《Java 设计模式》课后习题参考答案

教材：刘伟. Java 设计模式. 北京：清华大学出版社, 2018.
ISBN: 9787302488316

【说明：本答案供参考，如有意见或建议，请通过电子邮箱 weiliu_china@126.com 与作者联系！】

第 1 章 设计模式概述

1. D
2. C
3. B
4. 参见教材 P5。
5. 参见教材 P6-P7。
6. 参见教材 P9-P10。
7. 反模式(AntiPatterns)是指那些导致开发出现障碍的负面模式，即在软件开发中普遍存在、反复出现并会影响到软件成功开发的不良解决方案。反模式是关注于负面解决方案的软件研究方向，揭示出不成功系统中存在的反模式有利于在成功系统中避免出现这些模式，有助于降低软件缺陷和项目失败出现的频率。反模式清晰定义了大部分人在软件开发过程中经常会犯的一些错误，根据视角的不同，可分为开发性反模式、架构性反模式和管理性反模式。
8. JDK 中部分设计模式使用示例列举如下：

创建型模式：

(1) 抽象工厂模式(Abstract Factory)

- java.util.Calendar#getInstance()
- java.util.Arrays#asList()
- java.util.ResourceBundle#getBundle()
- java.net.URL#openConnection()
- java.sql.DriverManager#getConnection()
- java.sql.Connection#createStatement()
- java.sql.Statement#executeQuery()
- java.text.NumberFormat#getInstance()
- java.lang.management.ManagementFactory (所有 getXXX()方法)
- java.nio.charset.Charset#forName()
- javax.xml.parsers.DocumentBuilderFactory#newInstance()
- javax.xml.transform.TransformerFactory#newInstance()
- javax.xml.xpath.XPathFactory#newInstance()

(2) 建造者模式(Builder)

- java.lang.StringBuilder#append()
- java.lang.StringBuffer#append()
- java.nio.ByteBuffer#put() (CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer 和 DoubleBuffer 与之类似)
- javax.swing.GroupLayout.Group#addComponent()
- java.sql.PreparedStatement

• java.lang.Appendable 的所有实现类

(3) 工厂方法模式(Factory Method)

• java.lang.Object#toString() (在其子类中可以覆盖该方法)

• java.lang.Class#newInstance()

• java.lang.Integer#valueOf(String) (Boolean, Byte, Character, Short, Long, Float 和 Double 与之类似)

• java.lang.Class#forName()

• java.lang.reflect.Array#newInstance()

• java.lang.reflect.Constructor#newInstance()

(4) 原型模式(Prototype)

• java.lang.Object#clone() (支持浅克隆的类必须实现 java.lang.Cloneable 接口)

(5) 单例模式 (Singleton)

• java.lang.Runtime#getRuntime()

• java.awt.Desktop#getDesktop()

结构型模式:

(1) 适配器模式(Adapter)

• java.util.Arrays#asList()

• javax.swing.JTable(TableModel)

• java.io.InputStreamReader(InputStream)

• java.io.OutputStreamWriter(OutputStream)

• javax.xml.bind.annotation.adapters.XmlAdapter#marshal()

• javax.xml.bind.annotation.adapters.XmlAdapter#unmarshal()

(2) 桥接模式(Bridge)

• AWT (提供了抽象层映射于实际的操作系统)

• JDBC

(3) 组合模式(Composite)

• javax.swing.JComponent#add(Component)

• java.awt.Container#add(Component)

• java.util.Map#putAll(Map)

• java.util.List#addAll(Collection)

• java.util.Set#addAll(Collection)

(4) 装饰模式(Decorator)

• java.io.BufferedInputStream(InputStream)

• java.io.DataInputStream(InputStream)

• java.io.BufferedOutputStream(OutputStream)

• java.util.zip.ZipOutputStream(OutputStream)

• java.util.Collections#checked[List|Map|Set|SortedSet|SortedMap]()

(5) 外观模式(Facade)

• java.lang.Class

• javax.faces.webapp.FacesServlet

(6) 享元模式(Flyweight)

• java.lang.Integer#valueOf(int)

• java.lang.Boolean#valueOf(boolean)

• java.lang.Byte#valueOf(byte)

• java.lang.Character#valueOf(char)
(7) 代理模式(Proxy)
• java.lang.reflect.Proxy
• java.rmi.*
行为型模式：
(1) 职责链模式(Chain of Responsibility)
• java.util.logging.Logger#log()
• javax.servlet.Filter#doFilter()
(2) 命令模式(Command)
• java.lang.Runnable
• javax.swing.Action
(3) 解释器模式(Interpreter)
• java.util.Pattern
• java.text.Normalizer
• java.text.Format
• javax.el.ELResolver
(4) 迭代器模式(Iterator)
• java.util.Iterator
• java.util.Enumeration
(5) 中介者模式(Mediator)
• java.util.Timer (所有 scheduleXXX()方法)
• java.util.concurrent.Executor#execute()
• java.util.concurrent.ExecutorService (invokeXXX()和 submit()方法)
• java.util.concurrent.ScheduledExecutorService (所有 scheduleXXX()方法)
• java.lang.reflect.Method#invoke()
(6) 备忘录模式(Memento)
• java.util.Date
• java.io.Serializable
• javax.faces.component.StateHolder
(7) 观察者模式(Observer)
• java.util.Observer/java.util.Observable
• java.util.EventListener (所有子类)
• javax.servlet.http.HttpSessionBindingListener
• javax.servlet.http.HttpSessionAttributeListener
• javax.faces.event.PhaseListener
(8) 状态模式(State)
• java.util.Iterator
• javax.faces.lifecycle.LifeCycle#execute()
(9) 策略模式(Strategy)
• java.util.Comparator#compare()
• javax.servlet.http.HttpServlet
• javax.servlet.Filter#doFilter()
(10) 模板方法模式(Template Method)
• java.io.InputStream, java.io.OutputStream, java.io.Reader 和 java.io.Writer 的所有非抽象

方法

• java.util.AbstractList, java.util.AbstractSet 和 java.util.AbstractMap 的所有非抽象方法

• javax.servlet.http.HttpServlet#doXXX()

(11) 访问者模式(Visitor)

• javax.lang.model.element.AnnotationValue 和 AnnotationValueVisitor

• javax.lang.model.element.Element 和 ElementVisitor

• javax.lang.model.type.TypeMirror 和 TypeVisitor

参见：http://www.iteye.com/news/18725 和 http://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns。

第 2 章 面向对象设计原则

1. B A C D D C

2. C

3. D

4. D

5. “封装变化点”可对应“开闭原则”，“对接口进行编程”可对应“依赖倒转原则”，“多使用组合，而不是继承”可对应“合成复用原则”。

6. 类的粒度需满足单一职责原则，接口的粒度需满足接口隔离原则。

7. 在面向对象设计中，正方形不能作为长方形的子类，具体分析过程如下：

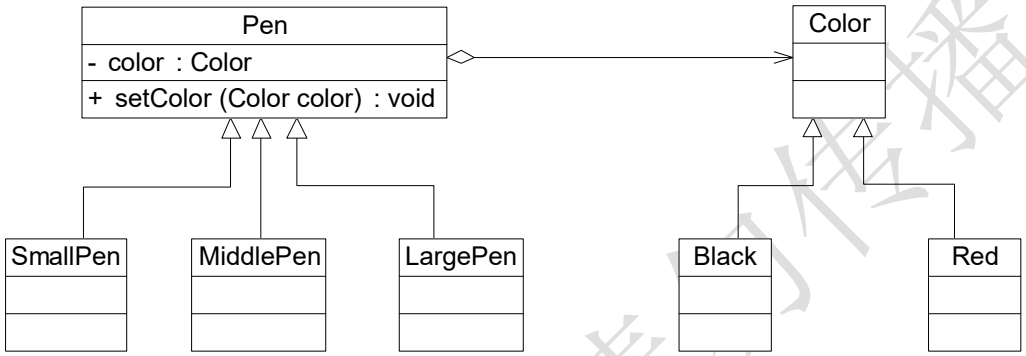
```
class Rectangle       //长方形
{
      private double width;
      private double height;

      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。

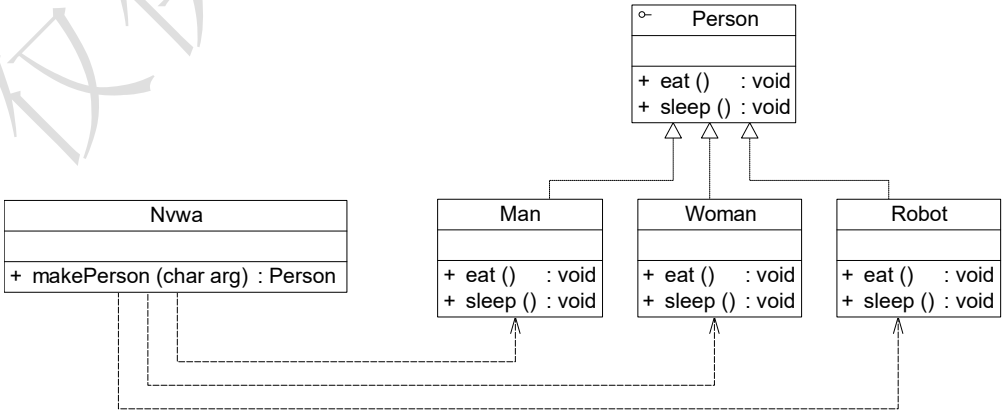
8. 重构方案如下所示：



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：本重构方案即为**桥接模式**）

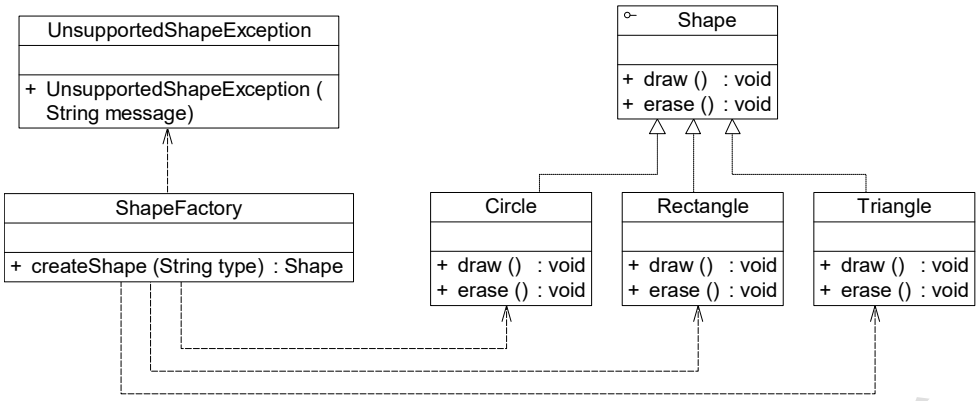
第 3 章 简单工厂模式

1. C
2. C
3. A
4. 参考类图如下：



其中，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。

5. 参考类图如下：



其中，Shape 接口充当抽象产品，其子类 Circle、Rectangle 和 Triangle 等充当具体产品，ShapeFactory 充当工厂类。

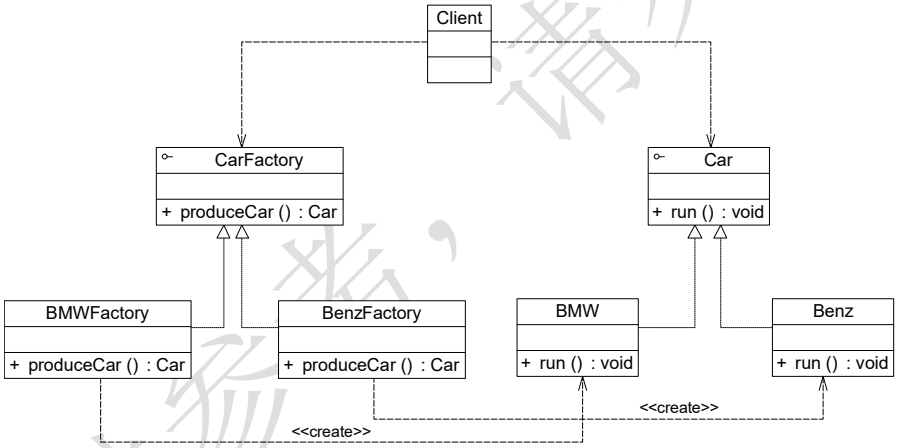
第 4 章 工厂方法模式

1. B

2. D

3. A B

4. 参考类图如下所示：

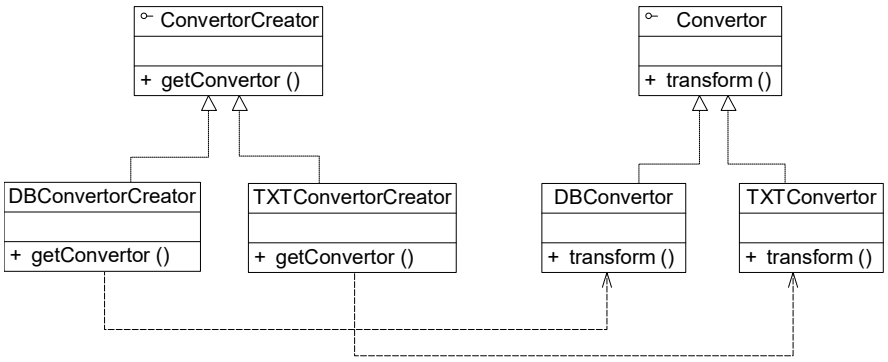


其中，Car 充当抽象产品，其子类 BMW 和 Benz 充当具体产品；CarFactory 充当抽象工厂，其子类 BMWFactory 和 BenzFactory 充当具体工厂。

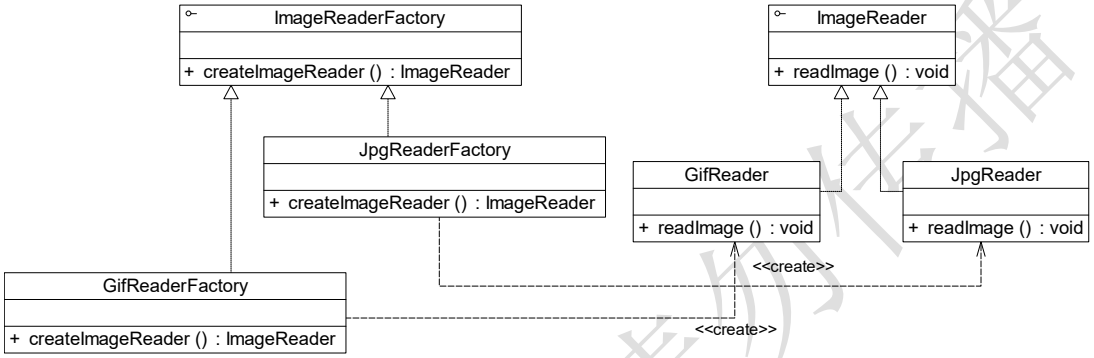
5. 抽象类/接口 Chart 充当抽象产品，其子类 LineChart 和 BarChart 充当具体产品；抽象类/接口 ChartFactory 充当抽象工厂，其子类 LineChartFactory 和 BarChartFactory 充当具体工厂。

6. 抽象类/接口 Convertor 充当抽象产品，其子类 TXTConvertor、DBConvertor 和 ExcelConvertor 充当具体产品；抽象类/接口 ConvertorCreator 充当抽象工厂，其子类 TXTConvertorCreator、DBConvertorCreator 和 ExcelConvertorCreator 充当具体工厂。

参考类图如下：



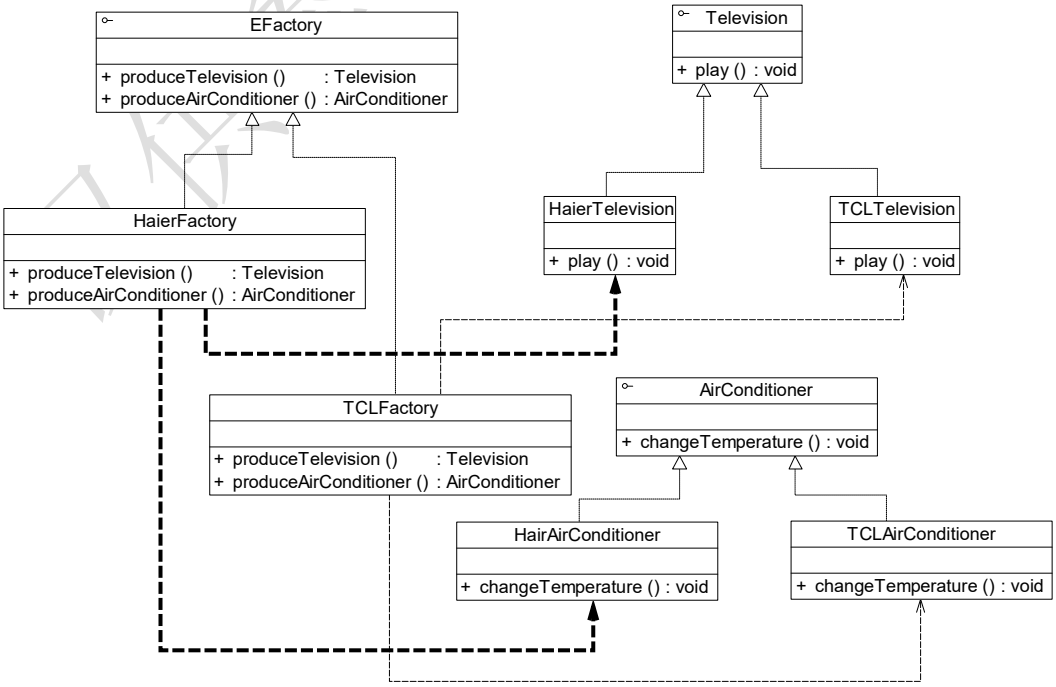
7. 参考类图如下：



其中，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。

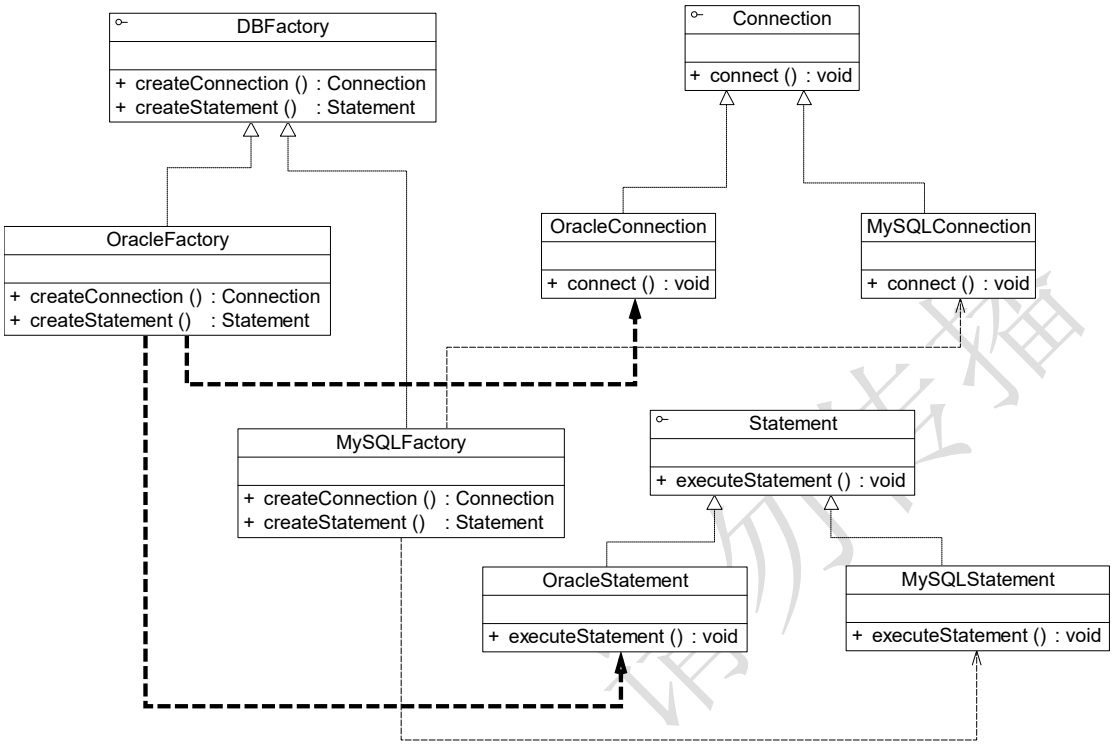
第 5 章 抽象工厂模式

1. D
2. D
3. A
4. 参考类图如下所示：



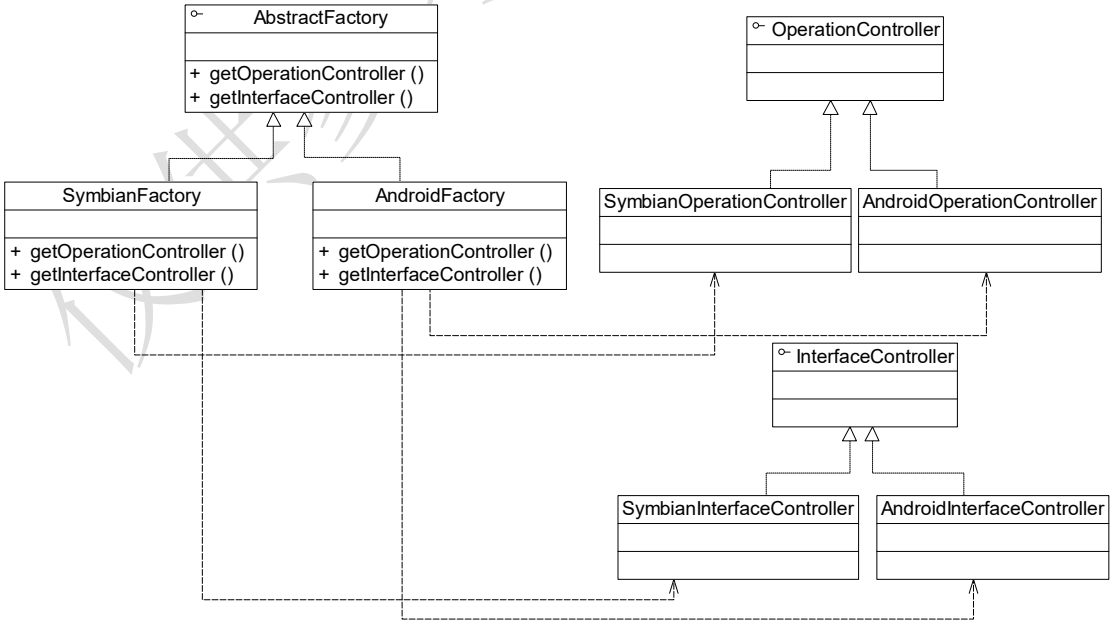
其中，EFactory 充当抽象工厂，HaierFactory 和 TCLFactory 充当具体工厂，Television 和 AirConditioner 充当抽象产品，HaierTelevision、TCLTelevision、HaierAirConditioner 和 TCLAirConditioner 充当具体产品。

5. 参考类图如下所示：



其中，接口 DBFactory 充当抽象工厂，其子类 OracleFactory 和 MySQLFactory 充当具体工厂，接口 Connection 和 Statement 充当抽象产品，其子类 OracleConnection、MySQLConnection 和 OracleStatement、MySQLStatement 充当具体产品。

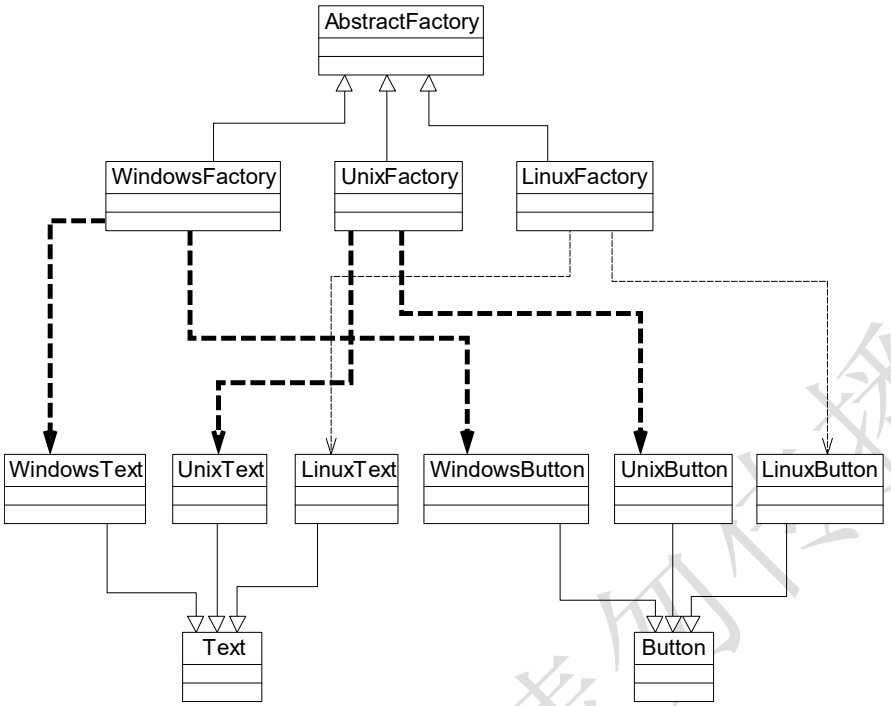
6. 参考类图如下所示：



其中，接口 AbstractFactory 充当抽象工厂，其子类 SymbianFactory 和 AndroidFactory 充当具体工厂；OperationController 和 InterfaceController 充当抽象产品，其子类 SymbianOperationController、AndroidOperationController、SymbianInterfaceController 和

AndroidInterfaceController 充当具体产品。

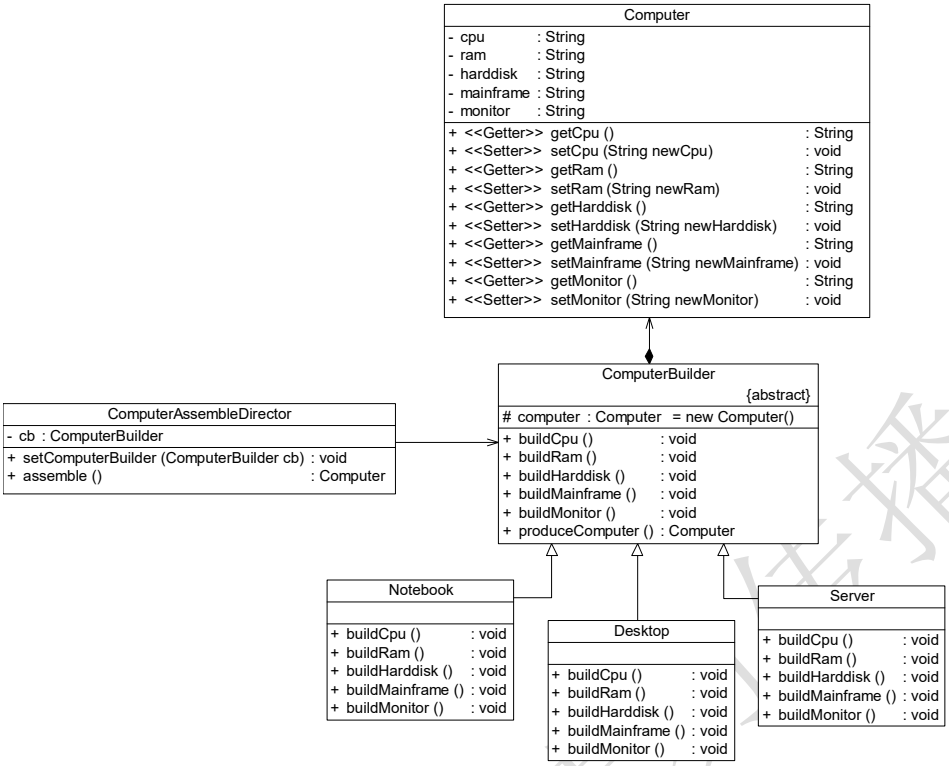
7. 参考类图如下所示：



其中，接口 AbstractFactory 充当抽象工厂，其子类 WindowsFactory、UnixFactory 和 LinuxFactory 充当具体工厂；Text 和 Button 充当抽象产品，其子类 WindowsText、UnixText、LinuxText 和 WindowsButton、UnixButton、LinuxButton 充当具体产品。

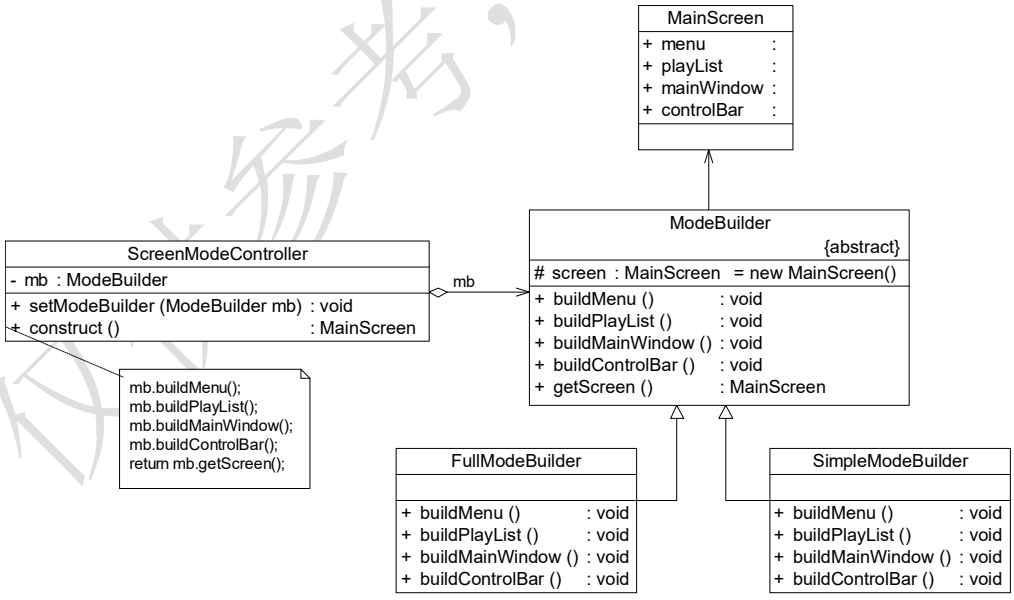
第 6 章 建造者模式

1. D
2. C
3. D
4. 参考类图如下所示：



其中，Computer 充当复合产品，ComputerBuilder 充当抽象建造者，Notebook、Desktop 和 Server 充当具体建造者，ComputerAssembleDirector 充当指挥者，其 assemble()方法用于定义产品的构造过程。

5. 参考类图如下所示：



在该设计方案中，MainScreen 是播放器的主界面，它是一个复合对象，包括菜单、播放列表、主窗口和控制条等成员。ModeBuilder 是一个抽象类，定义了一组抽象方法 buildXXX()用于逐步构造一个完整的 MainScreen 对象，getScreen()是工厂方法，用于返回一个构造好的 MainScreen 对象。ScreenModeController 充当指挥者，用于指导复合对象的创建，其中 construct()方法封装了具体创建流程，并向客户类返回完整的产品对象。

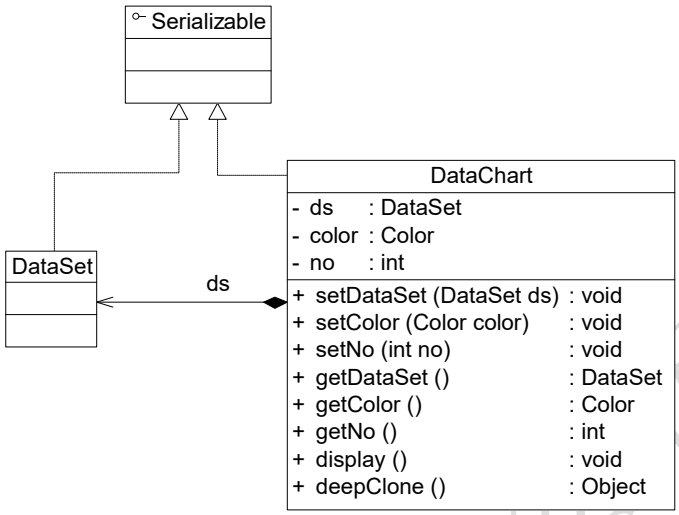
第 7 章 原型模式

1. A

2. D

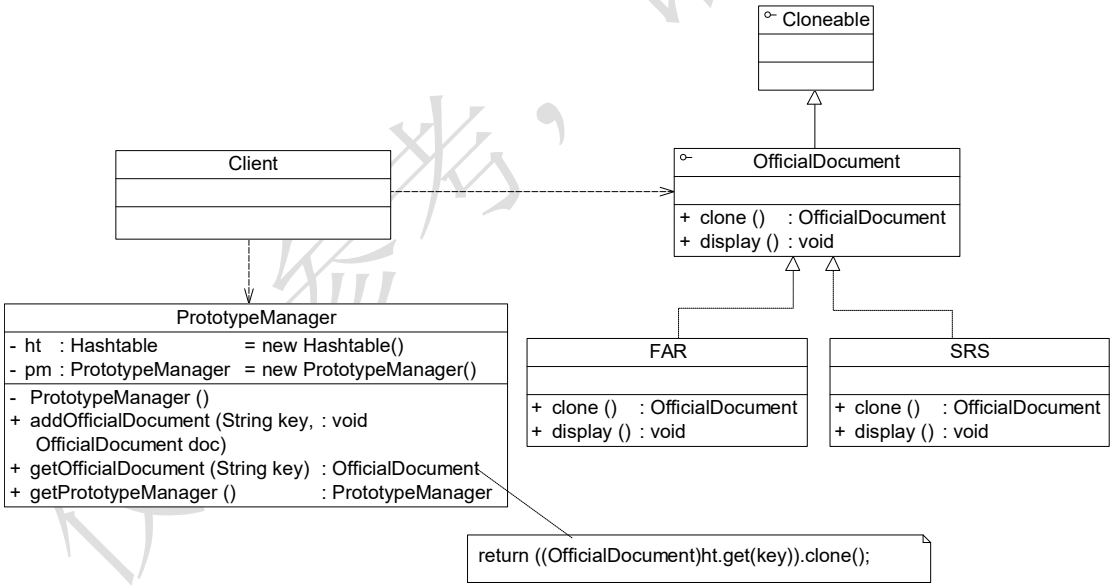
3. C

4. 参考类图如下所示：



在该设计方案中，DataChart 类包含一个 DataSet 对象，在复制 DataChart 对象的同时将复制 DataSet 对象，因此需要使用深克隆技术，可使用流来实现深克隆。

5. 参考类图如下所示：



其中，OfficialDocument（抽象公文类）充当抽象原型类，其子类 FAR（Feasibility Analysis Report，可行性分析报告）和 SRS（Software Requirements Specification，软件需求规格说明书）充当具体原型类，PrototypeManager 充当原型管理器。核心代码如下：

