加积分" +
score + "，当前积分为" + this.point + "。");
        }
        else if(result.equalsIgnoreCase("lose"))   //失利
```

```
            {
                this.point -= score;
                System.out.println("玩家" + this.role.getNickname() + "失利，减少积分" +
score + "，当前积分为" + this.point + "。");
            }
            this.check();
        }
        public void doubleScore(int score, String result)
        {
            System.out.println("暂不支持该功能！");
        }
        public void changeCards()
        {
            System.out.println("暂不支持该功能！");
        }
        public void peekCards()
        {
            System.out.println("暂不支持该功能！");
        }
        public void check()    //模拟
        {
            if(this.point >= 10000)
            {
                this.role.setState(new FinalState(this));
            }
            else if(this.point >= 5000)
            {
                this.role.setState(new ProfessionalState(this));
            }
            else if(this.point >= 1000)
            {
                this.role.setState(new SecondaryState(this));
            }
        }
}
//其他具体状态类代码省略
```

31. 某系统需要对重要数据（如用户密码）进行加密，并提供了几种加密方案（如凯撒加密、DES 加密等），对该加密模块进行设计，使得用户可以动态选择加密方式。要求绘制类图并编程模拟实现。

参考答案：

策略模式。参考类图如下所示：

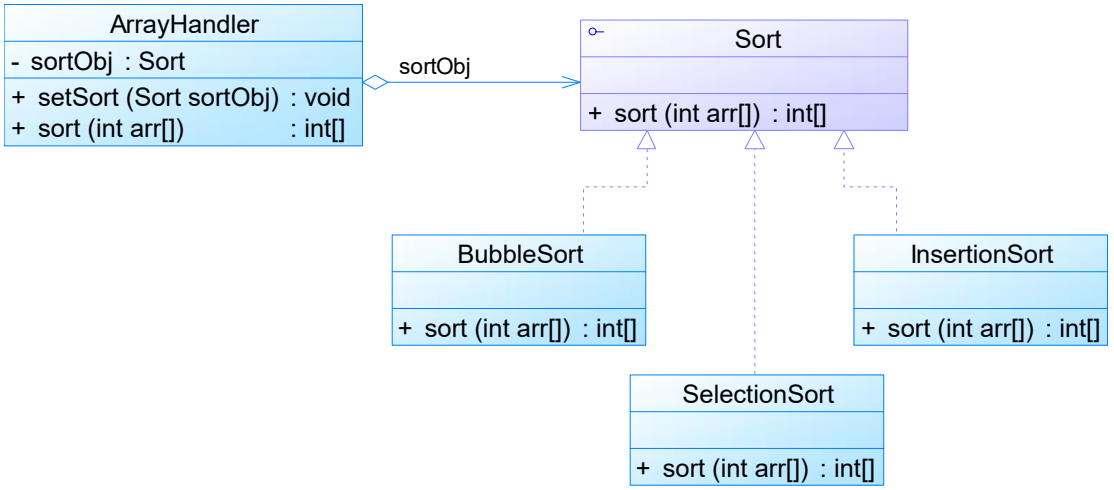


分析：在本实例中，DataOperation 充当环境类角色，Cipher 充当抽象策略角色，CaesarCipher 和 ModCipher 充当具体策略角色。其中，DataOperation 类的代码如下所示：