```
import java.util.*;

//抽象公文接口，也可定义为抽象类，提供 clone()方法的实现，将业务方法声明为抽象方法
interface OfficialDocument extends Cloneable {
    public OfficialDocument clone();
```

```
    public void display();
}

//可行性分析报告(Feasibility Analysis Report)类
class FAR implements OfficialDocument {
    public OfficialDocument clone() {
        OfficialDocument far = null;
        try {
            far = (OfficialDocument)super.clone();
        }
        catch(CloneNotSupportedException e) {
             System.out.println("不支持复制！");
        }
        return far;
    }

    public void display() {
        System.out.println("《可行性分析报告》");
    }
}

//软件需求规格说明书(Software Requirements Specification)类
class SRS implements OfficialDocument {
    public OfficialDocument clone() {
        OfficialDocument srs = null;
        try {
            srs = (OfficialDocument)super.clone();
        }
        catch(CloneNotSupportedException e) {
            System.out.println("不支持复制！");
        }
        return srs;
    }

    public void display() {
        System.out.println("《软件需求规格说明书》");
    }
}

//原型管理器（使用饿汉式单例实现）
class PrototypeManager {
    //定义一个 Hashtable，用于存储原型对象
    private Hashtable ht=new Hashtable();
    private static PrototypeManager pm = new PrototypeManager();
```

```
    //为 Hashtable 增加公文对象
    private PrototypeManager() {
        ht.put("far",new FAR());
        ht.put("srs",new SRS());
    }

    //增加新的公文对象
    public void addOfficialDocument(String key,OfficialDocument doc) {
        ht.put(key,doc);
    }

    //通过浅克隆获取新的公文对象
    public OfficialDocument getOfficialDocument(String key) {
        return ((OfficialDocument)ht.get(key)).clone();
    }

    public static PrototypeManager getPrototypeManager() {
        return pm;
    }
}
```

客户端代码如下所示：

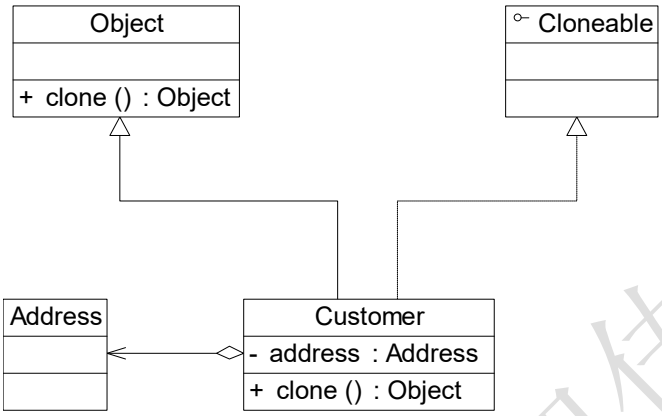
```
class Client {
    public static void main(String args[]) {
        //获取原型管理器对象
        PrototypeManager pm = PrototypeManager.getPrototypeManager();

        OfficialDocument doc1,doc2,doc3,doc4;

        doc1 = pm.getOfficialDocument("far");
        doc1.display();
        doc2 = pm.getOfficialDocument("far");
        doc2.display();
        System.out.println(doc1 == doc2);

        doc3 = pm.getOfficialDocument("srs");
        doc3.display();
        doc4 = pm.getOfficialDocument("srs");
        doc4.display();
        System.out.println(doc3 == doc4);
    }
}
```

编译并运行程序，输出结果如下：

```
《可行性分析报告》
```

```
《可行性分析报告》
false
《软件需求规格说明书》
《软件需求规格说明书》
false
```

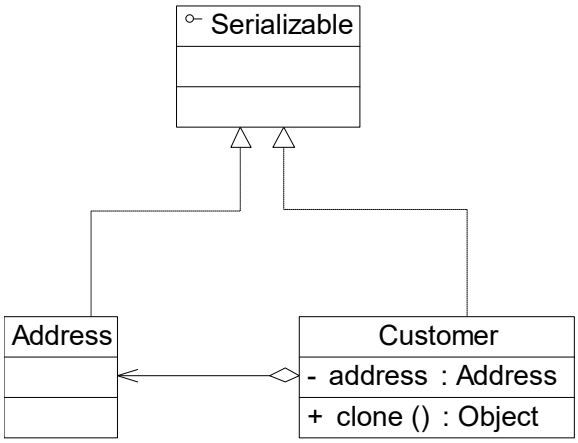
6. 浅克隆参考类图如下所示：



本实例实现了浅克隆，Object 类充当抽象原型类，Customer 类充当具体原型类，浅克隆只复制容器对象，不复制成员对象。Customer 类的代码片段如下所示：

```
public class Customer implements Cloneable
{
    private Address address = null;
    public Customer()
    {
        this.address = new Address();
    }
    //浅克隆方法
    public Object clone()
    {
        Object obj = null;
        try
        {
            obj = super.clone();
        }
        catch(CloneNotSupportedException e)
        {
            System.out.println("Clone failure!");
        }
        return obj;
    }
    //其他代码省略
}
```

深克隆参考类图如下所示：



本实例实现了深克隆，Customer 和 Address 类均实现了 Serializable 接口，深克隆既复制容器对象，又复制成员对象。Customer 的代码如下所示：

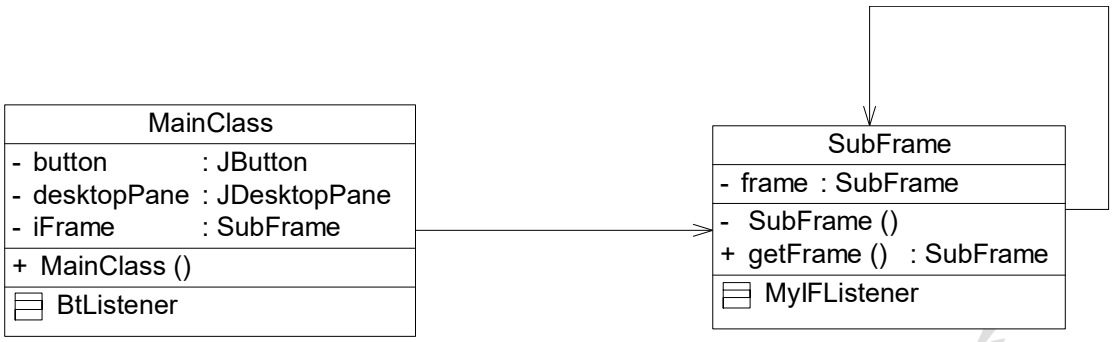
```
import java.io.*;
public class Customer implements Serializable
{
    private Address address = null;
    public Customer()
    {
        this.address=new Address();
    }
    //深克隆方法
    public Object deepClone() throws IOException, ClassNotFoundException, OptionalDataException
    {
        //将对象写入流中
        ByteArrayOutputStream bao=new ByteArrayOutputStream();
        ObjectOutputStream oos=new ObjectOutputStream(bao);
        oos.writeObject(this);

        //将对象从流中取出
        ByteArrayInputStream bis=new ByteArrayInputStream(bao.toByteArray());
        ObjectInputStream ois=new ObjectInputStream(bis);
        return(ois.readObject());
    }
    //其他代码省略
}
```

第 8 章 单例模式

1. B
2. B
3. B
4. 参见 P111-P114，可从延迟加载、线程安全、响应时间等角度进行分析与对比。
5. 参见 P112-P113。

6. 双重检查锁定方式实现代码参见 P113；IoDH 方式实现代码参见 P114。

7. 参考类图如下：



其中，SubFrame 类充当单例类，在其中定义了静态工厂方法 getFrame()。

参考代码如下所示：

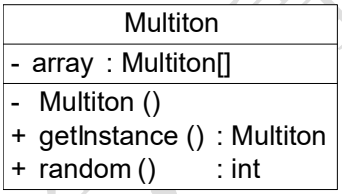
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

//子窗口：单例类
class SubFrame extends JInternalFrame
{
    private static SubFrame frame;//静态实例

    //私有构造函数
    private SubFrame()
    {
        super("子窗体", true, true, true, false);
        this.setLocation(20,20);  //设置内部窗体位置
        this.setSize(200,200);  //设置内部窗体大小
        this.addInternalFrameListener(new MyIFListener());//监听窗体事件
        this.setVisible(true);
    }

    //工厂方法，返回窗体实例
    public static SubFrame getFrame()
    {
        //如果窗体对象为空，则创建窗体，否则直接返回已有窗体
        if(frame==null)
        {
            frame=new SubFrame();
        }
        return frame;
    }

    //事件监听器
```

```
    class MyIFListener extends InternalFrameAdapter
    {
        //子窗体关闭时，将窗体对象设为 null
        public void internalFrameClosing(InternalFrameEvent e)
        {
            if(frame!=null)
            {
                frame=null;
            }
        }
    }
}

//客户端测试类
class MainClass extends JFrame
{
    private JButton button;
    private JDesktopPane desktopPane;
    private SubFrame iFrame=null;

    public MainClass()
    {
        super("主窗体");
        Container c=this.getContentPane();
        c.setLayout(new BorderLayout());

        button=new JButton("点击创建一个内部窗体");
        button.addActionListener(new BtListener());
        c.add(button, BorderLayout.SOUTH);

        desktopPane = new JDesktopPane(); //创建 DesktopPane
        c.add(desktopPane);

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);
        this.setSize(400,400);
        this.show();
    }

    //事件监听器
    class BtListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
```

```
                if(iFrame!=null)
                {
                        desktopPane.remove(iFrame);
                }
                iFrame=SubFrame.getFrame();
                desktopPane.add(iFrame);
            }
        }

        public static void main(String[] args)
        {
            new MainClass();
        }
}
```

其中，SubFrame 类是 JInternalFrame 类的子类，在 SubFrame 类中定义了一个静态的 SubFrame 类型的实例变量，在静态工厂方法 getFrame()中创建了 SubFrame 对象并将其返回。在 MainClass 类中使用了该单例类，确保子窗口在当前应用程序中只有唯一一个实例，即只能弹出一个子窗口。