```
class DataOperation
{
    private Cipher cipher;
    public void setCipher(Cipher cipher)
    {
        this.cipher = cipher;
    }
    public String doEncrypt(int key, String plainText)
    {
        return cipher.doEncrypt(key,plainText);
    }
}
```

32. 某系统提供了一个用于对数组数据进行操作的类，该类封装了对数组的常见操作，如查找数组元素、对数组元素进行排序等。现以排序操作为例，使用策略模式设计该数组操作类，使得客户端可以动态地更换排序算法，可以根据需要选择冒泡排序或选择排序或插入排序，也能够灵活地增加新的排序算法。（策略模式，课堂例题）

参考答案：



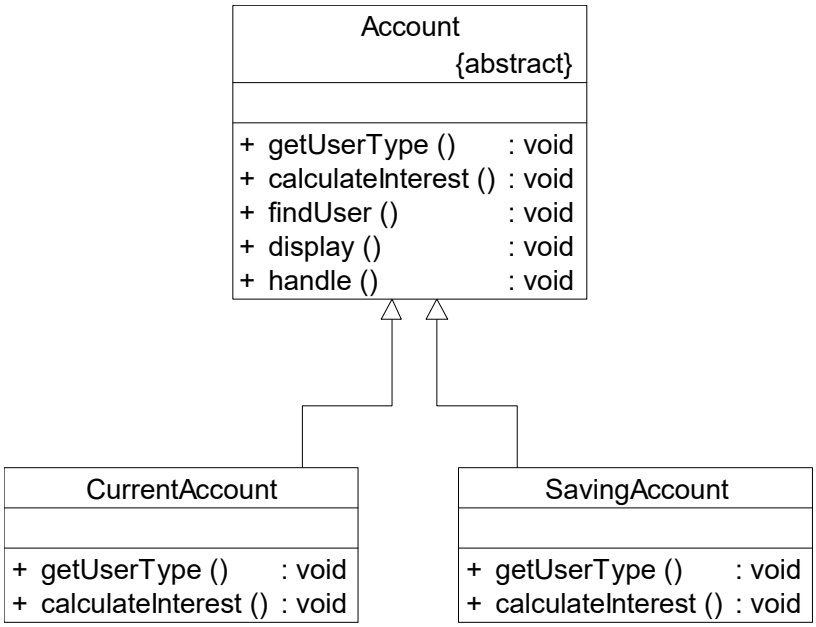
代码见教材或学习通课程资料。

33. 某银行软件的利息计算流程如下：系统根据账户查询用户信息；根据用户信息判断用户

类型；不同类型的用户使用不同的利息计算方式计算利息（如活期账户 CurrentAccount 和定期账户 SavingAccount 具有不同的利息计算方式）；显示利息。现使用模板方法模式来设计该系统，绘制类图并编程实现。

参考答案：

模板方法模式。参考类图如下所示：



分析：在本实例中，Account 充当抽象父类角色，CurrentAccount 和 SavingAccount 充当具体子类角色，其中 Account 和 CurrentAccount 的模拟代码如下所示：

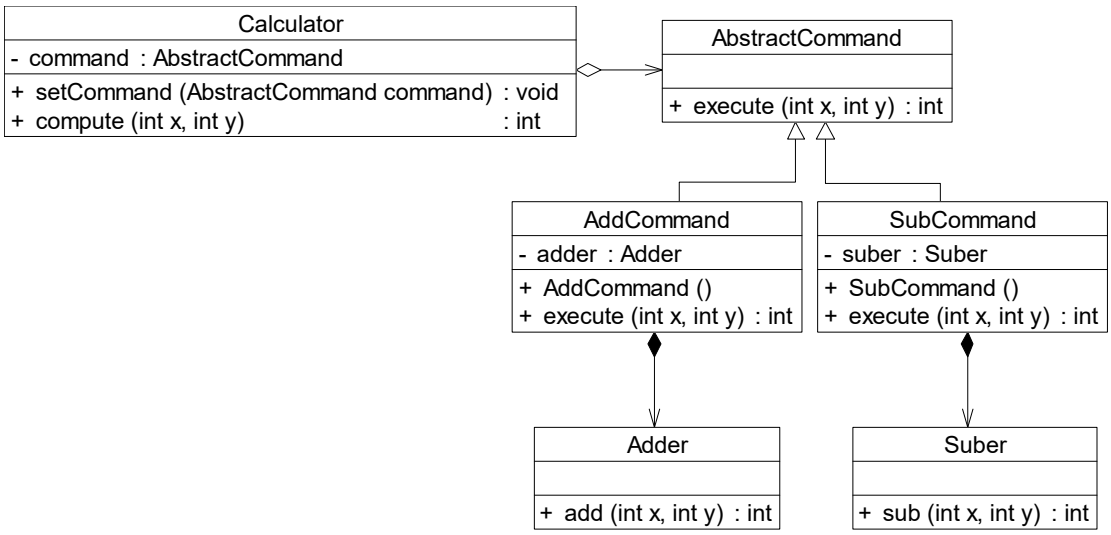
```
abstract class Account
{
    public abstract void getUserType();
    public abstract void calculateInterest();
    public void findUser()
    {
        System.out.println("查询用户信息！");
    }
    public void display()
    {
        System.out.println("显示利息！");
    }
    public void handle()
    {
        findUser();
        getUserType();
        calculateInterest();
        display();
    }
}

class CurrentAccount extends Account
```

```
{
    public void getUserType()
    {
        System.out.println("活期账户！");
    }
    public void calculateInterest()
    {
        System.out.println("按活期利率计算利息！");
    }
}
```

34. （模式综合题）利用设计模式，设计并实现一个加减计算器，设计时需考虑系统的可扩展性。

参考答案：

本题可使用命令模式，参考类图如下所示：



参考代码如下：

```
class Calculator       //请求调用者
{
    private AbstractCommand command;
    public void setCommand(AbstractCommand command)
    {
        this.command = command;
    }
    public int compute(int x,int y)
    {
        return command.execute(x,y);
    }
}
```

```
abstract class AbstractCommand       //抽象命令
{
    public abstract int execute(int x,int y);
}

class AddCommand extends AbstractCommand       //具体命令
{
    private Adder adder;
    public AddCommand()
    {
        adder = new Adder();
    }
    public int execute(int x,int y)
    {
        return adder.add(x,y);
    }
}

class Adder       //请求接收者
{
    public int add(int x,int y)
    {
        return x+y;
    }
}

class SubCommand extends AbstractCommand       //具体命令
{
    private Suber suber;
    public SubCommand()
    {
        suber = new Suber();
    }
    public int execute(int x,int y)
    {
        return suber.sub(x,y);
    }
}

class Suber       //请求接收者
{
    public int sub(int x,int y)
    {
        return x-y;
```

```
    }
}

class Client       //客户端测试类
{
    public static void main(String args[])
    {
        Calculator calculator = new Calculator();
        AbstractCommand command;
        command = new AddCommand();       //可通过配置文件实现
        calculator.setCommand(command);
        int result = calculator.compute(10,10);
        System.out.println(result);
    }
}
//输出结果如下：
//10
```