8. 参考类图如下所示：



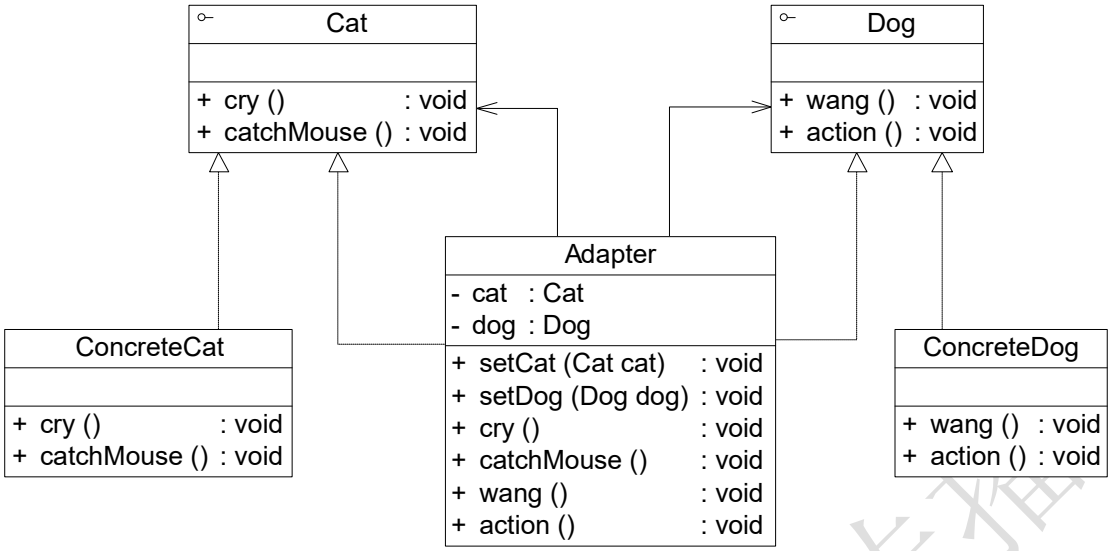
多例模式(Multiton Pattern)是单例模式的一种扩展形式，多例类可以有多个实例，而且必须自行创建和管理实例，并向外界提供自己的实例，可以通过静态集合对象来存储这些实例。多例类 Multiton 的代码如下所示：

```
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);
    }
}
```

第 9 章 适配器模式

1. B
2. C
3. A
4. A
5. 在对象适配器中，适配器与适配者之间是关联关系，一个适配器能够对应多个适配者类，只需要在该适配器类中定义对多个适配者对象的引用即可；在类适配器中，适配器与适配者是继承关系，一个适配器能否适配多个适配者类取决于该编程语言是否支持多重类继承，例如 C++语言支持多重类继承则可以适配多个适配者，而 Java、C#等语言不支持多重类继承则不能适配多个适配者。
6. 参考类图如下所示：

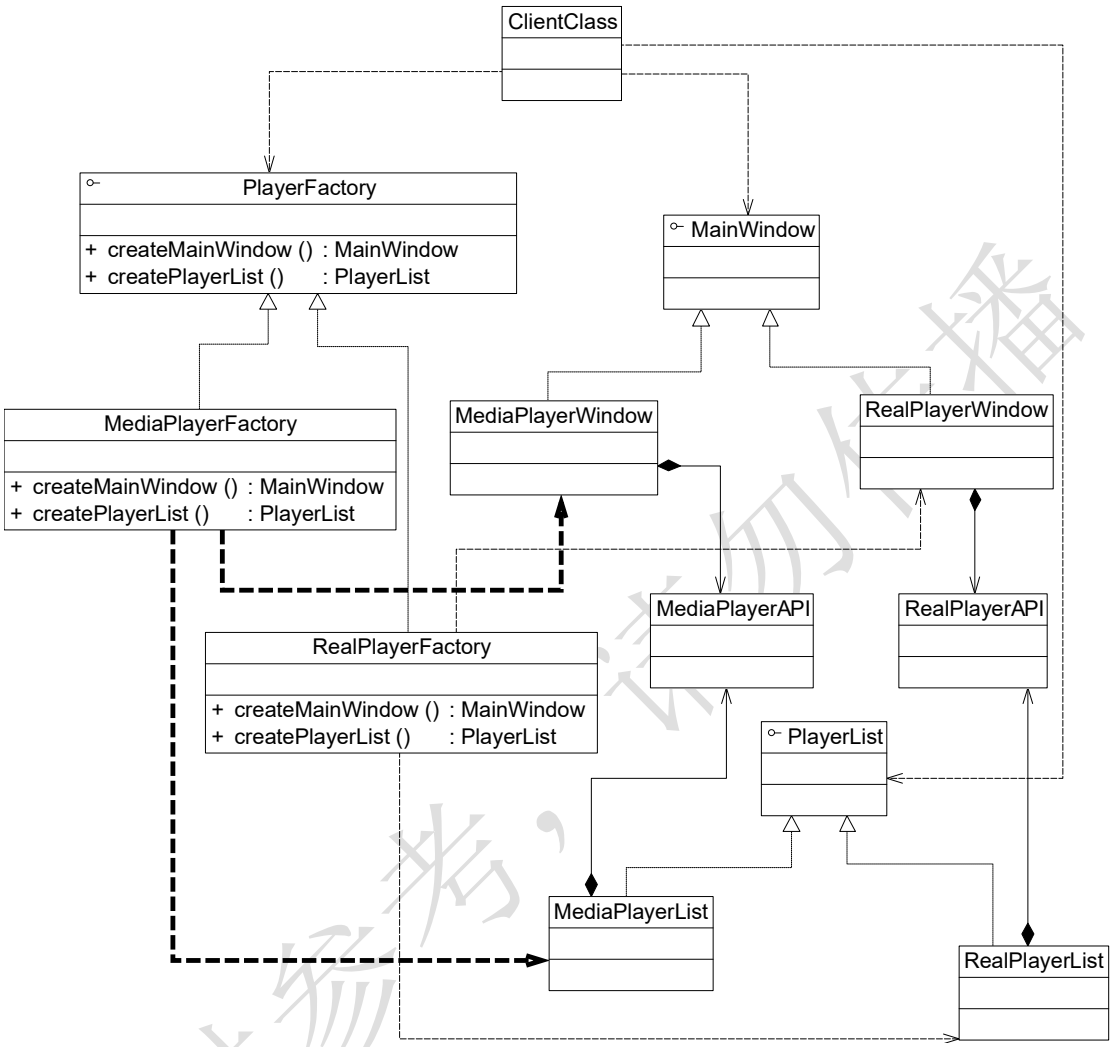


其中，Adapter 充当适配器，Cat 和 Dog 既充当抽象目标，又充当抽象适配者。如果客户端针对 Cat 编程，则 Cat 充当抽象目标，Dog 充当抽象适配者，ConcreteDog 充当具体适配者；如果客户端针对 Dog 编程，则 Dog 充当抽象目标，Cat 充当抽象适配者，ConcreteCat 充当具体适配者。在此使用对象适配器，Adapter 类的代码如下所示：

```
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
        {
```

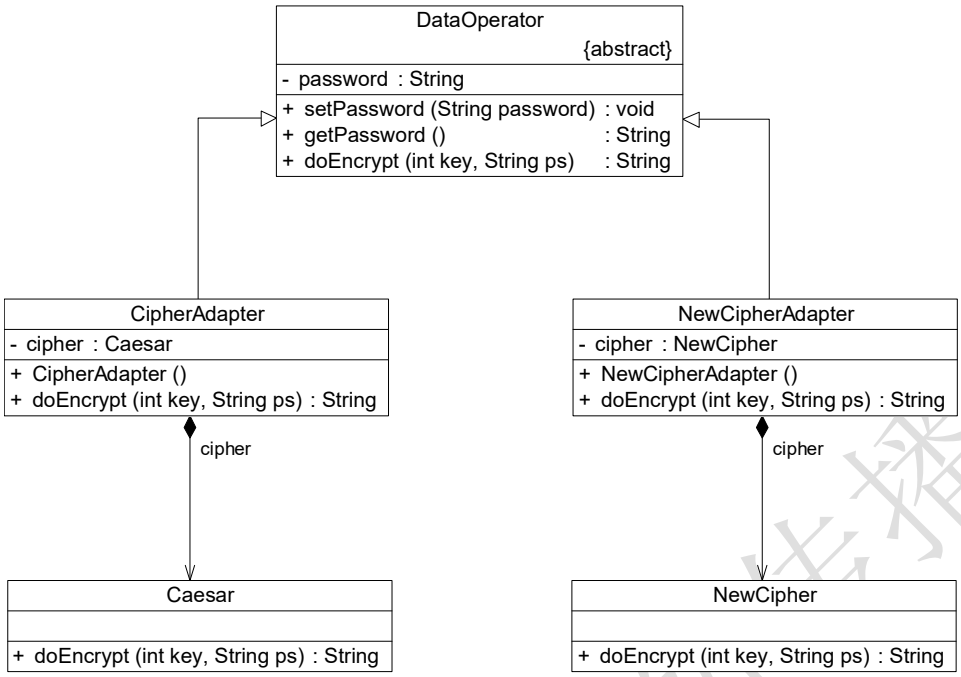
```
                cat.catchMouse();
            }
}
```

7. 本题可使用适配器模式和抽象工厂模式，参考类图如下所示：



在该类图中，我们为两种不同的播放器提供了两个具体工厂类 MediaPlayerFactory 和 RealPlayerFactory，其中 MediaPlayerFactory 作为 Windows Media Player 播放器工厂，可以创建 Windows Media Player 的主窗口(MediaPlayerWindow)和播放列表(MediaPlayerList)（为了简化类图，只列出主窗口和播放列表这两个播放器组成元素，实际情况下包含更多组成元素）；RealPlayerFactory 作为 RealPlayer 播放器工厂，创建 RealPlayer 的主窗口(RealPlayerWindow)和播放列表(RealPlayerList)，此时可以使用抽象工厂模式，客户端针对抽象工厂 PlayerFactory 编程，如果增加新的播放器，只需增加一个新的具体工厂来生产新产品族中的产品即可。由于需要调用现有 API 中的方法，因此还需要使用适配器模式，在具体产品类如 MediaPlayerWindow 和 MediaPlayerList 调用 Windows Media Player API 中的方法，在 RealPlayerWindow 和 RealPlayerList 中调用 RealPlayer API 中的方法，实现对 API 中方法的适配，此时具体产品如 MediaPlayerWindow、RealPlayerWindow 等充当适配器，而已有的 API 如 MediaPlayerAPI 和 RealPlayerAPI 是需要适配的适配者。

8. 参考类图（对象适配器）如下所示：



其中，DataOperator 充当目标抽象类角色，CipherAdapter 和 NewCipherAdapter 充当适配器角色，Caesar 和 NewCipher 充当适配者角色。

类适配器设计方案只需将上图中适配器与适配者之间的关联关系改为继承关系即可。

第 10 章 桥接模式

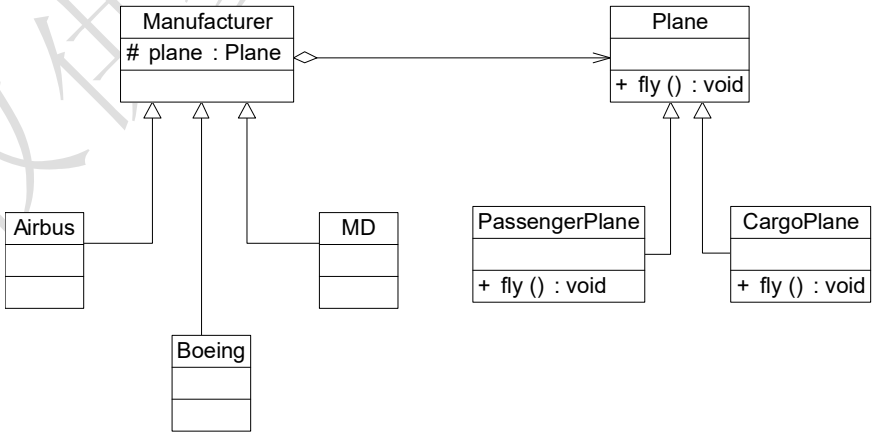
1. B D

2. C

3. C

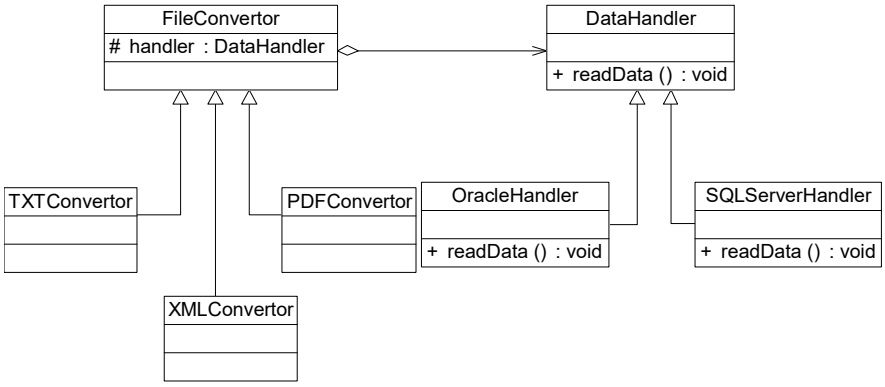
4. 桥接模式可以处理存在多个独立变化维度的系统，每一个独立维度对应一个继承结构，其中一个为“抽象类”层次结构，其他为“实现类”层次结构，“抽象类”层次结构中的抽象类与“实现类”层次结构中的接口之间存在抽象耦合关系。

5. 参考类图如下所示：



其中，Manufacturer 充当抽象类角色，Airbus、Boeing 和 MD 充当扩充抽象类角色，Plane 充当实现类接口角色，PassengerPlane 和 CargoPlane 充当具体实现类角色。

6. 参考类图如下所示：



其中，FileConvertor 充当抽象类角色，TXTConvertor、XMLConvertor 和 PDFConvertor 充当扩充抽象类角色，DataHandler 充当实现类接口角色，OracleHandler 和 SQLServerHandler 充当具体实现类角色。

第 11 章 组合模式

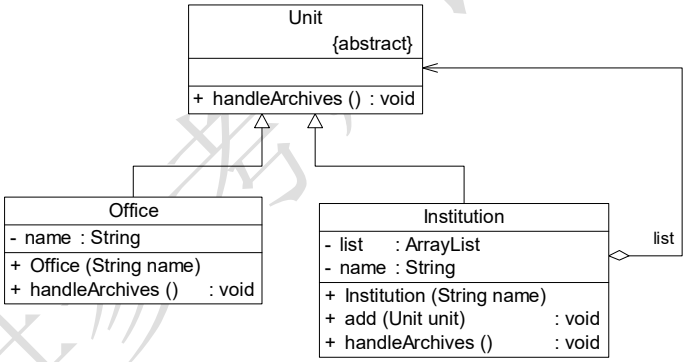
1. B

2. B

3. C

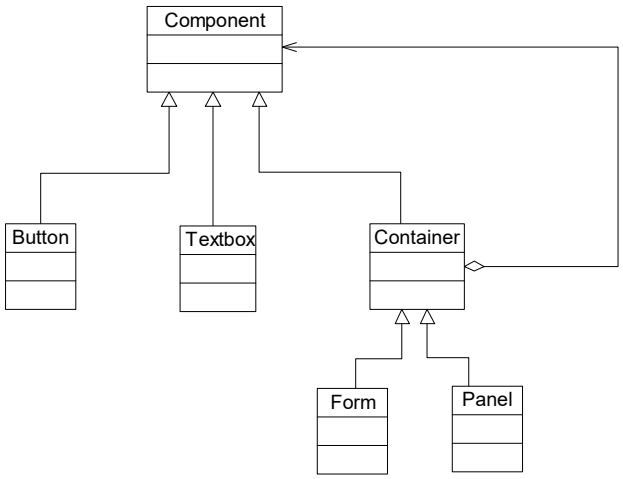
4. 结果：容器(Composite)对象中只能包含叶子(Leaf)对象，不能再继续包含容器对象，导致无法递归构造出一个多层树形结构。

5. 参考类图如下所示：



本题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。

6. 参考类图如下所示：



其中，Component 充当抽象构件角色，Button 和 Textbox 充当叶子构件角色，Container 充当抽象容器构件角色，Form 和 Panel 充当具体容器构件角色。

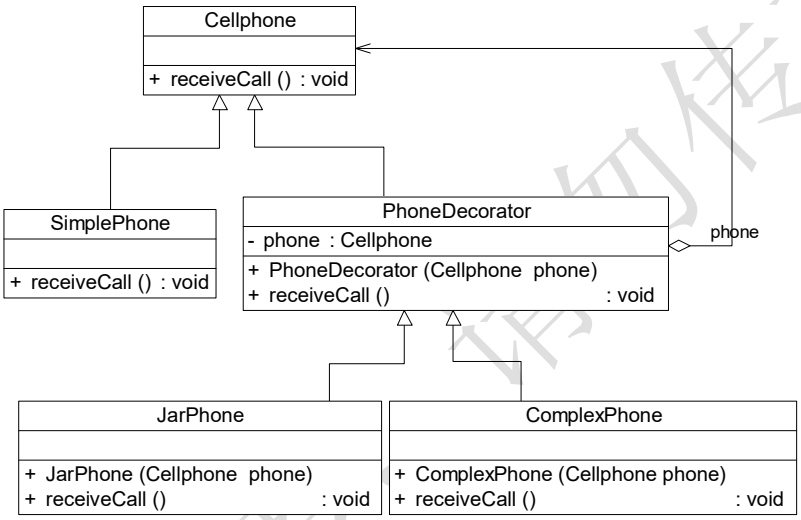
第 12 章 装饰模式

1. D

2. C

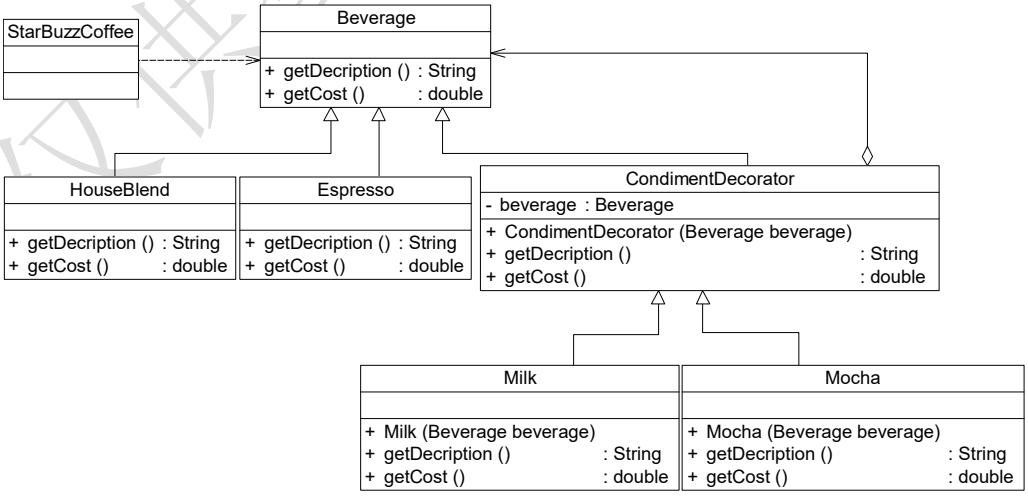
3. 不能实现对用一个对象的多次装饰。因为在半透明装饰模式中，使用具体装饰类来声明装饰之后的对象，具体装饰类中新增的方法并未在抽象构件类中声明，这样做的优点在于装饰后客户端可以单独调用在具体装饰类中新增的业务方法，但是将导致无法调用到之前装饰时新增的方法，只能调用到最后一次装饰时具体装饰类中新增加的方法，故对同一个对象实施多次装饰没有任何意义。

4. 参考类图如下所示：



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。

5. 参考类图如下所示：



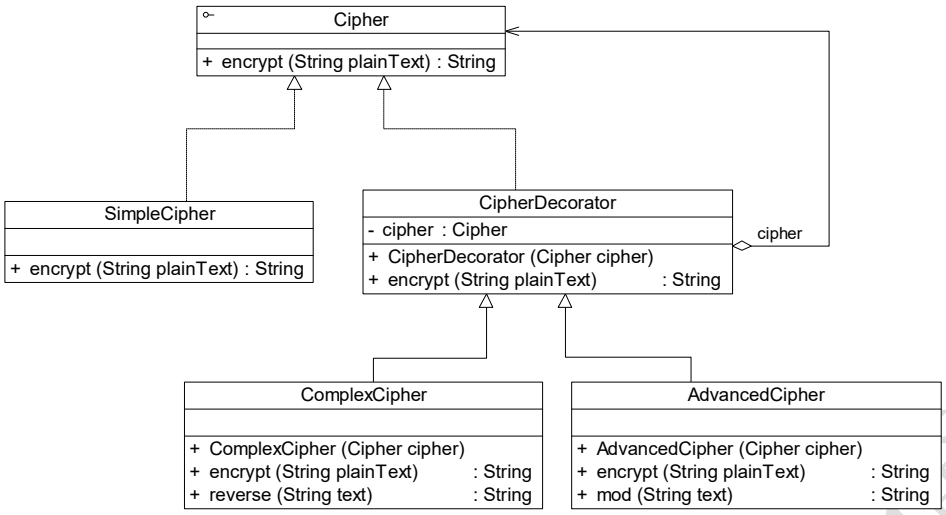
其中，Beverage 充当抽象组件，HouseBlend 和 Espresso 充当具体组件，CondimentDecorator 充当抽象装饰器，Milk 和 Mocha 充当具体装饰器，StarBuzzCoffee 充当客户端。本题完整代码示例如下所示：

```
abstract class Beverage       //抽象组件
{
      public abstract String getDescription();
      public abstract double getCost();
}

class HouseBlend extends Beverage       //具体组件
{
      public String getDescription()
      {
            return "HouseBlend 咖啡";
      }
      public double getCost()
      {
            return 10.00;
      }
}

class Espresso extends Beverage       //具体组件
{
      public String getDescription()
      {
            return "Espresso 咖啡";
      }
      public double getCost()
      {
            return 20.00;
      }
}

class CondimentDecorator extends Beverage       //抽象装饰器
{
      private Beverage beverage;
      public CondimentDecorator(Beverage beverage)
      {
            this.beverage = beverage;
      }
      public String getDescription()
      {
            return beverage.getDescription();
      }
      public double getCost()
      {
            return beverage.getCost();
```

```
    }
}

class Milk extends CondimentDecorator       //具体装饰器
{
    public Milk(Beverage beverage)
    {
        super(beverage);
    }
    public String getDescription()
    {
        String decription = super.getDescription();
        return decription + "加牛奶";
    }
    public double getCost()
    {
        double cost = super.getCost();
        return cost + 2.0;
    }
}

class Mocha extends CondimentDecorator       //具体装饰器
{
    public Mocha(Beverage beverage)
    {
        super(beverage);
    }
    public String getDescription()
    {
        String decription = super.getDescription();
        return decription + "加摩卡";
    }
    public double getCost()
    {
        double cost = super.getCost();
        return cost + 3.0;
    }
}

class StarBuzzCoffee       //客户端测试类
{
    public static void main(String args[])
    {
        String decription;
```

```
        double cost;
        Beverage beverage_e;

        beverage_e = new Espresso();
        decription = beverage_e.getDescription();
        cost = beverage_e.getCost();
        System.out.println("饮料：" + decription);
        System.out.println("价格：" + cost);
        System.out.println("----------------------");

        Beverage beverage_mi;
        beverage_mi = new Milk(beverage_e);
        decription = beverage_mi.getDescription();
        cost = beverage_mi.getCost();
        System.out.println("饮料：" + decription);
        System.out.println("价格：" + cost);
        System.out.println("----------------------");

        Beverage beverage_mo;
        beverage_mo = new Mocha(beverage_mi);
        decription = beverage_mo.getDescription();
        cost = beverage_mo.getCost();
        System.out.println("饮料：" + decription);
        System.out.println("价格：" + cost);
        System.out.println("----------------------");
    }
}
//输出结果如下：
//饮料：Espresso 咖啡
//价格：20.0
//----------------------
//饮料：Espresso 咖啡加牛奶
//价格：22.0
//----------------------
//饮料：Espresso 咖啡加牛奶加摩卡
//价格：25.0
```

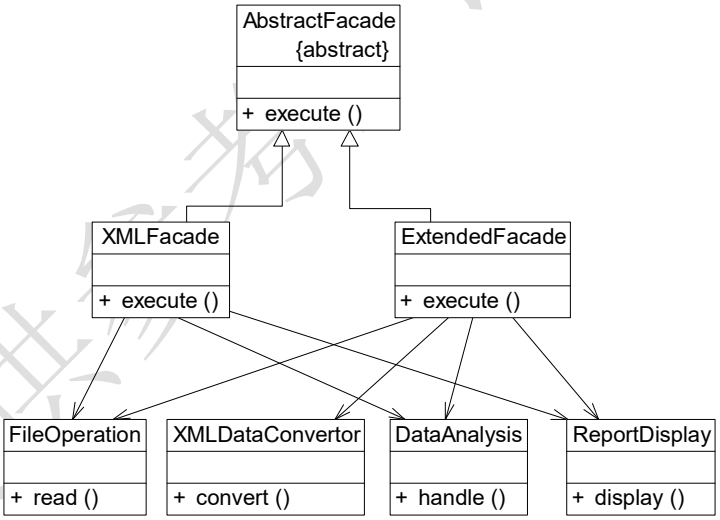
6. 参考类图如下所示：



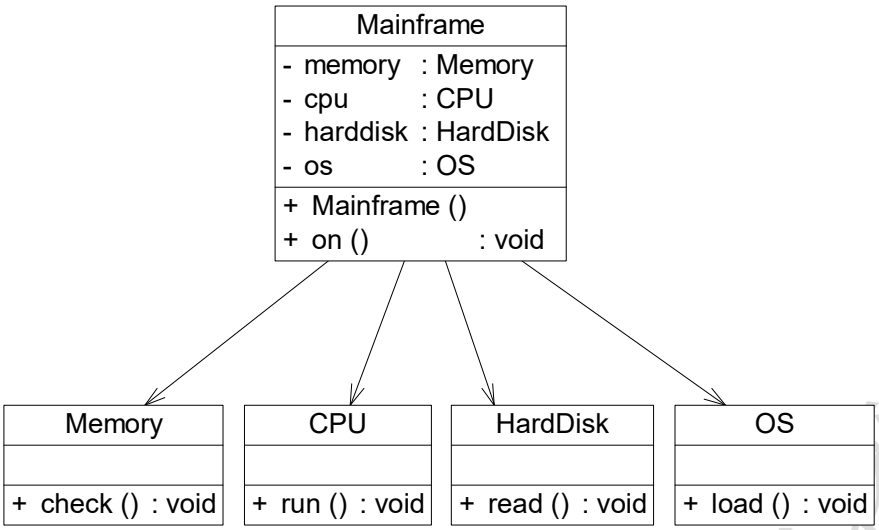
其中，Cipher 充当抽象构件角色，SimpleCipher 充当具体构件角色，CipherDecorator 充当抽象装饰类角色，ComplexCipher 和 AdvancedCipher 充当具体装饰类角色。

第 13 章 外观模式

1. A
2. D
3. C
4. 参考类图如下所示：

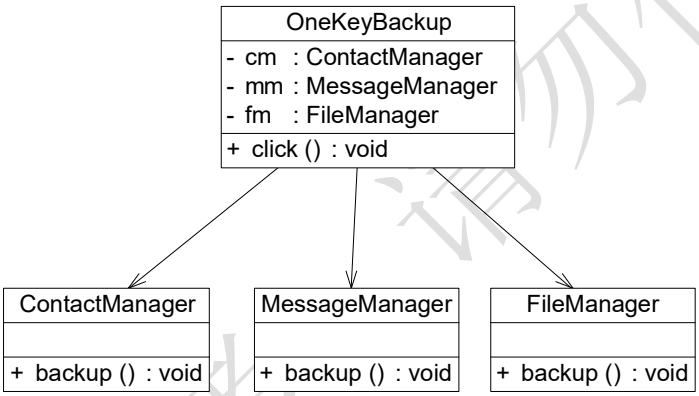


5. 参考类图如下所示：



其中，Mainframe 充当外观角色，Memory、CPU、HardDisk 和 OS 充当子系统角色。

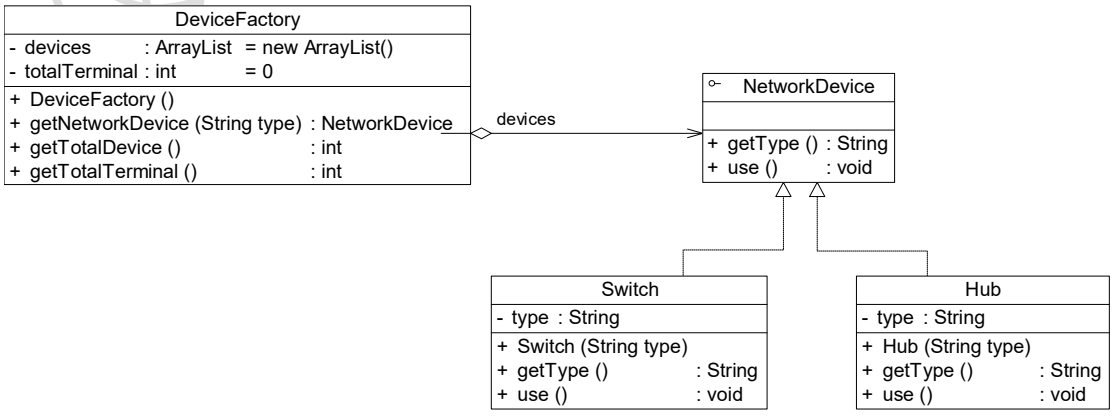
6. 参考类图如下所示：



其中，OneKeyBackup 充当外观角色，ContactManager、MessageManager 和 FileManager 充当子系统角色。

第 14 章 享元模式

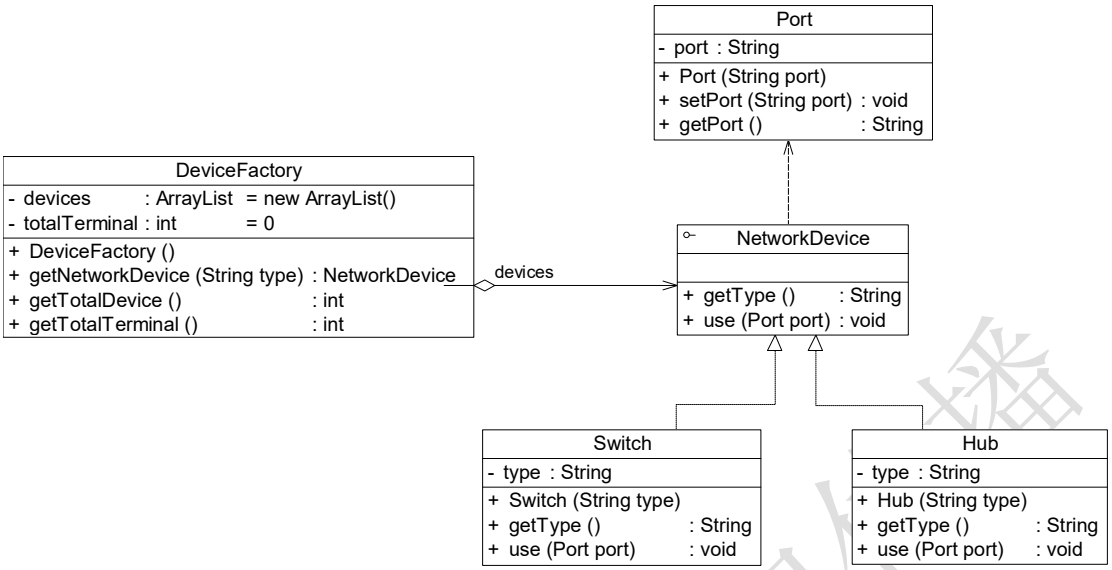
1. C
2. D
3. C
4. 参考类图如下所示（无外部状态）：



其中，DeviceFactory 充当享元工厂角色，NetworkDevice 充当抽象享元角色，Switch 和

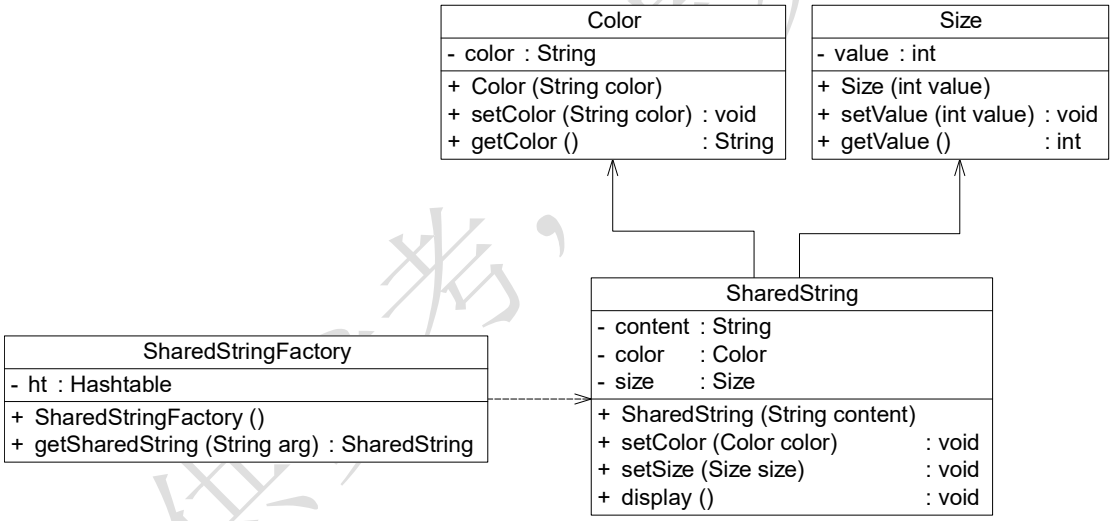
Hub 充当具体享元角色。

有外部状态的参考类图如下所示：



其中，Port（端口号）类充当外部状态类。

5. 参考类图如下所示：



在类图中省略了抽象享元角色，SharedString 充当具体享元角色，SharedStringFactory 充当享元工厂角色，Color 和 Size 充当外部状态类。本实例代码如下所示：

```
import java.util.*;
class Color
{
    private String color;
    public Color(String color)
    {
        this.color = color;
    }
    public void setColor(String color)
    {
        this.color = color;
```

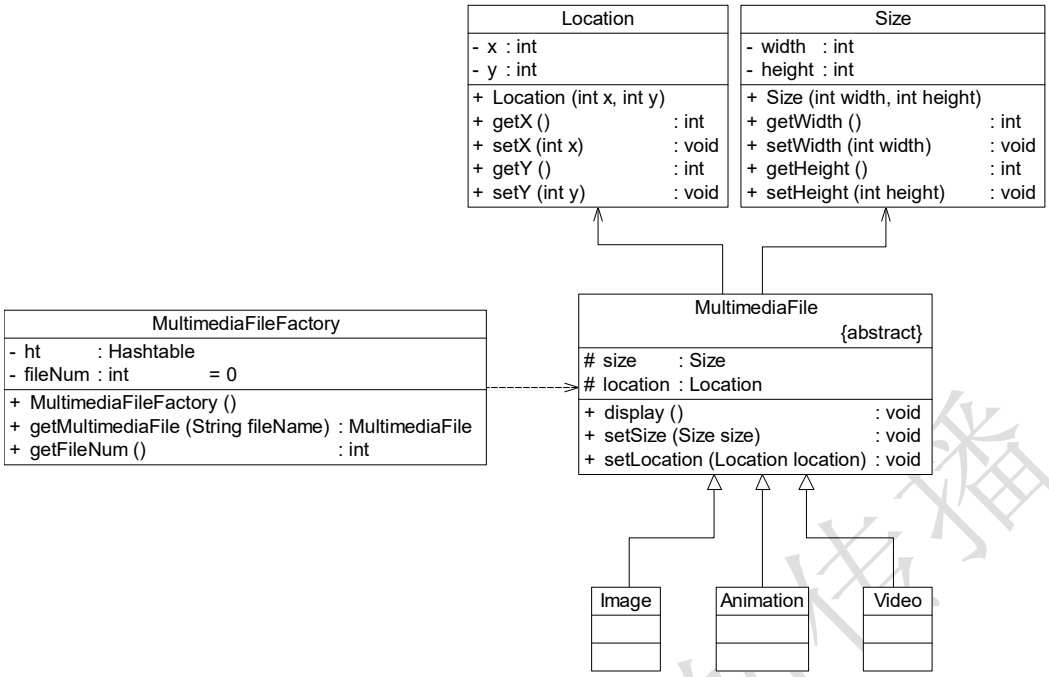
```
    }
    public String getColor()
    {
        return this.color;
    }
}

class Size
{
    private int value;
    public Size(int value)
    {
        this.value = value;
    }
    public void setValue(int value)
    {
        this.value = value;
    }
    public int getValue()
    {
        return this.value;
    }
}

class SharedString
{
    private String content;
    private Color color;
    private Size size;
    public SharedString(String content)
    {
        this.content = content;
    }
    public void setColor(Color color)
    {
        this.color = color;
    }
    public void setSize(Size size)
    {
        this.size = size;
    }
    public void display()
    {
        System.out.println("内容：" + this.content + "，颜色：" + this.color.getColor() + "，
```

```
大小：" + this.size.getValue());
    }
}

class SharedStringFactory
{
    private Hashtable ht;
    public SharedStringFactory()
    {
        ht = new Hashtable();
    }
    public SharedString getSharedString(String arg)
    {
        if(ht.containsKey(arg))
        {
            return (SharedString)ht.get(arg);
        }
        else
        {
            SharedString str = new SharedString(arg);
            ht.put(arg,str);
            return (SharedString)ht.get(arg);
        }
    }
}
```

在客户类中，如果需要显示两个颜色和大小不同的字符串“Java”，代码如下所示：

```
class Client
{
    public static void main(String args[])
    {
        SharedString str1,str2;
        SharedStringFactory factory = new SharedStringFactory();
        str1 = factory.getSharedString("Java");
        str1.setColor(new Color("红色"));
        str1.setSize(new Size(5));
        str1.display();   //输出“内容：Java，颜色：红色，大小：5”
        str2 = factory.getSharedString("Java ");
        str2.setColor(new Color("黑色"));
        str2.setSize(new Size(10));
        str2.display(); //输出“内容：Java，颜色：黑色，大小：10”
        System.out.println(str1==str2);   //输出“true”
    }
}
```

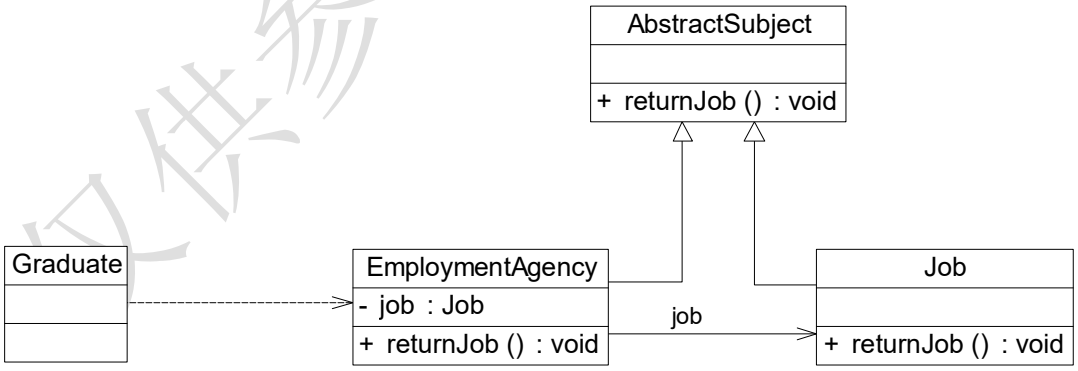
6. 参考类图如下所示：



其中，MultimediaFile 表示抽象享元，其子类 Image、Animation 和 Video 表示具体享元。对于相同的多媒体文件，其大小 Size 和位置 Location 可以不同，因此需要通过 Setter 方法来设置这些外部状态。MultimediaFileFactory 是享元工厂，在其中定义一个 Hashtable 对象作为享元池，存储和维护享元对象。

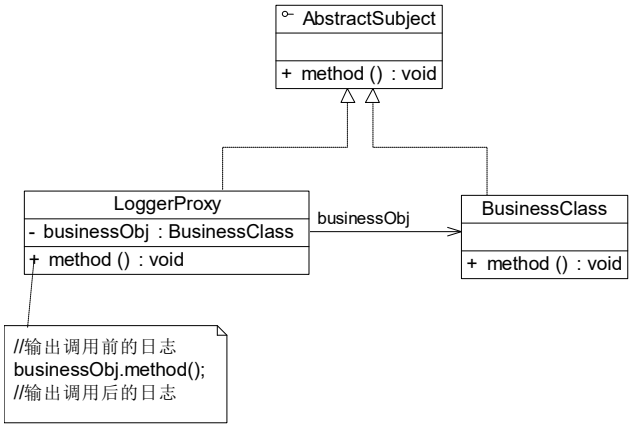
第 15 章 代理模式

1. A
2. B
3. D
4. 代理模式。参考类图如下所示：



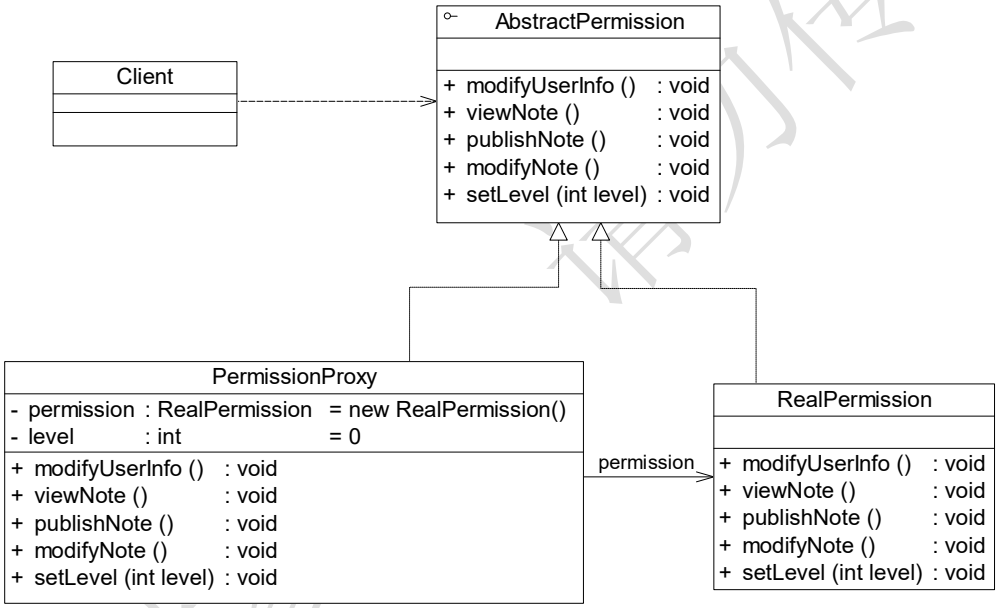
其中，AbstractSubject 充当抽象主题角色，Job（工作类）充当真实主题角色，EmploymentAgency（职业介绍所）充当代理主题角色， Graduate（毕业生）充当客户类。

5. 参考类图如下所示：



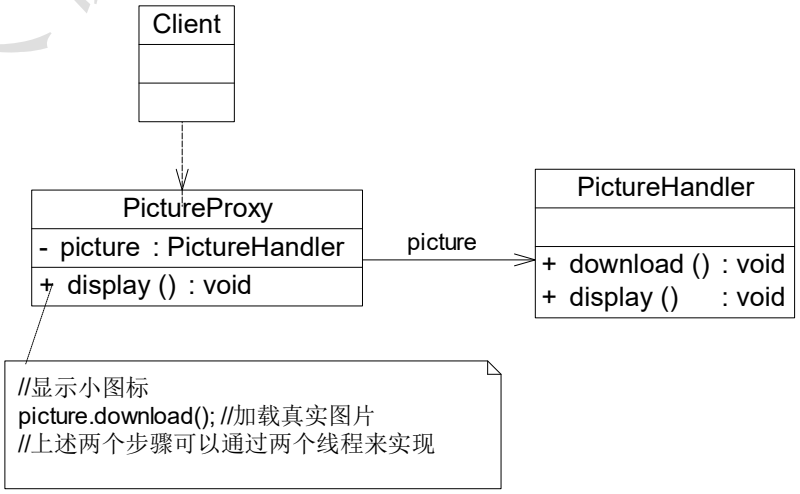
其中，AbstractSubject 充当抽象主题角色，BusinessClass 充当真实主题角色，LoggerProxy 充当代理主题角色。

6. 参考类图如下所示：



其中，AbstractPermission 为抽象主题角色，PermissionProxy 为代理主题角色，RealPermission 为真实主题角色。

7. 参考类图如下所示：



该类图是代理模式的一个变形，没有包含抽象主题角色，PictureHandler 充当真实主题角色，PictureProxy 充当代理主题角色。

8. 本题答案略【请自学 RMI 相关知识】。

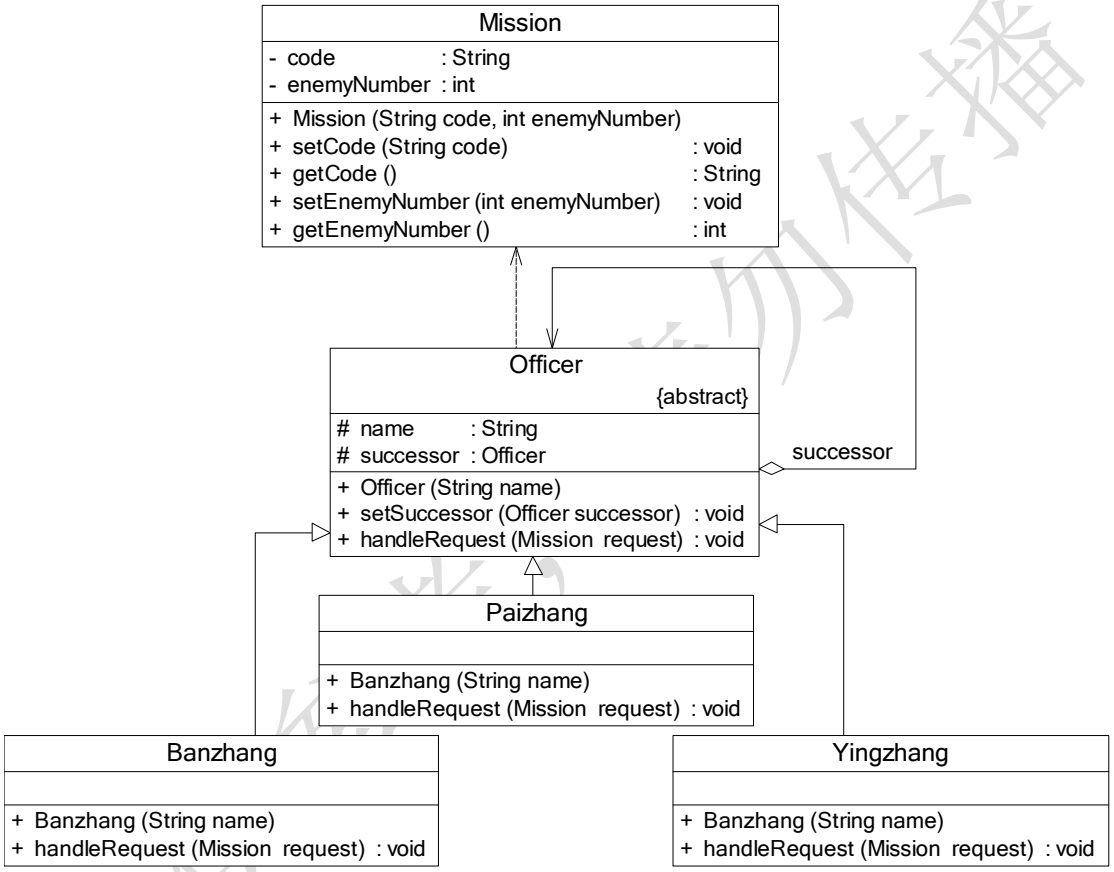
第 16 章 职责链模式

1. B

2. A

3. 异常处理使用的是不纯的职责链模式，存在一个 try 语句中的异常最终没有被任何 catch 语句处理的情况，即一个请求可能最终不被任何处理者对象接收并处理。

4. 参考类图如下所示：



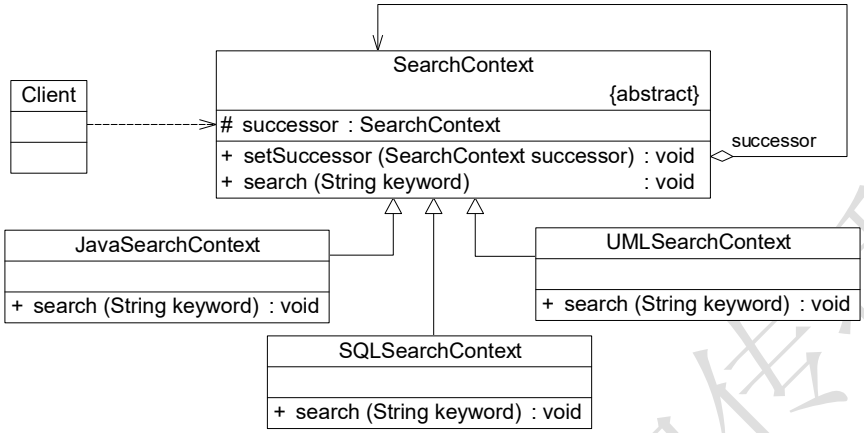
Mission 充当请求角色，Officer 充当抽象传递者角色，Banzhang、Paizhang 和 Yingzhang 充当具体传递者角色。其中，Officer 类、Banzhang 类和 Yingzhang 类的代码如下所示（其他类代码省略）：

```
abstract class Officer
{
    protected String name;
    protected Officer successor;
    public Officer(String name)
    {
        this.name=name;
    }
    public void setSuccessor(Officer successor)
    {
```

```
        this.successor=successor;
    }
    public abstract void handleRequest(Mission request);
}

class Banzhang extends Officer
{
    public Banzhang(String name)
    {
        super(name);
    }
    public void handleRequest(Mission request)
    {
        if(request.getEnemyNumber()<10)
        {
            System.out.println("班长" + name + "下达代号为" + request.getCode() + "的作战任务，敌人数量为" + request.getEnemyNumber());
        }
        else
        {
            if(this.successor!=null)
            {
                this.successor.handleRequest(request);
            }
        }
    }
}

class Yingzhang extends Officer
{
    public Yingzhang(String name)
    {
        super(name);
    }
    public void handleRequest(Mission request)
    {
        if(request.getEnemyNumber()<200)
        {
            System.out.println("营长" + name + "下达代号为" + request.getCode() + "的作战任务，敌人数量为" + request.getEnemyNumber());
        }
        else
        {
            System.out.println("开会讨论代号为" + request.getCode() + "的作战任务，敌
```

```
人数量为" + request.getEnemyNumber());
            }
        }
}
```

5. 参考类图如下所示：



其中，SearchContext 充当抽象处理者（抽象传递者），JavaSearchContext、SQLSearchContext 和 UMLSearchContext 充当具体处理者（具体传递者）。

代码如下所示：

```
//抽象查询请求处理上下文类：抽象传递者
abstract class SearchContext
{
    protected SearchContext successor;
    public void setSuccessor(SearchContext successor)
    {
        this.successor = successor;
    }
    public abstract void search(String keyword);
}

//具体查询请求处理上下文类：具体传递者
class JavaSearchContext extends SearchContext
{
    public void search(String keyword)
    {
        //模拟实现
        if(keyword.contains("Java"))
        {
            System.out.println("查询关键字 Java！");
        }
        else
        {
            successor.search(keyword);
        }
```

```
    }
}

//具体查询请求处理上下文类：具体传递者
class SQLSearchContext extends SearchContext
{
    public void search(String keyword)
    {
        //模拟实现
        if(keyword.contains("SQL"))
        {
            System.out.println("查询关键字 SQL！");
        }
        else
        {
            successor.search(keyword);
        }
    }
}

//具体查询请求处理上下文类：具体传递者
class UMLSearchContext extends SearchContext
{
    public void search(String keyword)
    {
        //模拟实现
        if(keyword.contains("UML"))
        {
            System.out.println("查询关键字 UML！");
        }
        else
        {
            successor.search(keyword);
        }
    }
}
```

客户端测试代码如下所示：

```
//客户端测试类
class Client
{
    public static void main(String args[])
    {
        SearchContext jContext,sContext,uContext;
        jContext = new JavaSearchContext();
```

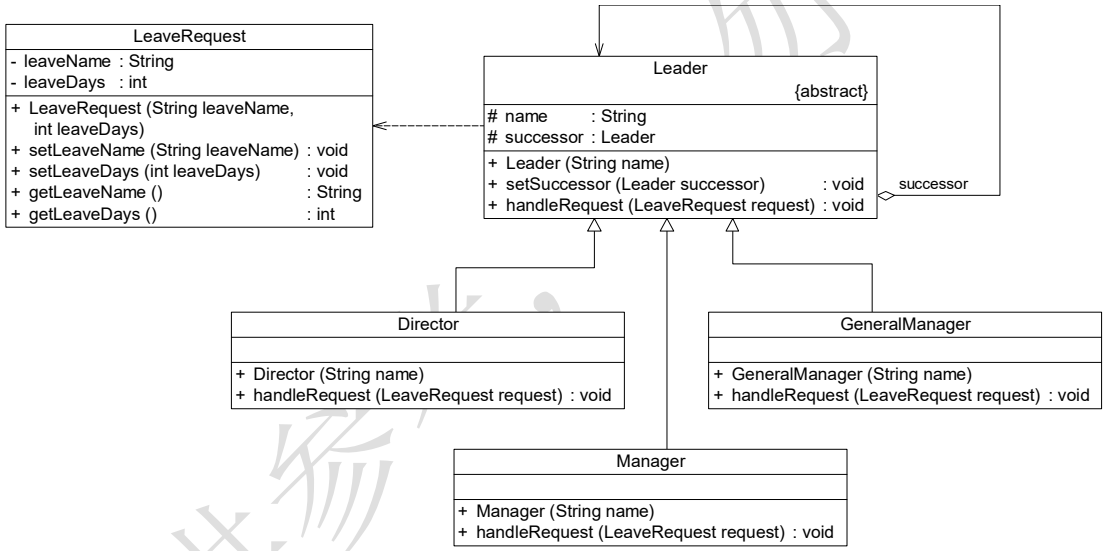
```
            sContext = new SQLSearchContext();
            uContext = new UMLSearchContext();
            jContext.setSuccessor(sContext);
            sContext.setSuccessor(uContext);
            String keyword = "UML 类图绘制疑惑";
            jContext.search(keyword);
        }
}
```

运行结果如下：

```
查询关键字 UML！
```

在本实例中，在客户端测试类中创建了职责链，当向 jContext 对象传递查询关键字"UML 类图绘制疑惑"时，jContext 首先处理该关键字，如果不能处理则转发请求给下家，直到链上某一个对象能够处理该关键字请求，对于该关键字，请求转发顺序为 jContext→ sContext→ uContext，最后由 uContext 对象处理该请求。职责链由客户端创建，因此请求的传递顺序也由客户端来确定。

6. 参考类图如下所示：



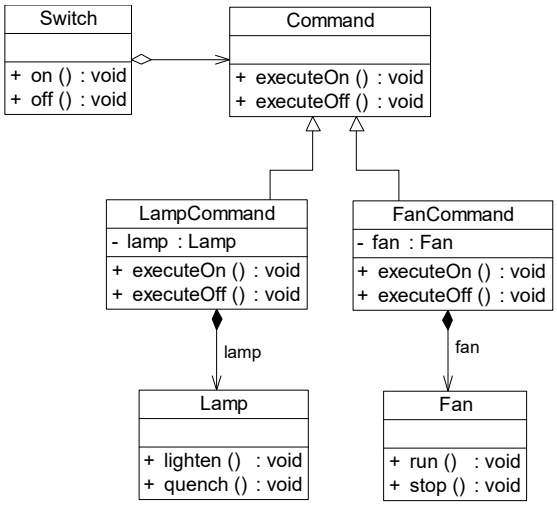
其中，Leader 充当抽象处理者角色，Director、Manager 和 GeneralManager 充当具体处理者角色，LeaveRequest 是请求类。

第 17 章 命令模式

1. D

2. C

3. 参考类图如下所示：

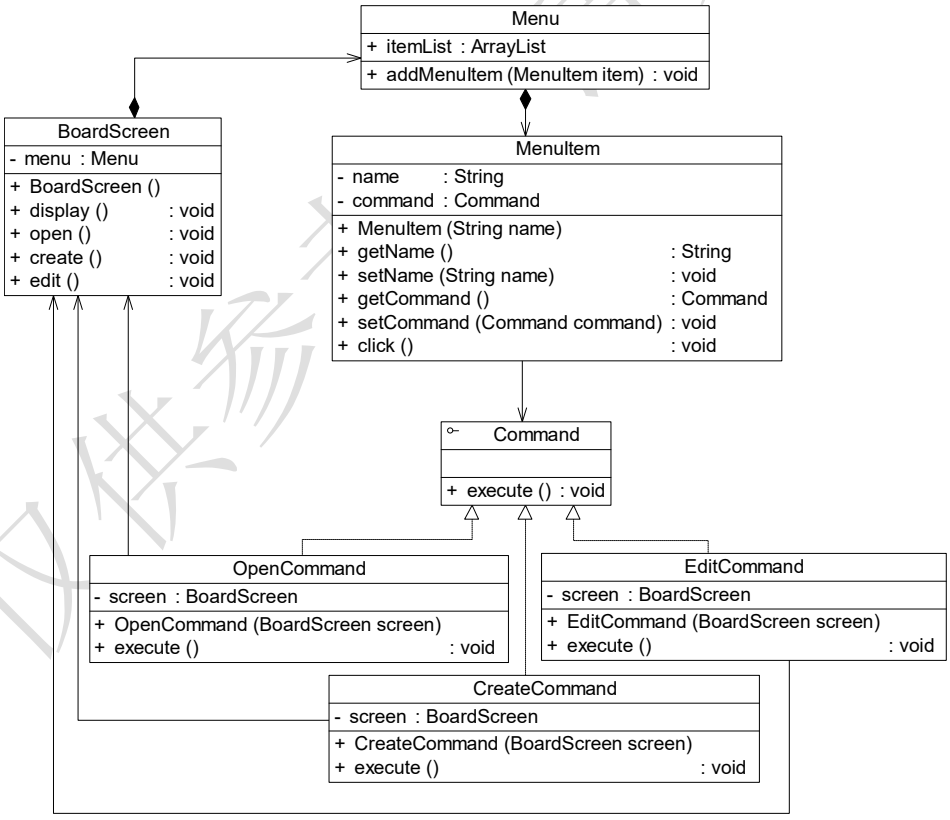


其中，Switch 充当调用者（发送者）角色，Command 是抽象命令类，LampCommand 和 FanCommand 充当具体命令角色，Lamp 和 Fan 充当接收者角色。

4. 解答思路：可以引入栈来实现 Undo 和 Redo 操作，将命令对象存储在两个栈中，一个栈用于实现 Undo，一个用于实现 Redo。

5. 解答思路：使用序列化机制将多个命令对象写入文件，形成日志文件；在执行批处理时，读取该日志文件，还原命令对象，再逐个调用命令对象的执行方法实现批处理。

6. 参考类图如下所示：



其中，BoardScreen 充当接收者角色，MenuItem 充当调用者角色，Command 充当抽象命令角色，OpenCommand、CreateCommand 和 EditCommand 充当具体命令角色。本实例代码如下：

```
import java.util.*;
```

```
//抽象命令
interface Command
{
	public void execute();
}

//菜单项类：请求发送者（调用者）
class MenuItem
{
	private String name;
	private Command command;
	public MenuItem(String name)
	{
		this.name = name;
	}
	public String getName()
	{
		return this.name;
	}
	public void setName(String name)
	{
		this.name = name;
	}
	public Command getCommand()
	{
		return this.command;
	}
	public void setCommand(Command command)
	{
		this.command = command;
	}
	public void click()
	{
		command.execute();
	}
}

//菜单类
class Menu
{
	public ArrayList itemList = new ArrayList();
	public void addMenuItem(MenuItem item)
	{
		itemList.add(item);
```

```
    }
}

//打开命令：具体命令
class OpenCommand implements Command
{
    private BoardScreen screen;
    public OpenCommand(BoardScreen screen)
    {
        this.screen = screen;
    }
    public void execute()
    {
        screen.open();
    }
}

//新建命令：具体命令
class CreateCommand implements Command
{
    private BoardScreen screen;
    public CreateCommand(BoardScreen screen)
    {
        this.screen = screen;
    }
    public void execute()
    {
        screen.create();
    }
}

//编辑命令：具体命令
class EditCommand implements Command
{
    private BoardScreen screen;
    public EditCommand(BoardScreen screen)
    {
        this.screen = screen;
    }
    public void execute()
    {
        screen.edit();
    }
}
```

```
//公告板系统界面：接收者
class BoardScreen
{
	private Menu menu;
	private MenuItem openItem,createItem,editItem;
	public BoardScreen()
	{
		menu = new Menu();
		openItem = new MenuItem("打开");
		createItem = new MenuItem("新建");
		editItem = new MenuItem("编辑");
		menu.addMenuItem(openItem);
		menu.addMenuItem(createItem);
		menu.addMenuItem(editItem);
	}
	public void display()
	{
		System.out.println("主菜单选项：");
		for(Object obj:menu.itemList)
		{
			System.out.println(((MenuItem)obj).getName());
		}
	}
	public void open()
	{
		System.out.println("显示打开窗口！");
	}
	public void create()
	{
		System.out.println("显示新建窗口！");
	}
	public void edit()
	{
		System.out.println("显示编辑窗口！");
	}
	public Menu getMenu()
	{
		return menu;
	}
}
```

客户端测试代码如下所示：

```
//客户端测试类
class Client
```

```
{
    public static void main(String args[])
    {
        BoardScreen screen = new BoardScreen();   //接收者
        Menu menu = screen.getMenu();
        Command openCommand,createCommand,editCommand;   //命令
        openCommand = new OpenCommand(screen);
        createCommand = new CreateCommand(screen);
        editCommand = new EditCommand(screen);
        MenuItem openItem,createItem,editItem;   //调用者
        openItem = (MenuItem)menu.itemList.get(0);
        createItem = (MenuItem)menu.itemList.get(1);
        editItem = (MenuItem)menu.itemList.get(2);
        openItem.setCommand(openCommand);
        createItem.setCommand(createCommand);
        editItem.setCommand(editCommand);
        screen.display();
        openItem.click();
        createItem.click();
        editItem.click();
    }
}
```

运行结果如下：

```
主菜单选项：
打开
新建
编辑
显示打开窗口！
显示新建窗口！
显示编辑窗口！
```

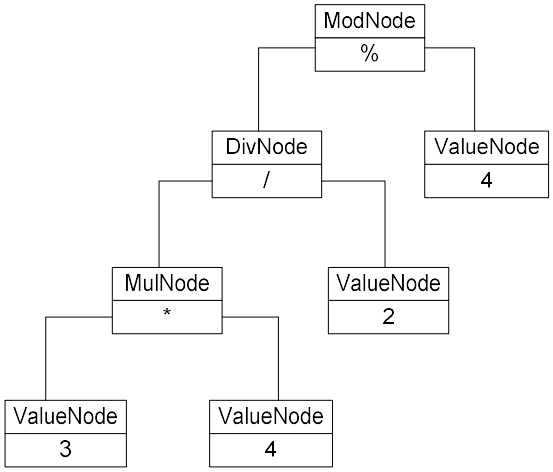
在本实例中，只需要在调用者 MenuItem 中注入不同的具体命令类，可以使得相同的菜单项 MenuItem 对应接收者 BoardScreen 的不同方法。无须修改类库代码，只需修改客户端代码即可更换接收者。在实际开发时，可以将 BoardScreen 中的 open()、create()和 edit()等方法封装在不同的类中，如果需要更换某菜单项的功能，只需对应增加一个新的具体命令类和一个接收者类，再将新的具体命令对象注入对应的 MenuItem 对象，即可实现菜单项功能的改变，且符合开闭原则。

第 18 章 解释器模式

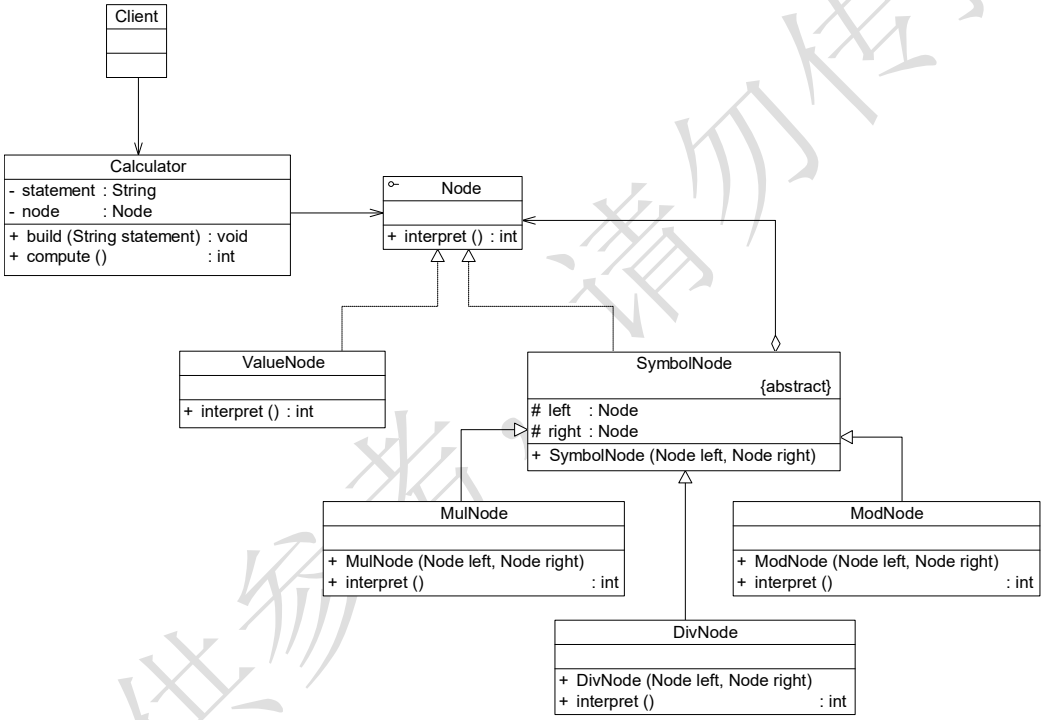
1. D

2. C

3. 构造的抽象语法树实例如下：



参考类图如下所示：



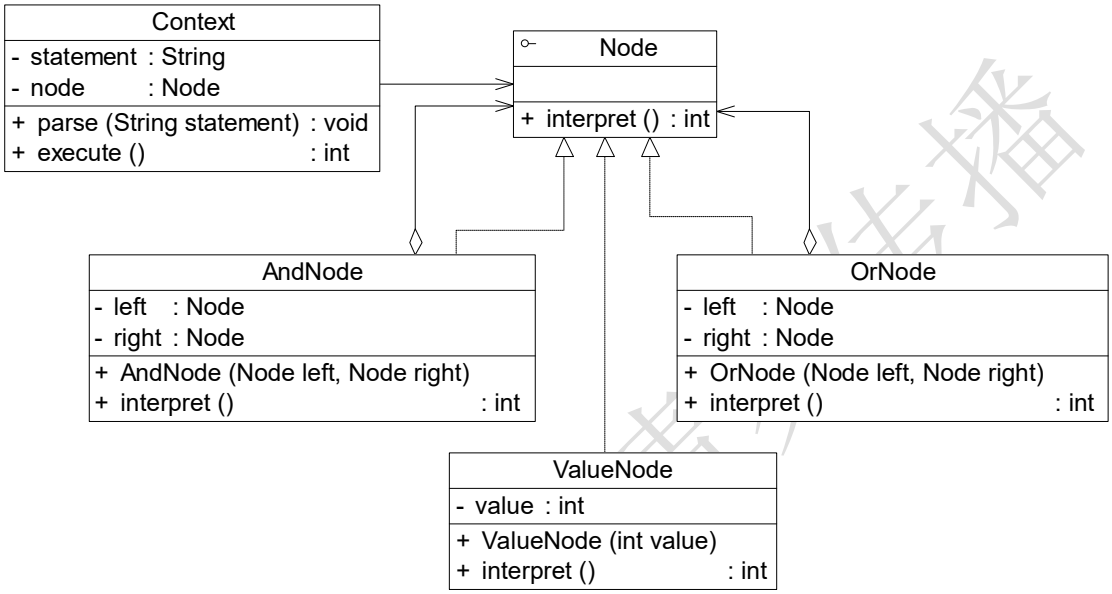
其中，Node 充当抽象表达式角色，ValueNode 是终结符表达式类，SymbolNode 是抽象非终结符表达式类，其子类 MulNode、DivNode 和 ModNode 是非终结符表达式类。Calculator 类是本实例的核心类之一，它的引入极大简化了客户类代码。Calculator 类是一个辅助类，在 Calculator 类中定义了如何构造一棵抽象语法树，在构造过程中使用了栈结构 Stack，注意加粗部分的代码，对字符串进行分割后如果判断子字符串既不是符号“*”，也不是符号“/”和“%”，则表示对应的子字符串为数字，实例化终结符表达式类 ValueNode，并通过栈的 push()方法将其压入栈中；如果判断子字符串为“*”，则将压入栈中的内容通过栈的 pop()方法取出作为其左表达式，而将之后输入的数字封装在 ValueNode 类型的对象中作为其右表达式，通过左表达式和右表达式创建非终结符表达式 MulNode 类型的对象，最后再将该表达式压入栈中。通过这一系列操作，放置在栈中的是一个完整的表达式，通过栈的 pop()方法将其取出，再在 compute()方法中调用该表达式的 interpret()方法，程序执行时将递归调用每一个子表达式的 interpret()方法，即执行每一个封装在终结符表达式类和非终结符表达式类中的 interpret()方法。

Calculator 类代码如下所示：

```
import java.util.*;

public class Calculator
{
	private String statement;
	private Node node;

	public void build(String statement)
	{
		Node left=null,right=null;
		Stack stack=new Stack();

		String[] statementArr=statement.split(" ");

		for(int i=0;i<statementArr.length;i++)
		{
			if(statementArr[i].equalsIgnoreCase("*"))
			{
				left=(Node)stack.pop();
				int val=Integer.parseInt(statementArr[++i]);
				right=new ValueNode(val);
				stack.push(new MulNode(left,right));
			}
			else if(statementArr[i].equalsIgnoreCase("/"))
			{
				left=(Node)stack.pop();
				int val=Integer.parseInt(statementArr[++i]);
				right=new ValueNode(val);
				stack.push(new DivNode(left,right));
			}
			else if(statementArr[i].equalsIgnoreCase("%"))
			{
				left=(Node)stack.pop();
				int val=Integer.parseInt(statementArr[++i]);
				right=new ValueNode(val);
				stack.push(new ModNode(left,right));
			}
			else
			{
				stack.push(new ValueNode(Integer.parseInt(statementArr[i])));
			}
		}
		this.node=(Node)stack.pop();
```

```
    }

    public int compute()
    {
        return node.interpret();
    }
}
```

4. 参考类图如下所示：



其中，Node 充当抽象表达式角色，AndNode 和 OrNode 充当非终结符表达式角色，ValueNode 充当终结符表达式角色。代码如下所示：

```
import java.util.*;
interface Node
{
    public int interpret();
}

class ValueNode implements Node
{
    private int value;
    public ValueNode(int value)
    {   this.value=value;   }

    public int interpret()
    {   return this.value;   }
}

class AndNode implements Node
{
    private Node left;
```

```
	private Node right;
	public AndNode(Node left,Node right)
	{
		this.left = left;
		this.right = right;
	}

	public int interpret()
	{
		if(left.interpret()==1&&right.interpret()==1)
		{
			return 1;
		}
		else
		{
			return 0;
		}
	}
}

class OrNode implements Node
{
	private Node left;
	private Node right;
	public OrNode(Node left,Node right)
	{
		this.left = left;
		this.right = right;
	}

	public int interpret()
	{
		if(left.interpret()==1||right.interpret()==1)
		{
			return 1;
		}
		else
		{
			return 0;
		}
	}
}

class Context
```

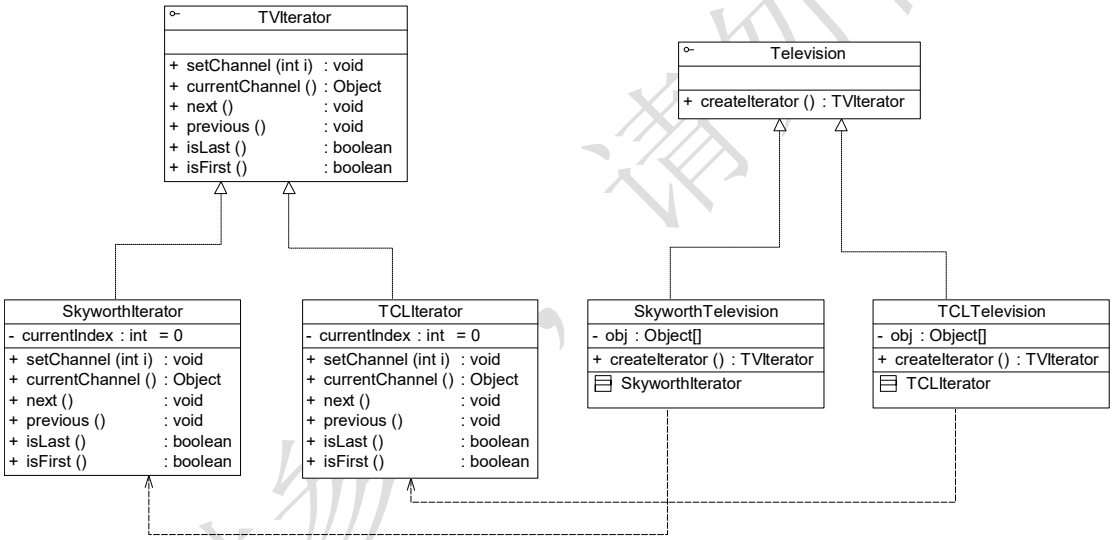
```
{
    private String statement;
    private Node node;

    public void parse(String statement)
    {
        Node left=null,right=null;
        Stack stack=new Stack();
        String[] statementArr=statement.split(" "); //分割输入字符串
        for(int i=0;i<statementArr.length;i++)
        {
            if(statementArr[i].equalsIgnoreCase("and"))
            {
                left=(Node)stack.pop();
                int val=Integer.parseInt(statementArr[++i]);
                right=new ValueNode(val);
                stack.push(new AndNode(left,right));
            }
            else if(statementArr[i].equalsIgnoreCase("or"))
            {
                left=(Node)stack.pop();
                int val=Integer.parseInt(statementArr[++i]);
                right=new ValueNode(val);
                stack.push(new OrNode(left,right));
            }
            else
            {
                stack.push(new ValueNode(Integer.parseInt(statementArr[i])));
            }
        }
        this.node=(Node)stack.pop();
    }

    public int execute()
    {
        return node.interpret();
    }
}

class Test
{
    public static void main(String args[])
    {
        String statement = "0 or 1 and 1 or 1";
```

```
            Context ctx = new Context();
            ctx.parse(statement);
            int result = ctx.execute();
            System.out.println(statement + " = " + result);
        }
}
//输出结果如下：
//0 or 1 and 1 or 1 = 1
```

5. 解答思路：首先提取指令中的关键词和变量，形成文法规则；然后为每一条文法规则设计一个表达式类。可采用与 P257 页类似的方法来设计并实现本习题。

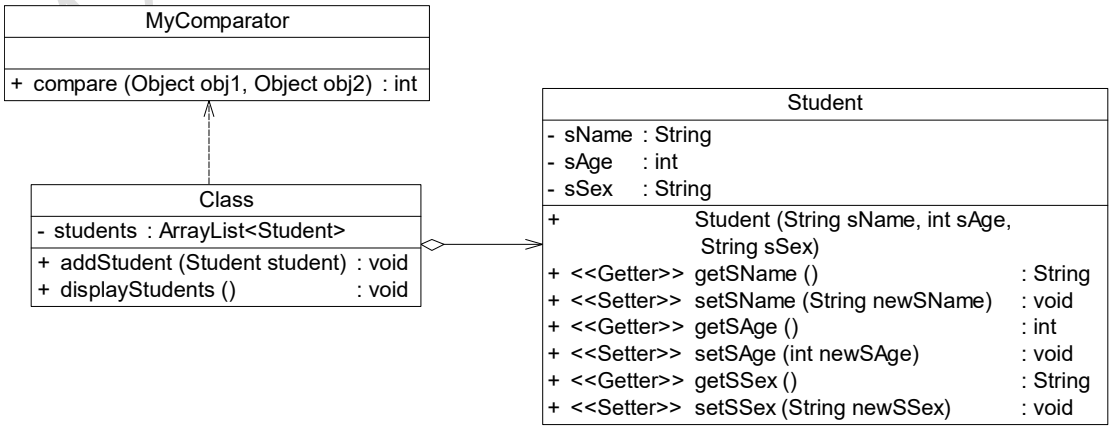
第 19 章 迭代器模式

1. B
2. D
3. C
4. 参考类图如下所示：



其中，TVIterator 充当抽象迭代器，其子类 SkyworthIterator 和 TCLIterator 充当具体迭代器；Television 充当抽象聚合类，其子类 SkyworthTelevision 和 TCLTelevision 充当具体聚合类。

5. 参考类图如下所示：



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。

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;
        }
```

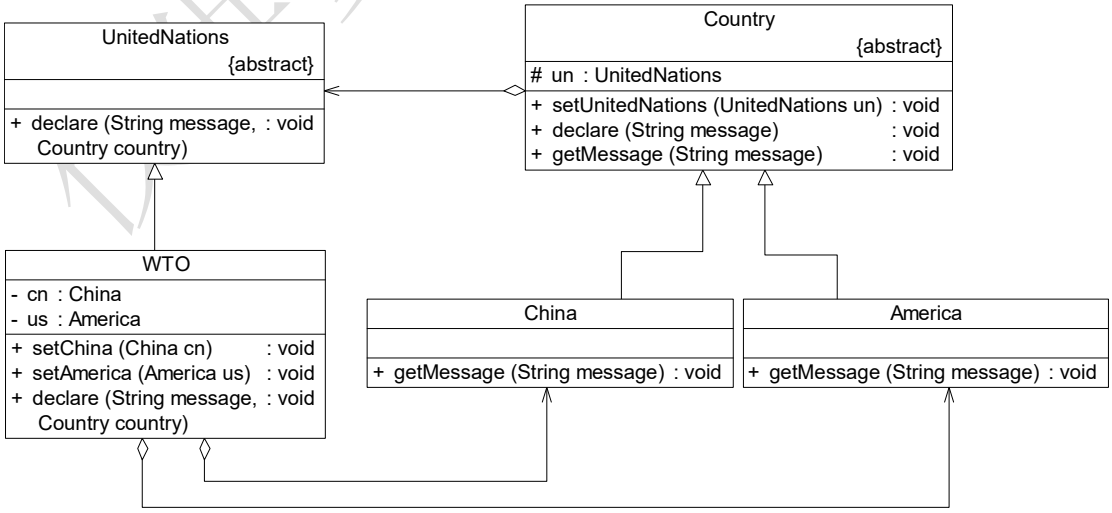
```
    }
}

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
```

6. 解答思路：逐页迭代器可以一次返回多条数据（一页数据），因此在其中需要定义一个返回集合类型的 Getter 方法，可以在初始化迭代器时指定页的大小（设置构造函数的参数），每次调用 Getter 方法时，返回一页数据，游标移至对应的位置，可以通过一个计数器来记录当前的页码。当最后剩余的数据不足一页时，返回全部数据。

第 20 章 中介者模式

1. C
2. A
3. B
4. 参考类图如下所示：



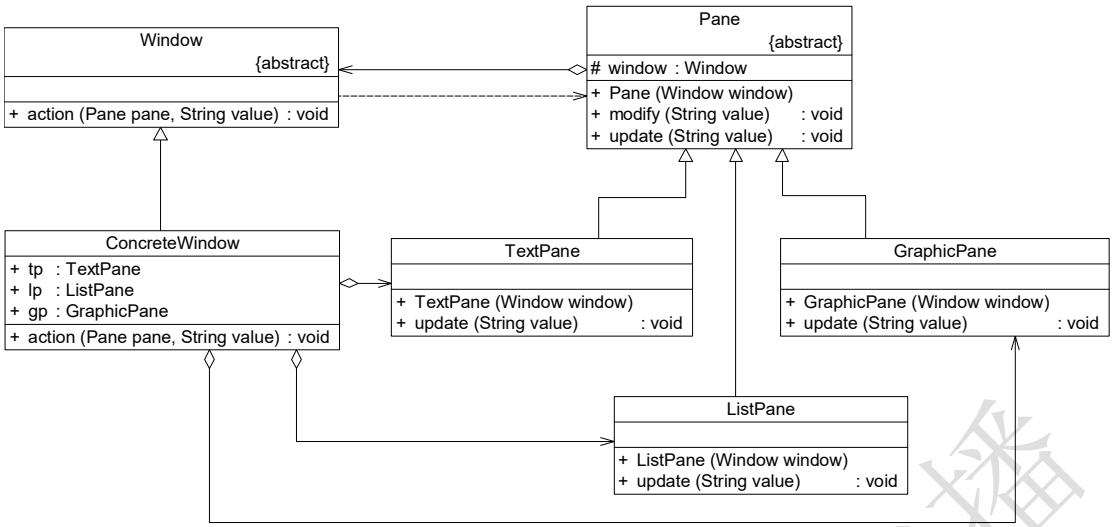
其中，UnitedNations 充当抽象中介者角色，WTO 充当具体中介者角色，Country 充当抽象同事角色，China 和 America 充当具体同事角色。

代码如下所示：

```
abstract class UnitedNations
{
    public abstract void declare(String message,Country country);
}

abstract class Country
{
    protected UnitedNations un;
    public void setUnitedNations(UnitedNations un)
    {
        this.un = un;
    }
    public void declare(String message)
    {
        un.declare(message,this);
    }
    public abstract void getMessage(String message);
}

class China extends Country
{
    public void getMessage(String message)
    {
        System.out.println("中国获取信息：" + message);
    }
}

class America extends Country
{
    public void getMessage(String message)
    {
        System.out.println("美国获取信息：" + message);
    }
}

class WTO extends UnitedNations
{
    private China cn;
    private America us;
    public void setChina(China cn)
    {
        this.cn = cn;
    }
```

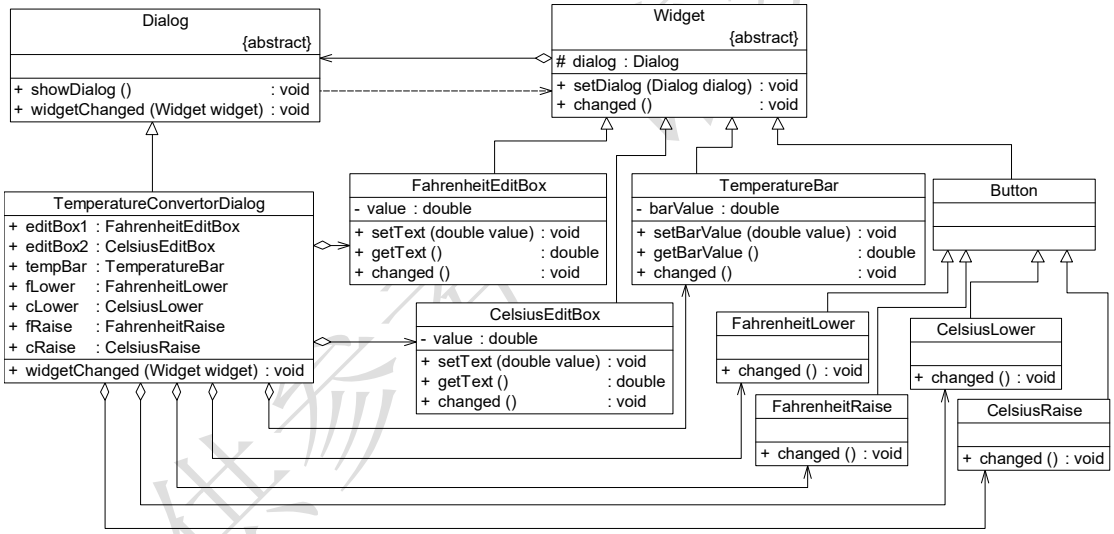
```
    public void setAmerica(America us)
    {
        this.us = us;
    }
    public void declare(String message,Country country)
    {
        if(country == cn)
        {
            us.getMessage(message);
        }
        else
        {
            cn.getMessage(message);
        }
    }
}

class MainClass
{
    public static void main(String args[])
    {
        WTO wto = new WTO();
        China cn = new China();
        America us = new America();
        cn.setUnitedNations(wto);
        us.setUnitedNations(wto);
        wto.setChina(cn);
        wto.setAmerica(us);
        cn.declare("中国是一个爱好和平的国家！");
        us.declare("美国将会为世界和平而努力！");
    }
}
//输出结果如下：
//美国获取信息：中国是一个爱好和平的国家！
//中国获取信息：美国将会为世界和平而努力！
```

5. 参考类图如下所示：



其中，抽象类 Window 充当抽象中介者，其中定义的 action()方法用于协调窗格之间的相互调用，ConcreteWindow 作为其子类充当具体中介者；抽象类 Pane 充当抽象同事类，包含改变方法 modify()和响应方法 update()，一个窗格的改变将引起其他窗格的响应，而且窗格与窗格之间不发生直接的相互引用。

6. 参考类图如下所示：



其中，Dialog 充当抽象中介者角色，TemperatureConvertorDialog 充当具体中介者角色，Widget 充当抽象同事角色，FahrenheitEditBox、CelsiusEditBox、TemperatureBar、FahrenheitLower、FahrenheitRaise、CelsiusLower 和 CelsiusRaise 充当具体同事角色。

代码如下所示：

```
//抽象窗口类：抽象中介者
abstract class Dialog
{
    public void showDialog()
    {
        System.out.println("显示主界面");
    }
    public abstract void widgetChanged(Widget widget);
}
```

```
//温度转换器窗口类：具体中介者
class TemperatureConvertorDialog extends Dialog
{
    public FahrenheitEditBox editBox1;
    public CelsiusEditBox editBox2;
    public TemperatureBar tempBar;
    public FahrenheitLower fLower;
    public CelsiusLower cLower;
    public FahrenheitRaise fRaise;
    public CelsiusRaise cRaise;
    public void widgetChanged(Widget widget)
    {
        if(widget==editBox1)   //华氏温度文本框
        {
            double value = editBox1.getText();
            double temp = (value - 32) * 5 /9;
            editBox2.setText(temp);
            tempBar.setBarValue(temp);
        }
        else if(widget==editBox2)   //摄氏温度文本框
        {
            double value = editBox2.getText();
            double temp = 9 * value /5 +32;
            editBox1.setText(temp);
            tempBar.setBarValue(value);
        }
        else if(widget==tempBar)   //温度调节条
        {
            double value = tempBar.getBarValue();
            double temp = 9 * value /5 +32;
            editBox1.setText(temp);
            tempBar.setBarValue(value);
        }
        else if(widget==fLower)   //华氏温度降低按钮
        {
            double temp1 = editBox1.getText() - 1;
            editBox1.setText(temp1);
            double temp2 = (temp1 - 32) * 5 /9;
            editBox2.setText(temp2);
            tempBar.setBarValue(temp2);
        }
        else if(widget==fRaise)   //华氏温度升高按钮
        {
```

```
                double temp1 = editBox1.getText() + 1;
                editBox1.setText(temp1);
                double temp2 = (temp1 - 32) * 5 /9;
                editBox2.setText(temp2);
                tempBar.setBarValue(temp2);
            }
            else if(widget==cLower)     //摄氏温度降低按钮
            {
                double temp1 = editBox2.getText() - 1;
                editBox2.setText(temp1);
                tempBar.setBarValue(temp1);
                double temp2 = 9 * temp1 /5 +32;
                editBox1.setText(temp2);
            }
            else if(widget==cRaise)    //摄氏温度升高按钮
            {
                double temp1 = editBox2.getText() + 1;
                editBox2.setText(temp1);
                tempBar.setBarValue(temp1);
                double temp2 = 9 * temp1 /5 +32;
                editBox1.setText(temp2);
            }
        }
}

//抽象窗口部件类：抽象同事类
abstract class Widget
{
        protected Dialog dialog;
        public void setDialog(Dialog dialog)
        {
                this.dialog = dialog;
        }
        public abstract void changed();
}

//华氏温度文本框：具体同事类
class FahrenheitEditBox extends Widget
{
        private double value = 50;
        public void setText(double value)
        {
                this.value = value;
                System.out.println("华氏温度设置为" + this.value + "。");
```

```
    }
    public double getText()
    {
        System.out.println("获取文本框中的华氏温度：" + this.value + "。");
        return this.value;
    }
    public void changed()
    {
        System.out.println("华氏温度文本框值改变：");
        dialog.widgetChanged(this);
    }
}

//摄氏温度文本框：具体同事类
class CelsiusEditBox extends Widget
{
    private double value = 10;
    public void setText(double value)
    {
        this.value = value;
        System.out.println("摄氏温度设置为" + this.value + "。");
    }
    public double getText()
    {
        System.out.println("获取文本框中的摄氏温度：" + this.value + "。");
        return this.value;
    }
    public void changed()
    {
        dialog.widgetChanged(this);
    }
}

//温度调节条：具体同事类
class TemperatureBar extends Widget
{
    private double barValue = 10;
    public void setBarValue(double value)
    {
        this.barValue = value;
        System.out.println("温度调节条值为" + this.barValue + "摄氏度。");
    }
    public double getBarValue()
    {
```

```
            System.out.println("获取温度调节条的摄氏温度：" + this.barValue + "。");
            return this.barValue;
        }
        public void changed()
        {
            dialog.widgetChanged(this);
        }
}

//按钮类：同事类
abstract class Button extends Widget
{
}

//华氏温度降低按钮：具体同事类
class FahrenheitLower extends Button
{
        public void changed()
        {
            System.out.println("点击华氏温度降低按钮：");
            dialog.widgetChanged(this);
        }
}

//华氏温度升高按钮：具体同事类
class FahrenheitRaise extends Button
{
        public void changed()
        {
            System.out.println("点击华氏温度升高按钮：");
            dialog.widgetChanged(this);
        }
}

//摄氏温度降低按钮：具体同事类
class CelsiusLower extends Button
{
        public void changed()
        {
            System.out.println("点击摄氏温度降低按钮：");
            dialog.widgetChanged(this);
        }
}
```

```
//摄氏温度升高按钮：具体同事类
class CelsiusRaise extends Button
{
    public void changed()
    {
        System.out.println("点击摄氏温度升高按钮：");
        dialog.widgetChanged(this);
    }
}
```

客户端测试代码如下所示：

```
//客户端测试类
class Client
{
    public static void main(String args[])
    {
        TemperatureConvertorDialog dialog;
        dialog = new TemperatureConvertorDialog();
        FahrenheitEditBox editBox1 = new FahrenheitEditBox();
        CelsiusEditBox editBox2 = new CelsiusEditBox();
        TemperatureBar tempBar = new TemperatureBar();
        FahrenheitLower fLower = new FahrenheitLower();
        CelsiusLower cLower = new CelsiusLower();
        FahrenheitRaise fRaise = new FahrenheitRaise();
        CelsiusRaise cRaise = new CelsiusRaise();
        editBox1.setDialog(dialog);
        editBox2.setDialog(dialog);
        tempBar.setDialog(dialog);
        fLower.setDialog(dialog);
        cLower.setDialog(dialog);
        fRaise.setDialog(dialog);
        cRaise.setDialog(dialog);
        dialog.showDialog();
        dialog.editBox1 = editBox1;
        dialog.editBox2 = editBox2;
        dialog.tempBar = tempBar;
        dialog.fLower = fLower;
        dialog.cLower = cLower;
        dialog.fRaise = fRaise;
        dialog.cRaise = cRaise;
        editBox1.changed();
        System.out.println("--------------------");
        fRaise.changed();
        System.out.println("--------------------");
        tempBar.setBarValue(20);
```

```
            tempBar.changed();
            System.out.println("--------------------");
        }
}
```

运行结果如下：

```
显示主界面
华氏温度文本框值改变：
获取文本框中的华氏温度：50.0。
摄氏温度设置为 10.0。
温度调节条值为 10.0 摄氏度。
--------------------
点击华氏温度升高按钮：
获取文本框中的华氏温度：50.0。
华氏温度设置为 51.0。
摄氏温度设置为 10.555555555555555。
温度调节条值为 10.555555555555555 摄氏度。
--------------------
温度调节条值为 20.0 摄氏度。
获取温度调节条的摄氏温度：20.0。
华氏温度设置为 68.0。
温度调节条值为 20.0 摄氏度。
--------------------
```

本实例中，在具体中介者类 TemperatureConvertorDialog 中封装了同事对象之间的相互调用。每一个同事对象都可以独立变化，在同事类中通过调用中介者的 widgetChanged()方法将自身对象传递给中介者，再在中介者的 widgetChanged()方法中对同事对象进行判断，以便调用其他同事对象相应的响应方法。在不修改现有同事类源代码的基础上，可以增加新的同事对象来接受响应，同事对象之间不产生直接的相互引用，从而降低了同事对象之间的耦合度。中介者模式可以简化对象之间的交互，将多个同事对象解耦。

第 21 章 备忘录模式

1. B
2. D
3. 可以使用原型模式来创建备忘录模式。在创建备忘录时可以通过克隆原发器对象来实现，即使用原型模式，此时原发器需要支持自我复制。为了简化系统设计，可以将原发器和备忘录合并，直接将克隆生成的原发器对象保存在负责人中。引入原型模式的原发器类的 Java 代码如下所示：

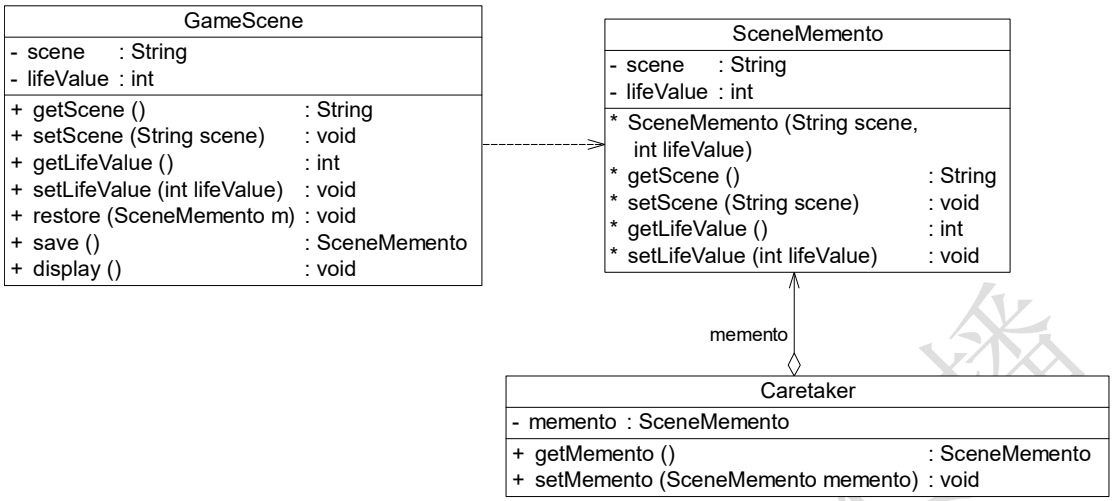
```
class Originator implements Cloneable, Serializable
{
    private String state;
    //省略 Getter 和 Setter 方法
    public Originator(){}

    //通过浅克隆创建一个备忘录对象
    public Originator createMementoC() {
```

```
        return (Originator)this.clone();
    }

    //通过深克隆创建一个备忘录对象
    public Originator createMementoDC(){
        try {
            return (Originator)this.deepClone();
        }
        catch(Exception e) {
            return null;
        }
    }

    public void restoreMemento(Originator m){
    state = m.getState();
    }

    //浅克隆
    public Object clone(){
        Object object = null;
        try {
            object = super.clone();
        }
        catch (CloneNotSupportedException exception) {
            System.err.println("Not support cloneable");
        }
        return object;
    }

    //深克隆
    public Object deepClone() throws IOException, ClassNotFoundException,
OptionalDataException {
        //将对象写入流中
        ByteArrayOutputStream bao=new ByteArrayOutputStream();
        ObjectOutputStream oos=new ObjectOutputStream(bao);
        oos.writeObject(this);

        //将对象从流中取出
        ByteArrayInputStream bis=new ByteArrayInputStream(bao.toByteArray());
        ObjectInputStream ois=new ObjectInputStream(bis);
        return(ois.readObject());
    }
}
```

4. 解决方案：将备忘录类作为原发器类的内部类，只有原发器才可以访问备忘录中的数据。

5. 解答思路：可使用 JDK 中内置的 Stack 类来实现栈，通过对栈顶元素进行操作可实现多次撤销和重做。

6. 参考类图如下所示：



其中，GameScene 充当原发器角色，它是待保存历史状态的类；SceneMemento 充当备忘录角色，它存储了 GameScene 的历史状态；Caretaker 充当负责人角色，它用于管理备忘录。

代码如下所示：

```
//游戏场景类：原发器
class GameScene
{
    private String scene;
    private int lifeValue;
    public void setScene(String scene)
    {
        this.scene = scene;
    }
    public void setLifeValue(int lifeValue)
    {
        this.lifeValue = lifeValue;
    }
    public String getScene()
    {
        return (this.scene);
    }
    public int getLifeValue()
    {
        return (this.lifeValue);
    }
    public void restore(SceneMemento m)
    {
        this.scene = m.getScene();
        this.lifeValue = m.getLifeValue();
```

```
    }
    public SceneMemento save()
    {
        return new SceneMemento(this.scene,this.lifeValue);
    }
    public void display()
    {
        System.out.print("当前游戏场景为：" + this.scene + "，");
        System.out.println("您还有" + this.lifeValue + "条命！");
    }
}

//场景备忘录：备忘录
class SceneMemento
{
    private String scene;
    private int lifeValue;
    SceneMemento(String scene,int lifeValue)
    {
        this.scene = scene;
        this.lifeValue = lifeValue;
    }
    void setScene(String scene)
    {
        this.scene = scene;
    }
    void setLifeValue(int lifeValue)
    {
        this.lifeValue = lifeValue;
    }
    String getScene()
    {
        return (this.scene);
    }
    int getLifeValue()
    {
        return (this.lifeValue);
    }
}

//负责人
class Caretaker
{
    private SceneMemento memento;
```

```
    public SceneMemento getSceneMemento()
    {
        return this.memento;
    }
    public void setSceneMemento(SceneMemento memento)
    {
        this.memento = memento;
    }
}
```

客户端测试代码如下所示：

```
//客户端测试类
class Client
{
    public static void main(String args[])
    {
        GameScene scene = new GameScene();
        Caretaker ct = new Caretaker();
        scene.setScene("无名湖");
        scene.setLifeValue(3);
        System.out.println("原始状态：");
        scene.display();
        ct.setSceneMemento(scene.save());
        System.out.println("------------------------");

        scene.setScene("魔鬼洞");
        scene.setLifeValue(0);
        System.out.println("牺牲状态：");
        scene.display();
        System.out.println("------------------------");

        scene.restore(ct.getSceneMemento());
        System.out.println("恢复到原始状态：");
        scene.display();
        System.out.println("------------------------");
    }
}
```

运行结果如下：

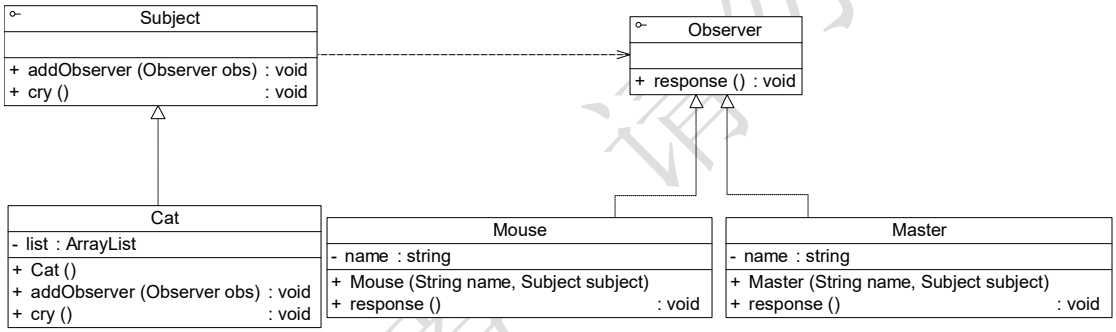
```
原始状态：
当前游戏场景为：无名湖，您还有 3 条命！
------------------------
牺牲状态：
当前游戏场景为：魔鬼洞，您还有 0 条命！
------------------------
恢复到原始状态：
```

```
当前游戏场景为：无名湖，您还有 3 条命！
```

在本实例中，原发器 GameScene 在调用 save()方法后将产生一个备忘录对象，该备忘录对象将保存在 Caretaker 中，原发器需要恢复状态时再将其从 Caretaker 中取出，可以通过调用 restore()方法来获取存储在备忘录中的状态信息。在真实开发中，除了原发器可以创建备忘录并给备忘录赋值外，其他对象不应该直接调用备忘录中的方法，也不能创建备忘录。如果备忘录被除原发器之外的第三者改动，则原发器恢复到的历史状态不是真实的历史状态，而是修改过的历史状态，这违背了备忘录模式的设计初衷，因此需要对备忘录进行封装。在 C++中可以将 Memento 类作为 Originator 类的友元类；而在 Java 语言中，一般将 Memento 类与 Originator 类定义在同一个 package 包中来实现封装，可使用默认访问标识符来定义 Memento 类，使其包内可见，只有 Originator 类可以对它进行访问，限制其他类对 Memento 的访问。

第 22 章 观察者模式

1. D
2. A
3. C
4. 参考类图如下所示：



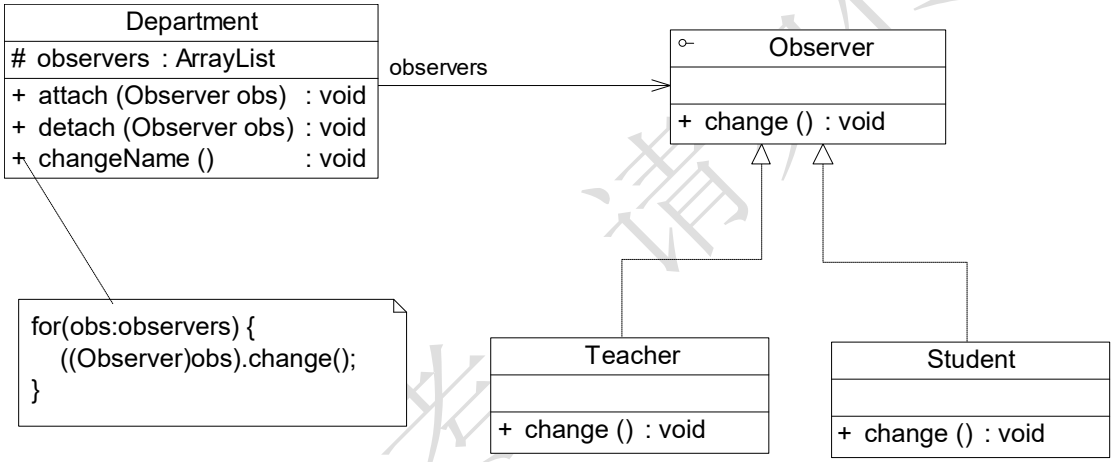
参考代码如下：

```
import java.util.*;
interface Subject       //抽象主题
{
      public void addObserver(Observer obs);
      public void cry();
}

interface Observer       //抽象观察者
{
      public void response();
}

class Cat implements Subject       //具体主题
{
      private ArrayList<Observer> list;
      public Cat()
      {
            list = new ArrayList<Observer>();
```

```
    }
    public void addObserver(Observer obs)
    {
        list.add(obs);
    }
    public void cry()
    {
        for(Object obj : list)
        {
            ((Observer)obj).response();
        }
    }
}

class Mouse implements Observer       //具体观察者
{
    private String name;
    public Mouse(String name, Subject subject)
    {
        this.name = name;
        subject.addObserver(this);
    }
    public void response()
    {
        System.out.println(this.name + "拼命逃跑！");
    }
}

class Master implements Observer        //具体观察者
{
    private String name;
    public Master(String name, Subject subject)
    {
        this.name = name;
        subject.addObserver(this);
    }
    public void response()
    {
        System.out.println(this.name + "从美梦中惊醒！");
    }
}

class Client       //客户端测试类
{
```

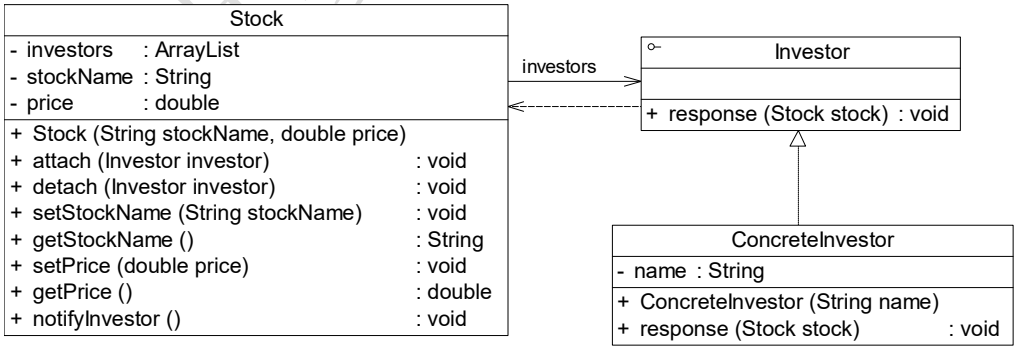
```
    public static void main(String args[])
    {
        Subject cat = new Cat();
        Observer mouse1,mouse2,master;
        mouse1 = new Mouse("大老鼠",cat);
        mouse2 = new Mouse("小老鼠",cat);
        master = new Master("小龙女",cat);
        cat.cry();
    }
}
//输出结果如下：
//大老鼠拼命逃跑！
//小老鼠拼命逃跑！
//小龙女从美梦中惊醒！
```

5. 参考类图如下所示：



其中，Department 充当观察目标，Observer 是抽象观察者，其子类 Teacher 和 Student 是具体观察者。

6. 参考类图如下所示：



其中，Stock 充当观察目标角色（省略抽象观察目标），Investor 充当抽象观察者，ConcreteInvestor 充当具体观察者。

参考代码如下：

```
import java.util.*;

//抽象股民：抽象观察者
```

```
interface Investor
{
	public void response(Stock stock);
}

//股票：观察目标
class Stock
{
	private ArrayList<Investor> investors;
	private String stockName;
	private double price;
	public Stock(String stockName,double price)
	{
		this.stockName = stockName;
		this.price = price;
		investors = new ArrayList<Investor>();
	}
	public void attach(Investor investor)
	{
		investors.add(investor);
	}
	public void detach(Investor investor)
	{
		investors.remove(investor);
	}
	public void setStockName(String stockName)
	{
		this.stockName = stockName;
	}
	public String getStockName()
	{
		return this.stockName;
	}
	public void setPrice(double price)
	{
		double range = Math.abs(price-this.price)/this.price;
		this.price = price;
		if(range>=0.05)
		{
			this.notifyInvestor();
		}
	}
	public double getPrice()
	{
```

```
        return this.price;
    }
    public void notifyInvestor()
    {
        for(Object obj:investors)
        {
            ((Investor)obj).response(this);
        }
    }
}

//股民：具体观察者
class ConcreteInvestor implements Investor
{
    private String name;
    public ConcreteInvestor(String name)
    {
        this.name = name;
    }
    public void response(Stock stock)
    {
        System.out.print("提示股民: " + name);
        System.out.print("------股票： " + stock.getStockName());
        System.out.print("价格波动幅度超过 5%------") ;
        System.out.println("新价格是:"+ stock.getPrice() + "。");
    }
}
```

客户端测试代码如下所示：

```
//客户端测试类
class Client
{
    public static void main(String args[])
    {
        Investor investor1,investor2;
        investor1 = new ConcreteInvestor("杨过");
        investor2 = new ConcreteInvestor("小龙女");

        Stock haier = new Stock("青岛海尔",20.00);
        haier.attach(investor1);   //注册
        haier.attach(investor2);   //注册

        haier.setPrice(25.00);
    }
}
```

运行结果如下：

```
提示股民: 杨过------股票：青岛海尔价格波动幅度超过 5%------新价格是:25.0。
提示股民: 小龙女------股票：青岛海尔价格波动幅度超过 5%------新价格是:25.0。
```

在本实例中，股票 Stock 是股民的观察目标，每次调用其 setPrice()方法设置股票价格时，将对价格变化幅度进行判断，如果变化幅度大于 0.05，则调用通知方法 notifyInvestor()来通知所有购买该股票的股民，股民在接收到通知后将执行 response()方法作出响应。

7. 参考代码如下：

```
//LoginEvent：事件类
import java.util.EventObject;

public class LoginEvent extends EventObject
{
    private String userName;
    private String password;
    public LoginEvent(Object source,String userName,String password)
    {
        super(source);
        this.userName=userName;
        this.password=password;
    }
    public void setUserName(String userName)
    {
        this.userName=userName;
    }
    public String getUserName()
    {
        return this.userName;
    }
    public void setPassword(String password)
    {
        this.password=password;
    }
    public String getPassword()
    {
        return this.password;
    }
}

//LoginEventListener（登录事件监听器）：抽象观察者
import java.util.EventListener;

public interface LoginEventListener extends EventListener
{
    public void validateLogin(LoginEvent event);  //声明响应方法
```

```
}

//LoginBean（登录控件类）：具体目标类
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class LoginBean extends JPanel implements ActionListener
{
    JLabel labUserName,labPassword;
    JTextField txtUserName;
    JPasswordField txtPassword;
    JButton btnLogin,btnClear;

    LoginEventListener lel;  //定义一个抽象观察者对象

    LoginEvent le;  //定义一个事件对象用于传输数据

    public LoginBean()
    {
        this.setLayout(new GridLayout(3,2));
        labUserName=new JLabel("User Name:");
        add(labUserName);

        txtUserName=new JTextField(20);
        add(txtUserName);

        labPassword=new JLabel("Password:");
        add(labPassword);

        txtPassword=new JPasswordField(20);
        add(txtPassword);

        btnLogin=new JButton("Login");
        add(btnLogin);

        btnClear=new JButton("Clear");
        add(btnClear);

        btnClear.addActionListener(this);
        btnLogin.addActionListener(this);
    }

    //实现注册方法
```

```
    public void addLoginEventListener(LoginEventListener lel)
    {
        this.lel=lel;
    }

    //实现通知方法
    private void fireLoginEvent(Object object,String userName,String password)
    {
        le=new LoginEvent(btnLogin,userName,password);
        lel.validateLogin(le);
    }

    public void actionPerformed(ActionEvent event)
    {
        if(btnLogin==event.getSource())
        {
            String userName=this.txtUserName.getText();
            String password=this.txtPassword.getText();

            fireLoginEvent(btnLogin,userName,password);
        }
        if(btnClear==event.getSource())
        {
            this.txtUserName.setText("");
            this.txtPassword.setText("");
        }
    }
}

//LoginValidatorA（登录界面类）：具体观察者类
import javax.swing.*;
import java.awt.*;

public class LoginValidatorA extends JFrame implements LoginEventListener
{
    private JPanel p;
    private LoginBean lb;   //定义具体目标
    private JLabel lblLogo;
    public LoginValidatorA()
    {
        super("Bank of China");
        p=new JPanel();
        this.getContentPane().add(p);
        lb=new LoginBean();
```

```
        lb.addLoginEventListener(this);   //调用目标对象的注册方法

        Font f=new Font("Times New Roman",Font.BOLD,30);
        lblLogo=new JLabel("Bank of China");
        lblLogo.setFont(f);
        lblLogo.setForeground(Color.red);

        p.setLayout(new GridLayout(2,1));
        p.add(lblLogo);
        p.add(lb);
        p.setBackground(Color.pink);
        this.setSize(600,200);
        this.setVisible(true);
    }

    //实现在抽象观察者中声明的响应方法
    public void validateLogin(LoginEvent event)
    {
        String userName=event.getUserName();
        String password=event.getPassword();

        if(0==userName.trim().length()||0==password.trim().length())
        {
            JOptionPane.showMessageDialog(this,new String("Username or Password is
empty!"),"alert",JOptionPane.ERROR_MESSAGE);
        }
        else
        {
            JOptionPane.showMessageDialog(this,new String("Valid Login
Info!"),"alert",JOptionPane.INFORMATION_MESSAGE);
        }
    }
    public static void main(String args[])
    {
        new LoginValidatorA().setVisible(true);
    }
}

// LoginValidatorB（登录界面类）：具体观察者类
import javax.swing.*;
import java.awt.*;

public class LoginValidatorB extends JFrame implements LoginEventListener
{
```

```
    private JPanel p;
    private LoginBean lb;
    private JLabel lblLogo;

    public LoginValidatorB()
    {
        super("China Mobile");
        p=new JPanel();
        this.getContentPane().add(p);
        lb=new LoginBean();
        lb.addLoginEventListener(this);

        Font f=new Font("Times New Roman",Font.BOLD,30);
        lblLogo=new JLabel("China Mobile");
        lblLogo.setFont(f);
        lblLogo.setForeground(Color.blue);

        p.setLayout(new GridLayout(2,1));
        p.add(lblLogo);
        p.add(lb);
        p.setBackground(new Color(163,185,255));
        this.setSize(600,200);
        this.setVisible(true);
    }

    public void validateLogin(LoginEvent event)
    {
        String userName=event.getUserName();
        String password=event.getPassword();

        if(userName.equals(password))
        {
            JOptionPane.showMessageDialog(this,new String("Username must be different from password!"),"alert",JOptionPane.ERROR_MESSAGE);
        }
        else
        {
            JOptionPane.showMessageDialog(this,new String("Rigth details!"),"alert",JOptionPane.INFORMATION_MESSAGE);
        }
    }

    public static void main(String args[])
    {
```

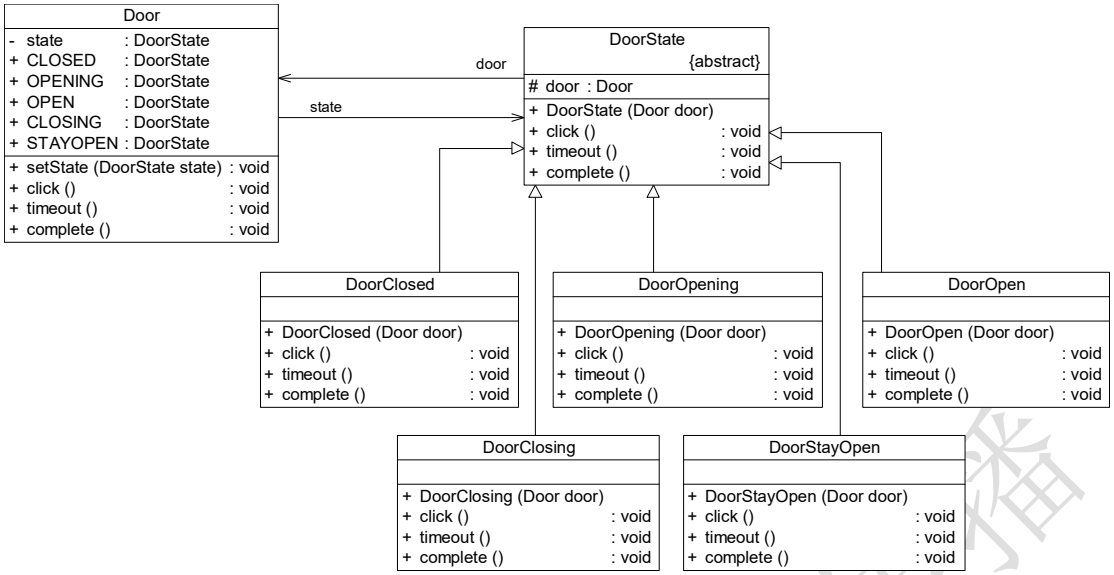
```
            new LoginValidatorB().setVisible(true);
        }
}
```

图 1 LoginValidatorA 运行效果图

图 2 LoginValidatorB 运行效果图

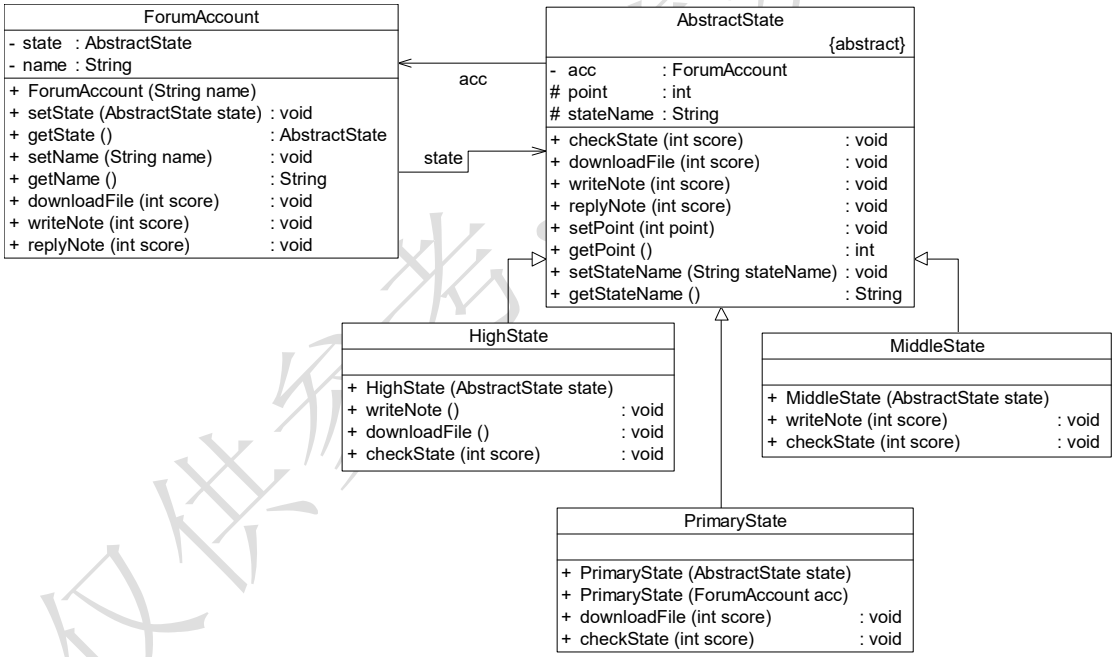
第 23 章 状态模式

1. D
2. C
3. D
4. 参考类图如下所示：



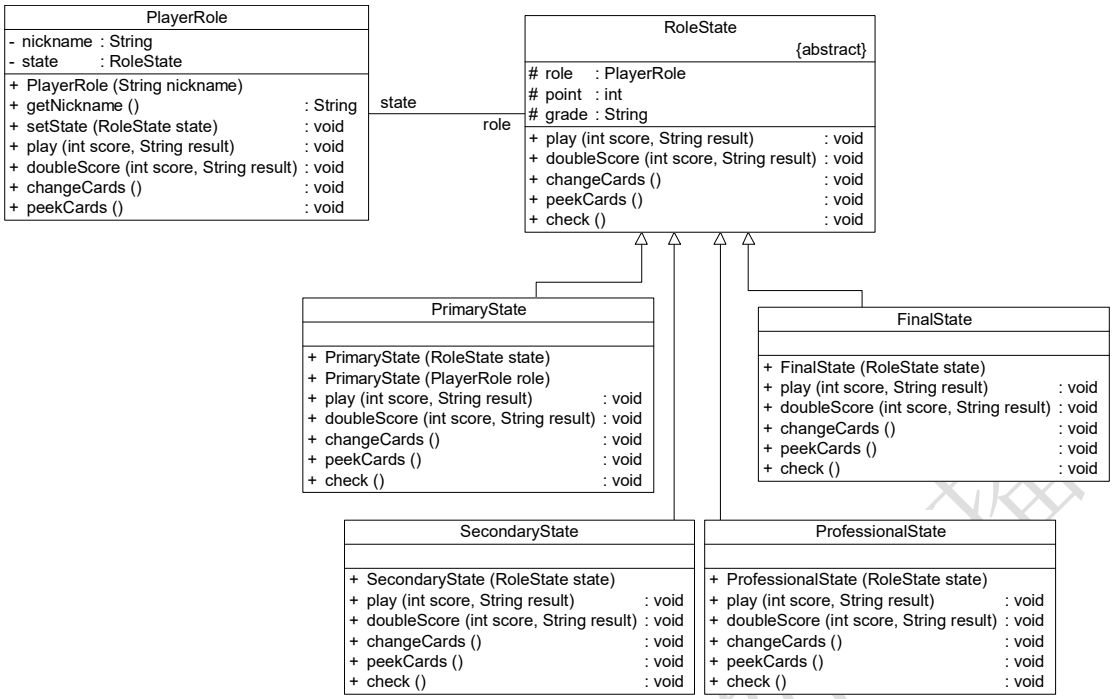
其中，Door 充当环境类角色，DoorState 充当抽象状态类角色，其子类 DoorClosed、DoorOpening、DoorOpen、DoorClosing 和 DoorStayOpen 充当具体状态类角色，分别对应传输门所具有的五种状态。

5. 参考类图如下所示：



其中，ForumAccount 充当环境类角色，AbstractState 充当抽象状态类角色，HighState、MiddleState 和 PrimaryState 充当具体状态类角色，分别对应专家状态、高手状态和新手状态。

6. 参考类图如下所示：



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 + "。");
        }
```

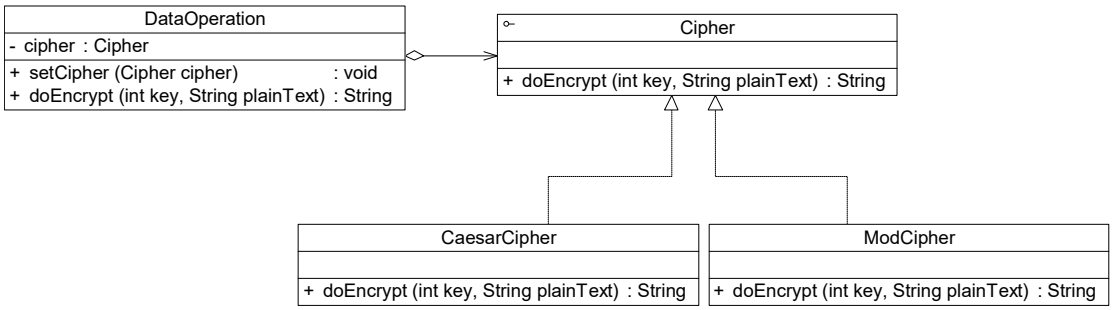
```
            else if(result.equalsIgnoreCase("lose"))   //失利
            {
                this.point -= score;
                System.out.println("玩家" + this.role.getNickname() + "失利，减少积分" +
score + "，当前积分为" + this.point + "。");
            }
            this.check();
        }
        public void 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
```

第 24 章 策略模式

1. B
2. B
3. A
4. 一个环境类可以对应多个不同的策略等级结构。在环境类中维持对每一个策略等级结构中抽象策略类的引用即可，在程序运行时再分别从每一个策略等级结构中选择一个具体策略

对象注入到环境类中。

5. 参考类图如下所示：

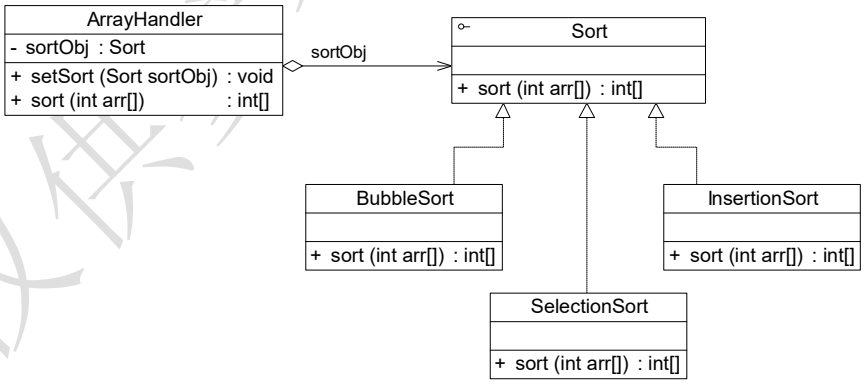


在本实例中，DataOperation 充当环境类角色，Cipher 充当抽象策略角色，CaesarCipher 和 ModCipher 充当具体策略角色。

其中，DataOperation 类的代码如下所示：

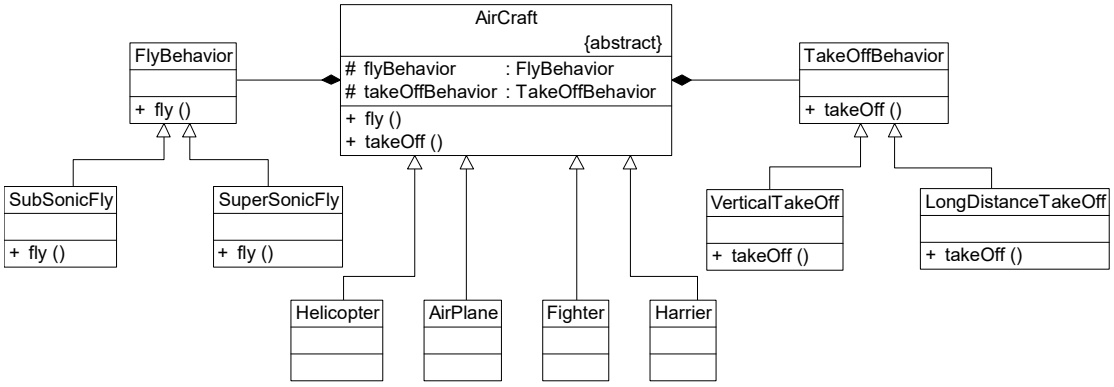
```
class DataOperation
{
    private Cipher cipher;
    public void setCipher(Cipher cipher)
    {
        this.cipher = cipher;
    }
    public String doEncrypt(int key, String plainText)
    {
        return cipher.doEncrypt(key,plainText);
    }
}
```

6. 参考类图如下所示：



其中，ArrayHandler 充当环境类角色，Sort 充当抽象策略类，其子类 BubbleSort、SelectionSort 和 InsertionSort 是具体策略类。

7. 参考类图如下所示：



其中，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。

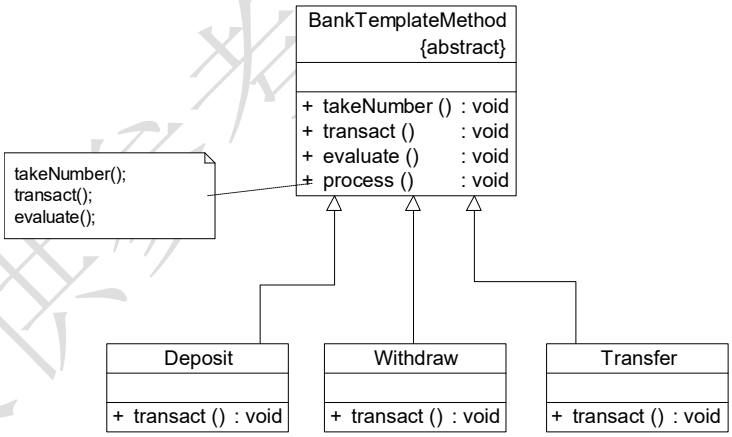
第 25 章 模板方法模式

1. C

2. B

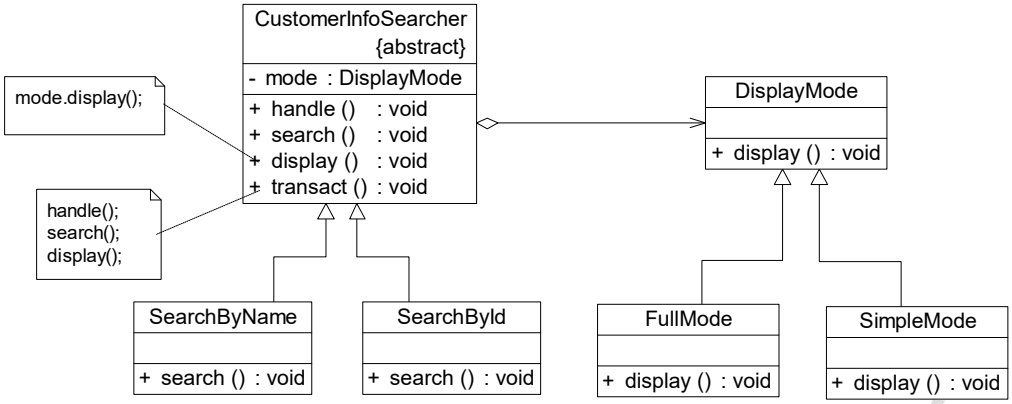
3. 由于钩子方法通常返回一个 boolean 类型的值，并以此来判断是否执行某一基本方法，因此在子类中可以通过覆盖钩子方法来决定是否执行父类中的某一方法，从而实现子类对父类行为的控制。

4. 参考类图如下所示：



其中，BankTemplateMethod 充当抽象类，Deposit、Withdraw 和 Transfer 充当其具体子类，process()方法是模板方法，定义了其他基本方法的执行次序。

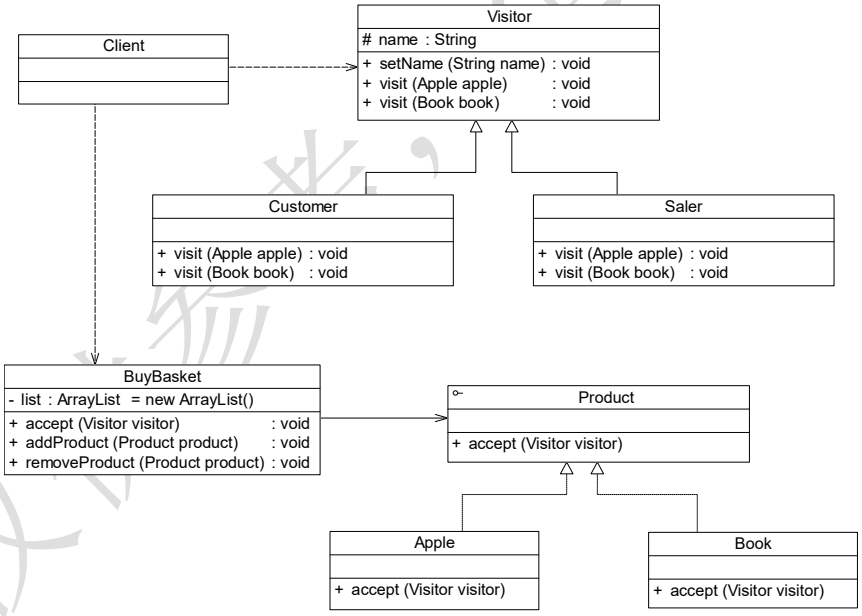
5. 参考类图如下所示：



其中，CustomerInfoSearcher 充当抽象类，SearchByName 和 SearchById 充当具体子类，将显示模式提取为一个独立的继承接口，在 CustomerInfoSearcher 和抽象显示模式 DisplayMode 之间建立抽象耦合关系，本实例引入了桥接模式，方便增加新的查询方式和新的查询结果显示模式。

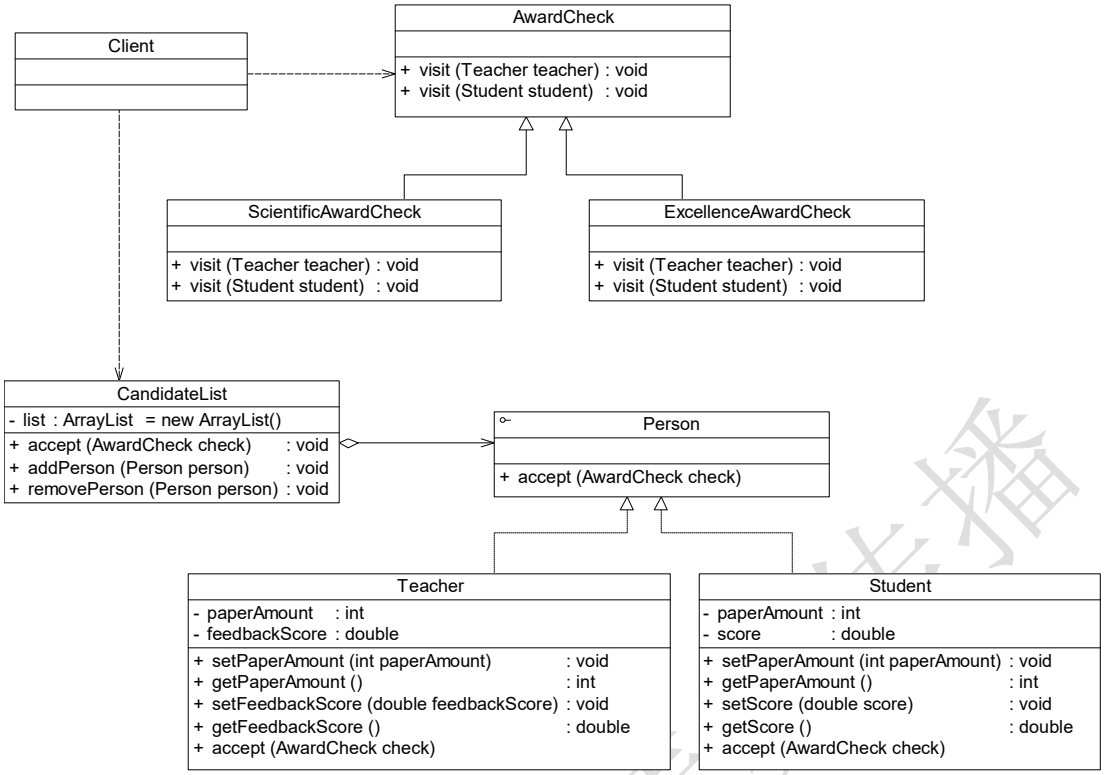
第 26 章 访问者模式

1. A
2. D
3. “双重分派”机制参见教材 P380。
4. 参考类图如下所示：



其中，Visitor 充当抽象访问者角色，Customer 和 Saler 充当具体访问者角色，Product 充当抽象元素角色，Apple 和 Book 充当具体元素角色，BuyBasket 作为对象结构。

5. 参考类图如下所示：



其中，AwardCheck 充当抽象访问者角色，ScientificAwardCheck 和 ExcellenceAwardCheck 充当具体访问者角色，Person 充当抽象元素角色，Teacher 和 Student 充当具体元素角色，CandidateList 作为对象结构。